Autodesk®
Scaleform®

Font and Text Configuration Overview

This document describes the font and text rendering system used in Scaleform 4.4, providing details on how to configure both the art assets and Scaleform C++ APIs for internationalization.

Authors: Maxim Shemanarev, Michael Antonov, Artem Bolgar

Version: 2.4
Last Edited: February 19, 2014

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.4

How to Contact Autodesk Scaleform:

Address	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Introduction: Fonts in Scaleform

Autodesk Scaleform ships with a flexible font system that provides high-quality transformable text with HTML formatting. Using the font system, fonts can be substituted for localization and taken from different sources, including locally embedded text, shared GFX/SWF files, operating system font data or the FreeType 2 library. The font rendering quality has also been significantly improved, allowing developers to trade off between animation performance, memory use and readability of text.

Many of the Scaleform font features rely on the functionality inherent in Flash® Studio, including the ability to embed font character sets into SWF files and to utilize system fonts when they are available. However, Flash Studio is primarily tailored to developing individual files and thus has limited font localization capabilities. In particular, Flash does not allow embedded fonts or translation tables to be easily shared across files, a feature that is critical for efficient memory use and game asset development. Furthermore, Flash relies on the operating system to handle font substitution of international characters that are not embedded in the file, which is not an option on game consoles where system fonts are not available.

Scaleform addresses these issues by using the GFx::Translator class as a centralized callback for all translatable text and the GFx::FontLib class for dynamic font mapping. Additional interfaces are provided to control the caching mechanism used for fonts, to substitute font names during localization, and to enable operating system and FreeType 2 font lookup when applicable.

In order to effectively use the Scaleform font system, developers need to understand the font features of both Flash Studio and Scaleform runtime. Knowing the Flash side will allow artists to develop consistent-looking content that can render efficiently with the desired quality, while minimizing the memory use. Understanding the Scaleform font runtime options will allow developers to select a configuration that has the best possible quality, performance and memory use characteristics for a given platform.

This document is designed to teach developers that are not intimately familiar with Flash and Scaleform the ins and outs of Game UI font setup. The remainder of the introduction is organized as follows:

- The “Flash Text and Font Basics” section introduces the fundamentals of using text and fonts in Flash. Developers familiar with Flash can skip the first two parts, which describe TextFields and character embedding.
- The “Font Decisions for a Game UI” section describes the fundamental font related decisions developers will need to make when creating a game UI in Flash. It is recommended that every developer read this section, as it describes the structure for the rest of the document.

1.1 Flash Text and Font Basics

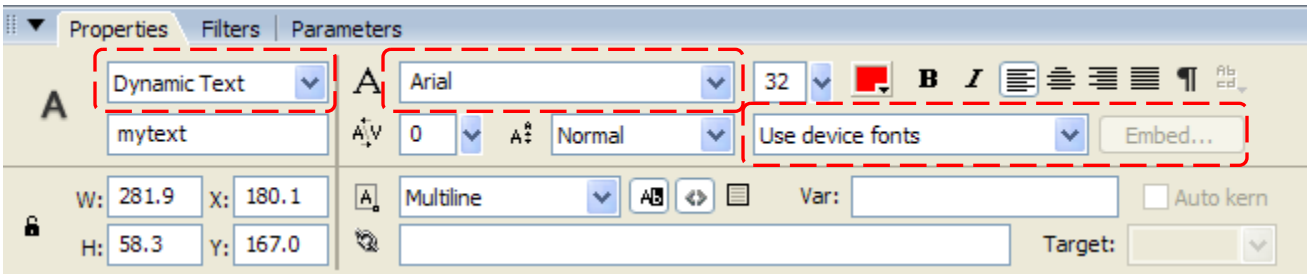
In Flash Studio artists create text fields visually by drawing them on stage with a mouse and then entering the corresponding text. The font applied to a text field is selected by name out of the list of fonts installed on the artists system, with an embedding option available for content playback on systems that do not have the required fonts.

In addition to using font names directly, artists can also create font symbols in a library and then use them indirectly based on a symbol name. Combined with character embedding, font symbols create a powerful conceptual framework for portable and consistent game UI asset development. The rest of this section briefly goes over the basics of text field creation and character embedding. Developers are encouraged to refer to the Flash Studio documentation for a more detailed explanation.

The Flash font symbol system is convenient and easy to setup; however, it has a number of limitations that make it hard to use for sharing and substituting fonts for internationalization. The details of how these limitations are addressed in Scaleform will be presented in Part 1 – Creating the Game Font Library.

1.1.1 Text Fields

Artists create text fields in Flash by first selecting the Text Tool and then drawing the corresponding rectangular regions on stage. The various attributes of text, including the text field type, font, size and style are configured in the Text Field properties panel, commonly located at the bottom of Flash Studio. This panel is illustrated below.



Although the text field can have many options, the critical attributes for this discussion are circled above. The most important option for a text field is text type in the top-left corner, which can be set to one of the following values:

- Static Text
- Dynamic Text
- Input Text

For game development the Dynamic Text property is going to be the most commonly used field type, as it supports content modification through ActionScript, as well as font substitution and internationalization through the user-installed GFx::Translator class. Static text does not possess any of those features and thus closely resembles vector art in functionality. Input text can be used whenever an editable textbox is needed.

The other two fields circled above are font name, in the top line of the property sheet, and font rendering method. The font name can be selected to use (a) any one of the fonts installed on the developer's system or (b) any one of the font symbol names created in the movies library. In the example above, "Arial" is selected for the font name. Also, the font style can be configured through the Bold and Italic toggle buttons.

For internationalization, the font rendering method is the most important setting, as it controls the character embedding in the SWF file. In Flash 8, it can be set to one of the following values:

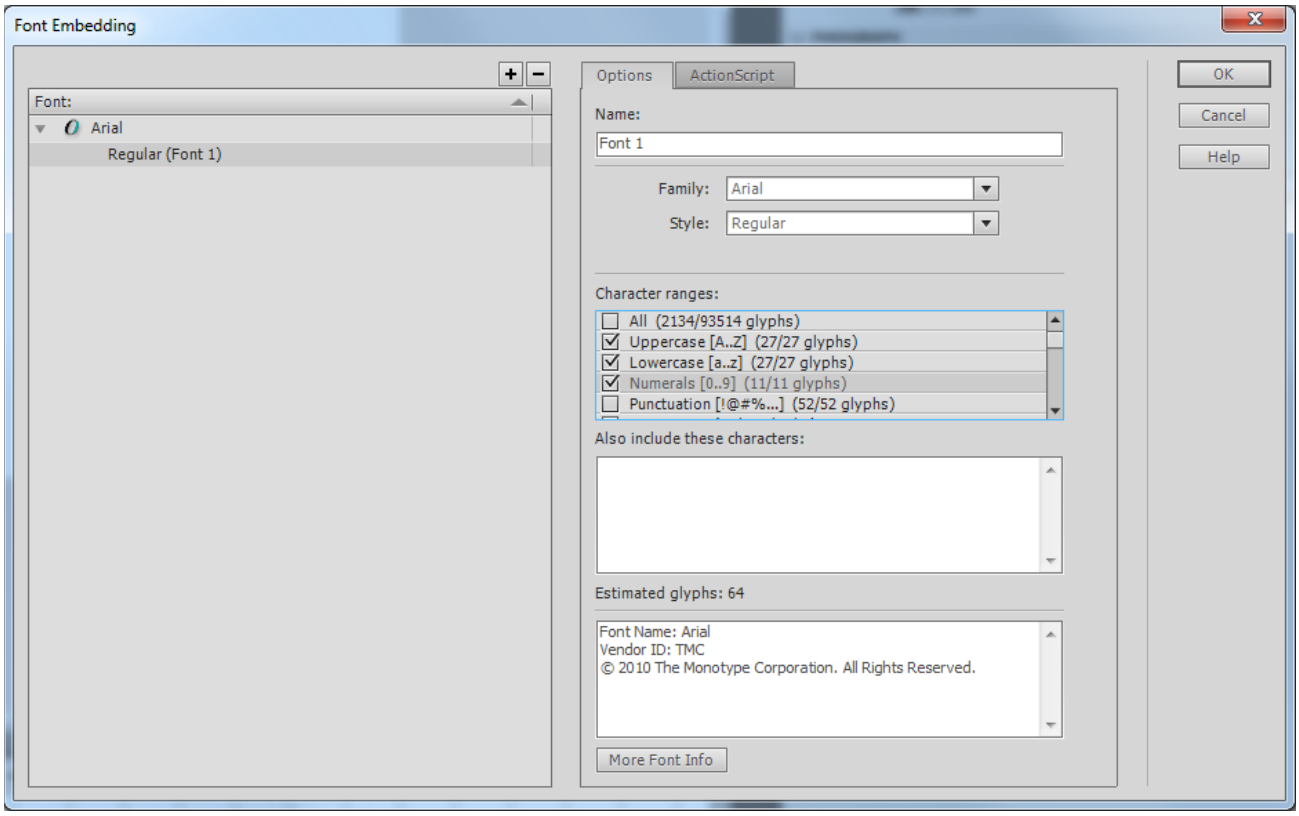
- Use device fonts
- Bitmap text (no anti-alias)
- Anti-alias for animation
- Anti-alias for readability

If you select "Use device fonts", a system-specific method of font rendering will be used, with font data taken from the operating system. One advantage of using device fonts is that the resulting SWF file will be small and use less system memory when played back. However, problems may occur if the target system does not have the requested font, in which case an alternative will be chosen by the Flash player. This alternative font may not look the same as the original and/or many not have all of the required glyphs. For instance, on a game console where there is no operating system font library, no text will be displayed. The unavailable glyphs will be rendered as squares in Scaleform.

Instead of using system fonts, the "Anti-alias for animation" and "Anti-alias for readability" settings rely on the embedded font characters, rendering them with animation performance and high quality optimizations, respectively. Whenever one of these options is selected, the Embed button becomes available, which allows users to select character ranges that will be embedded for the font.

1.1.2 Character Embedding

In order for embedded font rendering to work, characters must be embedded for the specified font in at least one TextField in the file (if font is exported then no textfields are required). If the necessary characters are not embedded, device fonts will be used with all of the limitations described above (unless GFx::FontLib is relied on in Scaleform as described later on). To embed font characters, users need to press the Embed button that will show the Character Embedding dialog illustrated below.



In the embedding dialog, users can select the desired character ranges that will be embedded for the text fields currently selected font and style. All text fields that use the same font style will share the same embedded character set, thus it is usually enough to specify embedding for only one of the text fields.

Users should be aware that from the point of view of embedding, bold and italic attributes of the font are treated independently (see "Style" combobox on a screenshot above). As an example, if both plain Arial and Arial Bold styles are used, they must both be embedded separately, increasing the memory footprint of the file. Changing font size, however, can be done without increasing the file size or memory overhead. For more details on fonts and character embedding, please refer to the Using Fonts section in Flash Studio documentation.

Embedding characters is the only truly portable way to use fonts. When font characters are embedded their vector representation is saved into the SWF/GFX file and later loaded into memory on use. Since glyph data is a part of the file, it will always render correctly, independently of the fonts installed on the system in use. In Scaleform, embedded fonts will work equally well on game consoles (such as Xbox 360, PS3, Wii) and desktop PCs (such as Windows, Mac, Linux). The unavoidable side effect of the embedded font use is the increased file size and memory use. Since the size increase can be significant, in particular for Asian languages, developers will need to plan game font use ahead of time and share fonts whenever possible.

1.1.3 Memory Use by Embedded Fonts

In order to understand the kind of memory effect character embedding has in Scaleform, consider the following table that outlines memory used as the number of embedded characters increases.

Embed Character Count	Uncompressed SWF Size	Scaleform Runtime Size
1 Character	1 KB	450 K
114 Chars – Latin + Punctuation	12 KB	480 K
596 Chars – Latin and Cyrillic	70 KB	630 K
7,583 Chars – Latin and Japanese	2,089 KB	3,500 K
18,437 Chars – Latin and Traditional Chinese	5,131 KB	8,000 K

The sample table was created for Scaleform 2.1 by embedding the "Arial Unicode MS" font. As outlined above, including European character sets is relatively inexpensive, with developers being able to include 500 characters for roughly 150K memory use increase. This is enough for several different font styles in localized distributions. With Asian languages, however, the amount of memory used becomes much greater due to the large number of glyphs involved.

1.1.4 Controlling Font Memory Use

Since embedding large character sets can consume several megabytes of memory, developers will need to plan their font use ahead of time to reduce the amount of memory used. There are several techniques that can be used to keep font memory under control:

1. *Limit fonts and font styles used to a pre-determined set before UI art assets are developed.* Bold and italic font styles are embedded as separate character sets in Flash, so you should treat them as entire separate fonts and use them only when necessary. Different font sizes, however, incur no additional overhead, so they can be used without causing memory increase. In the future, we will provide a fake bold and italic option that avoids requiring additional font characters to be stored for bold and italic versions.
2. *Share fonts across files by either using GFx::FontLib and/or importing.* Embedding the same characters in every single file may cause unreasonable increase in asset size, memory use (if the later files are simultaneously loaded) and load time. When possible, it is best to store embedded fonts into separate SWF/GFX files which are shared and thus do not need to be duplicated in memory or reloaded. The use of GFx::FontLib will be discussed later in this document.
3. *Only embed the used characters for Asian character sets.* In Asian localized games it is possible to avoid embedding all characters in the language and instead embed only those characters that are used in the game text. After translation, all of the game strings can be scanned to generate the required unique character set, which is then embedded. As long as the game does not require

arbitrary dynamic text input through IME, limiting the character set will result in significant space savings.

4. *Consider using GFxEport font compession (-fc option).* Generally 10%-30% size reduction can be achieved without noticeable reduction in quality (see section 5.3).

For a best looking internationalized game, users may choose to combine all of these techniques, limiting the fonts used and sharing them through GFx::FontLib. For Asian languages, embedding only the characters used or if IME is required, embedding only one complete font, would result in substantial memory savings.

1.2 Font Decisions for a Game UI

When creating a game user interface in Scaleform, developers will need to make a number of decisions related to font use, configuration, and internationalization approach. In this document, these decisions are broken down into four separate categories:

- *Part 1: Creating the Game Font Library* - helps you decide on the number and style of fonts to use when creating the font library.
- *Part 2: Selecting the Internationalization Approach* – describes the different approaches to creating the internationalization friendly game assets.
- *Part 3: Configuring Font Sources* – describes the order of font lookup and how the different font sources available in Scaleform can be configured.
- *Part 4: Configuring Font Rendering* – describes the Scaleform text rendering approaches and the different configuration options they have.

Creating the Game Font Library refers to choosing the fonts styles that the game interface should rely on, a step that should normally take place before the game UI assets are developed. On the art side, using a standard font library is important to make sure that different fonts are used consistently for the same purpose in all of the UI files. On the technical side, it is important to keep the number of embedded fonts in check so that they fit into the applications memory budget.

Once the game fonts are determined, it is time to decide how they are going to be internationalized. Part 2 discusses three possible models for internationalization:

- *Imported font substitution*, relies on imports and separate language specific font files loaded into the player and shared through GFx::FontLib.
- *Device font emulation*, is similar to the above but relies on device fonts instead of imported fonts, and requires the appropriate underlying system font support (not available on some platforms).

- *Custom asset generation*, where Flash-built translation facilities are used to generate custom versions of the SWF/GFX files for each target market.

Developers can select one or a well-understood mix of these approaches and follow it throughout development.

Since fonts can come from a number of sources in Scaleform, correct configuration is very important. To help with this font configuration, Part 3: Configuring Font Sources describes the Scaleform font lookup process in detail, including the examples of how GFx::FontLib and font providers are set up. Font providers are user-configurable classes used to support system fonts and FreeType 2 fonts, which can be used instead of GFx::FontLib to access fonts without embedding.

The Part 4 of this document, Configuring Font Rendering, describes the font rendering approaches used in Scaleform and focuses on the three font texture initialization choices available: pre-rendering textures during preprocessing by gfxexport, packing static textures generated at load time, and using a dynamic texture cache that is updated on demand. The document discusses the configuration options available for each approach allowing developers to pick the right one for the application and target platform.

2 Part 1: Creating the Game Font Library

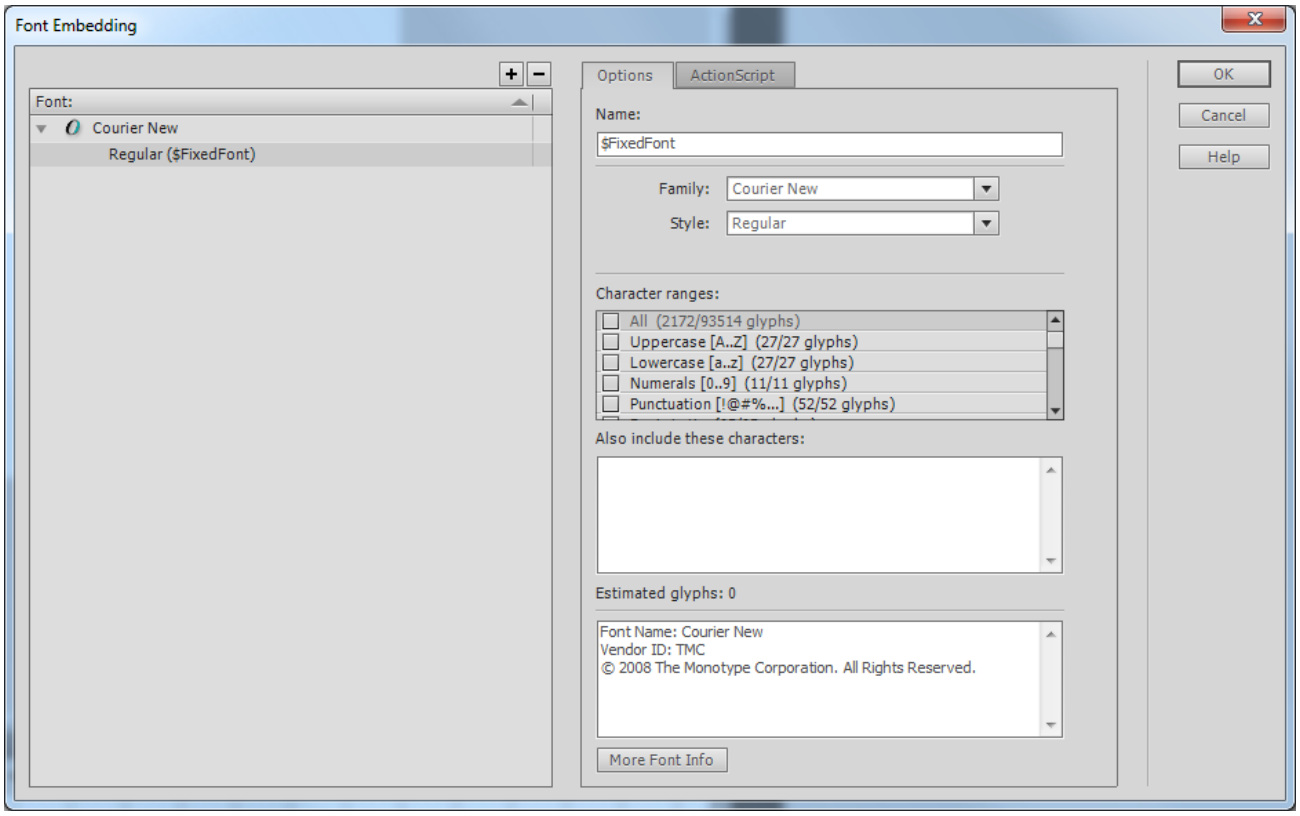
One of the first things artists should do when starting to work on a game UI is identify a set of potential font styles that will be used throughout the game. They can decide, for example, that there will be a unique font type for all titles, while another type is used for the rest of the text in the game. After this decision is made, additional font types should not be used in any of the game's UI screens without very careful consideration. There are three main reasons for enforcing such a limitation:

1. By using the same fonts, artists make sure that all of the game content appears consistent to the user. If different game screens used different fonts, they would produce a disorganized UI that is harder to read and comprehend.
2. During internationalization, development fonts often need to be substituted to different fonts that have characters for the target language. Having a predetermined fixed set of fonts allows this to be done easily.
3. Having a fixed set of fonts that is shared through many UI screens allows font data to be shared in memory, significantly reducing memory use and reducing screen load times. Since font data can occupy a lot of memory, this is a very important technical consideration.

Under the Scaleform imported font substitution approach described in the next part, the process of selecting a font set for a game is formalized into creation of a font library consisted of a set of files, one per language (" e.g. fonts_en.swf", "fonts_kr.swf", etc). Artists can create these file by creating named font symbols in the Flash file library and then exporting them into a corresponding SWF. After the library files are created, their font symbols can be imported into the other Flash files and used during development. The upcoming sections describe the details of font symbol creation and how they can be used to create a game font library.

2.1 Font Symbols

In the introduction we described how fonts can be applied to text fields and how their characters can be embedded for portable playback. In addition to using system fonts in TextField properties, Flash Studio allows artists to define new font symbols by right-clicking on the library and selecting the "New Font…" item, which displays the following dialog:



Font symbols created this way are added to the library of the FLA file and can later be applied to text fields similar to regular system fonts. For example, if you named your game font "$FixedFont", you should be able to select that name as a valid font for your TextField.

Adding font names to the library has a benefit of identifying the fonts used in a unique way. When library fonts are displayed in the font list, they will have a star next to them, as in "$ArialGame*". If artists only use font names from the library, it will help ensure that other fonts are not referenced in the UI files by mistake. Furthermore, if all fonts' symbols are named starting with the dollar sign character '$', they will always show up at the top of the font list in TextField properties, helping development.

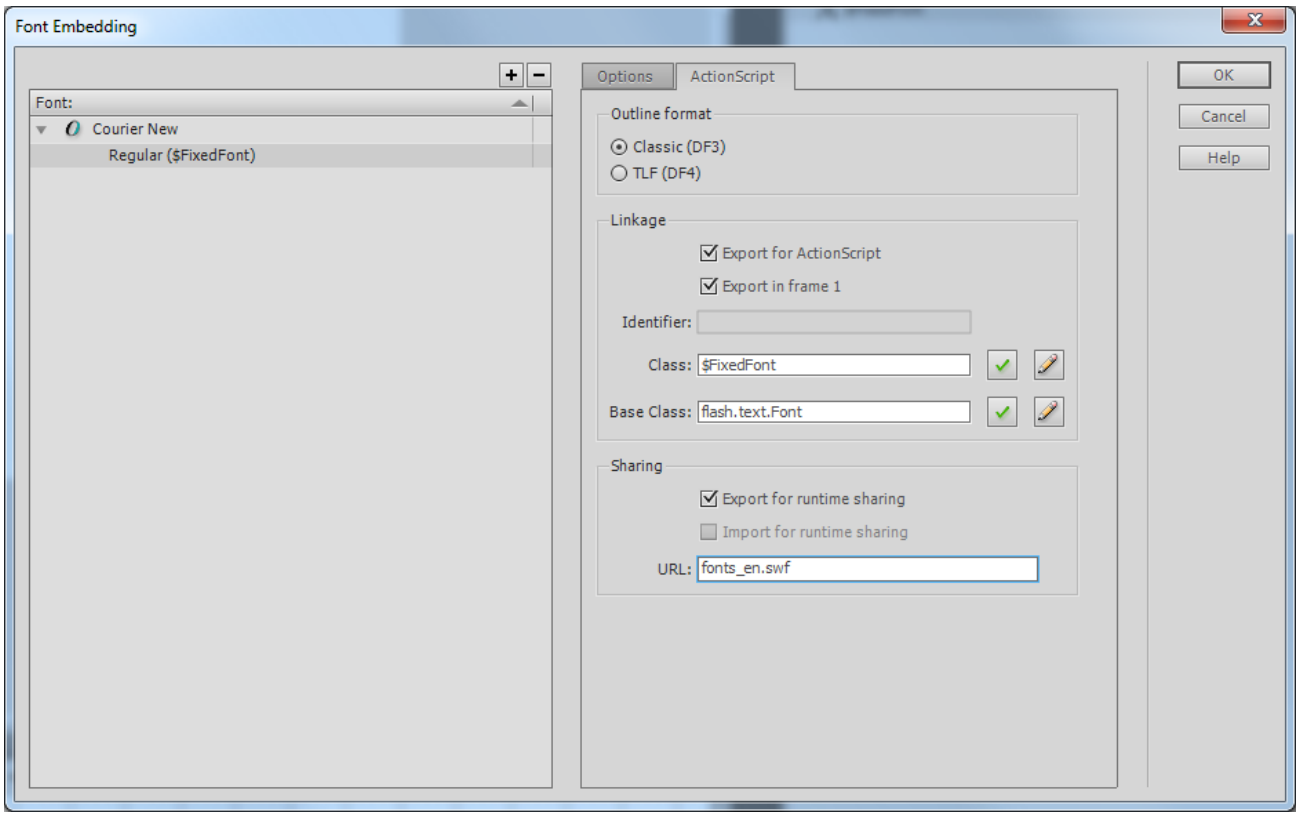
A number of websites recommend using the underscore '_' character as a prefix for font symbol names. Although underscore looks cleaner, using it will cause the text field "Font rendering method" selection box to break, since Flash Studio will treat any font name that starts with an underscore as a built-in font. To avoid this problem, we pick the dollar sign character in our examples instead.

2.1.1 Exporting and Importing Font Symbols

Much like other library entries, fonts' symbols located in an FLA file can be used in other files by either copying their content or relying on the import/export mechanism. To copy a library symbol, right-click on the source FLA library and select "Copy"; after that is done, switch to the target FLA files library, right click on it

and select “Paste”. A copy of a symbol will be created, with all of its data content duplicated in the second file.

In order to avoid data duplication, an export/import mechanism can be used in Flash. To export a library symbol, developers can right-click on it and select “Properties” and select “ActionScript” tab, which will display the following properties sheet:



By pressing “OK” button Flash will warn about absence of definition for the class, it is safe to ignore this warning message.

When exporting a symbol, you give it an export identifier and mark it with “Export for runtime sharing”, “Export for ActionScript” and “Export in first frame” flags. It is recommended that you set the import identifier to be the same as the symbol name. The URL should specify the path to this SWF file that will be used when this symbol is imported; if the font symbol is going to be substituted through a GFx::FontLib object, the URL field should be set to your default font library. We will use “fonts_en.swf” as a default name in this document; however, any other name can be used. **Please note that you must set the default font library in GFx to use font substitution.**

For example

```
Loader.SetDefaultFontLibName(“fonts_en.swf”);
```

You can set the default font library for the GFxPlayer you in the fonconfig.txt

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

The first fontlib appearance will be used as the default library

Once a symbol is exported, it will be marked with an "Export: $identifier" label in the library "AS Linkage" column. If the Copy / Paste or drag and drop operations are applied to an imported library symbol, they will no longer make a copy of it but will rather create an import link in the target file. This means that the target file will be smaller and when it is loaded into memory it will pull its imported data from the source SWF. In Scaleform, imported SWF data will be loaded only once even if it is imported from multiple files, potentially saving a significant amount of system memory.

Font symbol names will not be returned as a part of the TextField.htmlText string or as a font name in TextFormat; the original system font name, such as "Arial" will be returned instead. Similarly, symbol names cannot be assigned through ActionScript unless they match import names for an exported font (not recommended).

2.1.2 Exported Fonts and Character Embedding

Up to version CS4 Flash Studio did not allow to specify which characters to embed for exported fonts. Instead, the character set to be embedded was determined by the system locale setting. On Windows, this was controlled by the "Language for non-Unicode programs" setting in the Control Panel\Regional and Language Options\Advanced panel. If the language was set to "English", only 243 glyphs were exported for every font. If Korean language was selected, 11,920 characters were exported. Clearly this was not convenient for game development, and therefore the whole approach with 'gfxfontlib.swf' was created.

Fortunately, starting from Flash CS5, it is possible to specify which glyphs to be exported and therefore 'gfxfontlib.swf' is not necessary anymore. Instead, developer can create language specific font libraries (such as 'fonts_en.swf', 'fonts_jp.swf', etc) and directly specify which glyphs should be embedded in each of them.

However, if CS4 Flash Studio (or older) is still used you must use the 'gfxfontlib.swf' approach (it is still supported thought not recommended in the current version). Refer to older revision of this document for GFx 4.0 and below for further details about the approach.

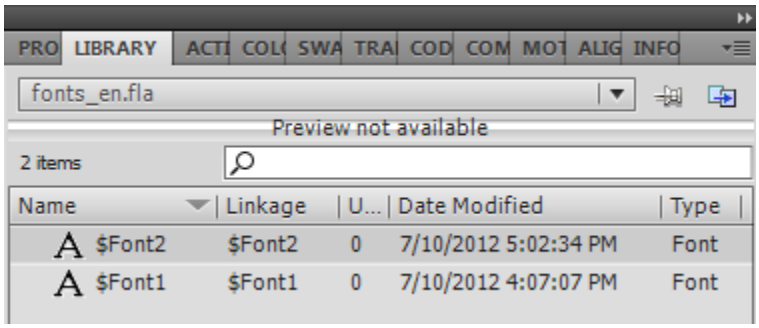
2.2 Steps for creating font library SWFs

With understanding of font symbols in place, we are now ready to create font library files. To reiterate, these files are created to provide a per-language source fonts. The details of how library fonts are mapped and substituted will be described in the next section.

To create a font library file for a specific language you create a new FLA in the Flash Studio and populate its library with font symbols. Specifically, you can go through the following steps.

1. Identify fonts used for all of the game screens and give them unique names based on their purpose. '$TitleFont' and '$NormalFont' can be good names to use for the game's title and normal size fonts, respectively.
2. Create a new FLA file for the library.
3. Create a new font symbol for every font identified in step 1.
4. If you plan to use imported font substitution, configure linkage properties for each font so that it is exported with the same identifier as the font symbol name.
5. Specify which glyphs should be embedded.
6. Save the resulting file as fonts_<lang>.swf, where 'lang' represents the language (for example, 'en' – English, 'ru' – Russian, 'jp' – Japanese, 'cn' – Chinese, 'kr' – Korean, etc). The name can be any, but you'll need to use this name later for FontMap configuration.

Unless you would like to have use of additional text fields for documentation purposes, there is no need to add any textfields into a stage of the font library. Aside from the fonts, no other symbol types should be added to the font library file. After the FLA is finished, the library should look similar to the following.



With the font library complete, its symbols are ready for use in the rest of the application's user interface screens. To make library fonts available in the importing files, you can either drop the font symbols into the target movie stage or use the Copy / Paste technique described earlier.

3 Part 2: Selecting the Internationalization Approach

The two primary entities that need to be substituted during UI internationalization are text field strings and fonts. Text field strings are substituted to achieve language translation, replacing development text with the corresponding phrases in the target language. Fonts are substituted to provide the correct character set for the target language, as the original development fonts may not contain all of the required characters.

In this document we describe three different approaches you can use for internationalizing art assets, which are:

1. *Imported font substitution*.
2. *Device font emulation*.
3. *Custom asset generation*.

Imported font substitution is the recommended approach for internationalizing fonts and relies on the GFx::FontLib and GFx::FontMap objects for substituting font symbols originally provided by the default 'fonts_en.swf' library.

Device font emulation is similar to imported font substitution, except it relies on font mapping to provide actual fonts instead of their imported symbols, which is a bit easier to setup but has several limitations.

Both imported font substitution and device font emulation rely on the user-created GFx::Translator object for translating text strings.

The custom asset generation is different from the above approaches in that it does not utilize font mapping or translator objects, relying instead on the Flash Studio features to generate custom UI asset files for each target language. This approach may be chosen on extremely memory limited platforms with a small number of fairly static UI assets.

3.1 Imported Font Substitution

Imported font substitution relies on the Flash font symbol import/export mechanism for binding text fields to substitutable fonts. To use this method developers first create a default font library file called, for example, "fonts_en.swf" as described in Part 1, and then use exported font symbols from that file in all of the game UI files. When the UI files are tested in the Adobe Flash player, they will be imported from "fonts_en.swf", allowing content to be rendered correctly in the development language. When running those assets in Scaleform, however, international configurations can be loaded instead, enabling translation and font substitution.

Note, that you should call *Loader::SetDefaultFontLibName (const char* filename)* C++ method and pass a filename (just filename, no path!) as a 'filename' parameter. For example, if the default development language is Korean you need to make the following call:

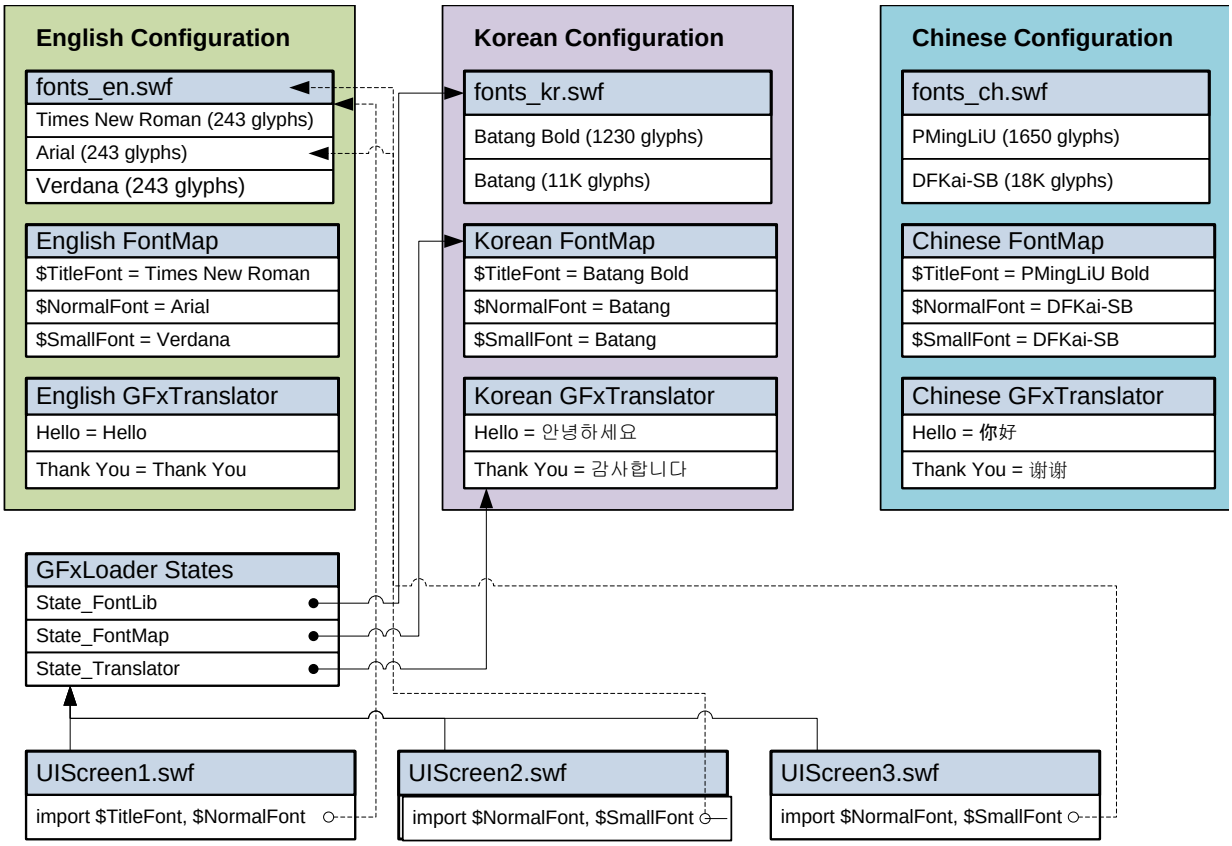
```
Loader.SetDefaultFontLibName("fonts_kr.swf");
```

This method should be called BEFORE calling CreateInstance for the main SWF file.

Fundamentally, international configuration for a game consists of the following data sets:

1. String translation table that maps each phrase in the game to its translated version.
2. A set of fonts that provide glyphs for characters used in the translation.
3. Font substitution table that maps font symbol names, such as "$NormalFont", to their fonts available in the configuration.

In Scaleform, these components are represented by the GFx::Translator, GFx::FontLib and GFx::FontMap state objects installable on GFx::Loader. When a movie file is loaded with a call to GFx::Loader::CreateMovie, it is bound to the font configuration states provided by the loader, relying on fonts in those states for rendering text. To create a different font binding, users need to set a new state combination on the loader and call CreateMovie again.



The figure above demonstrates a potential internationalized font setup. In this setup, three user interface files named "UIScreen1.swf" through "UIScreen3.swf" import a set of fonts from the "fonts_en.swf" library. The import file links are illustrated by the dashed arrow lines. Instead of taking the fonts from the import file, however, Scaleform can substitute fonts based on the FontLib and GFx::FontMap states set on GFx::Loader. This substitution will take place automatically if (1) the import file is matched with the one specified by Loader::SetDefaultFontLibName() method and (2) a non-null GFx::FontLib state is set on the loader before the UI screen is loaded.

To support international translation, the above figure uses three configurations: English, Korean and Chinese. Each configuration includes its own font file, such as "fonts_kr.swf" used for the Korean language, and a font map that maps imported font symbol names to the actual fonts embedded into the font file. In our example, the loader states are set to the Korean configuration, causing Korean fonts and translation tables to be used. Loader states, however, could have just as easily been set to the English or Chinese configuration, in which case the user interface would have been translated to those languages instead. The UIScreen files do not need to be modified for this translation to take place.

3.1.1 Configuring International Text

When internationalizing a game with imported font substitution, developers will first need to create the international art assets and the required language configurations. The art and font assets will normally be stored in the SWF format (convertible to GFX before shipment), while the font map and translation table data can be stored in game specific data formats that support creation of GFx::FontMap and GFx::Translator objects. In Scaleform samples, we rely on the "fontconfig.txt" configuration file to provide this information; however, game developers should create a more advanced scheme for production.

Formally, the process of creating internationalized configurations can be broken down into the following steps:

1. Create a default library file (e.g. "fonts_en.swf") as described in Part 1 and use it to create your game assets.
2. Create a translation string table for each language that can be used to create a corresponding GFx::Translator objects.
3. Decide on what font mappings will be used for each language; store this map in a format that can be used to create GFx::FontMap.
4. Create language-specific font files such as "fonts_kr.swf" for each target language.

Once the international configurations are created, developers can load internationalized UI screens into their game. With the target language selected, they need to take the following steps:

1. Create a translator object for the language and set it on GFx::Loader. Translation is implemented by deriving your own class from GFx::Translator and overriding the Translate virtual function. By overriding this class, developers can represent translation data in any format they choose.
2. Create a GFx::FontMap object and set it on GFx::Loader. Add the necessary font mappings to it by calling the MapFont function. An example of using GFx::FontMap is provided in Part 3.
3. Set any other font related states necessary for the game, such as GFx::FontProvider and/or GFx::FontPackParams, on the loader. Configure dynamic cache through GlyphCacheConfig on the render thread, if necessary.
4. Create a GFx::FontLib object and set it on GFx::Loader.
5. Call the SetDefaultFontLibName(filename) method for the default font library (the one that was used in the swf content):

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

   After GFx::FontLib is created, load the font source SWF/GFX files used in the target language and add them to the library by calling GFx::FontLib::AddFromFile. This call will make the fonts embedded in the argument files available for use as font imports or device fonts.

6. Load the user interface files by calling GFx::Loader::CreateMovie. The font map, fontlib, and translator states will automatically apply to the user interface.

3.1.2 Internationalization in Scaleform Player

To demonstrate internationalization, Scaleform Player supports the use of the international profile file, normally called "fontconfig.txt". When you drag and drop an SWF/GFX file into the player, the 'fontconfig.txt' configuration will automatically load from the files local directory, if available; it can also be loaded with the /fc option at command line. When the international profile is loaded, user can switch between its language configurations by pressing the Ctrl+N key. The current configuration is displayed in the bottom of the player HUD when the F2 key is pressed.

The international profile file can be saved in either 8-byte ASCII or UTF-16 format and has a linear structure created by listing font configurations followed by their attributes. The following international profile can be used to describe the game setup presented in the previous section.

```
[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"
```

As can be seen from the sample, there are three separate configuration sections each beginning with a `[FontConfig "name"]` header. The name of the configuration is displayed in the Scaleform Player HUD when that configuration is selected. Within each configuration, a list of statements is used that is applied to that configuration. The possible statements are outlined in the following table.

Configuration Statement	Meaning

`fontlib "fontfile"`	Loads the specified SWF/GFX font file into GFx::FontLib. The first fontlib appearance will be used as a default font library.
`map "$UIFont" = "PMingLiU"`	Adds an entry to GFx::FontMap that maps a font used in game UI screens to a target font provided by the font library. With imported font substitution, the `$UIFont` should be a name of the font symbol originally imported into UI files from the default "fonts_en.swf".
`tr  "Hello" = "你好"`	Adds a translation from the source string to its equivalent in the target language. Developers should use a more advanced solution for storing translation tables.

Users should be aware that configuration file functionality is provided primarily as a sample of how internationalization can be achieved with Scaleform. The structure of configuration file can change in the future without notice; furthermore, we plan to transition it to XML format as Scaleform internationalization functionality evolves. Developers can refer to source code in the FontConfigParser.h/.cpp file for an example of how font states can be configured.

A more complete imported font substitution sample is shipped with Scaleform in the Bin\Data\AS2\Samples\FontConfig and Bin\Data\AS3\Samples\FontConfig directories. This directories includes the "sample.swf" file that can be dropped into the Scaleform Player and a "fontconfig.txt" file, which translates it into a variety of languages. Developers are encouraged to examine the files in that directory to better understand the internationalization process.

3.1.3 Device Font Emulation

Device font emulation is different from imported font substitution in that it does not make use of imported font symbols and relies on GFx::FontMap to substitute system font names instead. During UI creation, artists choose a set of development system fonts, such as "Arial" and "Verdana", and use them directly as device fonts for all of the UI assets. When running in Scaleform Player, a font configuration file can be used to map system font names to their alternatives in the target language. As an example of such mapping, "Arial" can be mapped to "Batang" for Korean user interfaces. The "Batang" font, in turn, can be loaded from a "font_kr.swf" file through the use of GFx::FontLib.

Device font emulation can be useful if developers plan to use system fonts directly through GFx::FontProviderWin32 or font files through GFx::FontProviderFT2, without relying on GFx::FontLib. Although this approach may seem simpler to setup, device fonts have a number of limitations that make using them less flexible than imported font substitution:

- Device fonts cannot be transformed in Flash. Adobe Flash player will fail to display the device font text field if any rotational transformation is applied to it. Furthermore, the player will not scale device font text correctly and will ignore masks applied to it. Although all of these features will work fine in Scaleform, limited support in Flash makes UI art asset testing more difficult.

- Device fonts cannot be visually configured to have the "Anti-alias for animation" setting. Scaleform Player will always anti-alias device font text for readability, unless specified otherwise through the TextField.antiAliasType property in ActionScript.
- When device fonts are used it becomes impossible for artists to modify development fonts without re-editing all of the UI assets. The use of the font map is possible on the Scaleform side, of course, but that will not apply when the files are tested in the Adobe Flash player. When the imported font substitution is used instead, different development fonts can be chosen by simply editing and re-generating the default font lib file ("fonts_en.swf").
- When working with TextFormat in ActionScript, Flash will not recognize font symbol names unless they have been imported. This means that developers will need to use direct font names, such as "Arial", instead of symbolic names, such as "$TitleFont".

With device font emulation the "fonts_en.swf" file is not strictly necessary, although it is still useful as a font symbol repository during development. If artists choose to use "fonts_en.swf" file, they should NOT export its font symbols; instead those symbols can be copied into the target UI files where they can act as aliases to the mapped device fonts they represent.

3.1.4 Step-by-step guide to create a font library

Here we will show steps how to create a simple SWF that uses font library with two languages and with using fontconfig.txt.

1. Create a new FLA, choose ActionScript 2.0 or 3.0. Call it 'main-app.fla'. This will a main application SWF that uses font library.
2. Create another FLA, choose the same ActionScript you chose on previous step. Call it 'fonts_en.swf'. This will be our default font library.
3. Go to 'Library' window, click right mouse button and choose 'New Font…' Specify name as $Font1, choose "Family" as "Arial", "Style" as "Regular".
4. In "Character ranges" choose characters you want to embed. For English it is enough to choose 'Basic Latin'.
5. Switch to 'ActionScript' tab. Check 'Export for ActionScript', 'Export for frame 1', 'Export for run-time sharing'. In 'URL' input field type 'fonts_en.swf'. Click 'OK' button. For ActionScript 3.0 Flash Studio will show you a warning 'A definition for this class could not be found…' – ignore it and click 'OK' again.
6. Repeat steps 3 – 5 for another font, name it '$Font2' and choose DIFFERENT 'Family' and/or 'Style'. If both 'Family' and 'Style' are the same then Flash Studio may discard the second font as duplication of the first one. Let say we choose 'Times New Roman' this time.
7. Create one more FLA, same ActionScript setting. Call it 'fonts_kr.fla' or 'fonts_ru.fla' or any other name depending on language you want to use. Let say it is Korean, so 'fonts_kr.fla'.

8. Go to 'Library' window, click right mouse button and choose 'New Font…' Specify name as '$Font1', but choose "Family" as some Korean font, for example as "바탕", "Style" as "Regular".
9. In "Character ranges" choose characters you want to embed. For Korean it might be 'Korean Hangul (All)'.
10. Switch to 'ActionScript' tab. Check 'Export for ActionScript', 'Export for frame 1', 'Export for run-time sharing'. In 'URL' input field type 'fonts_kr.swf'. Click 'OK' button. For ActionScript 3.0 Flash Studio will show you a warning 'A definition for this class could not be found…' – ignore it and click 'OK' again.
11. Repeat steps 8 – 10 for another font, name it '$Font2' and choose 'Family' as, let say, '바탕체'
12. Publish both 'fonts_en.swf' and 'fonts_kr.swf'
13. Now it is time to come back to our main-app.fla. But first go to our default font library file (which is 'fonts_en.swf'), go to Library, select both '$Font1' and '$Font2' there and copy them into a clipboard (Ctrl-C or right mouse click -> 'Copy').
14. Switch to 'main-app.fla', go to Library and paste font symbols (Ctrl-V or right mouse click -> 'Paste'). You will see your fonts with 'Import:' prefix.
15. Create a text field on the stage of 'main-app.fla'. Make it 'Classic Text', 'Dynamic Text'. Choose '$Font1*' from the dropdown 'Family' list. Type '$TEXT1' as a content of the text field (this will be used as ID for translation).
16. Create a second text field, choose '$Font2*', type '$TEXT2' in it.
17. Publish the 'main-app.swf', we are done here.
18. Now, need to create a fontconfig.txt in the same directory where we store all our swfs. Open 'Notepad' or any other text editor that can work with Unicode or UTF-8. Type the following:

```
[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!"
```

IMPORTANT: Even though we used Korean names for fonts ("바탕" and "바탕체"), Flash may use English names in SWFs ("Batang" and "BatangChe"). This is why we use English font names in fontconfig.txt. To make sure which font names were generated in the SWF you may use *gfxexport –fntlst <swfname>* command and analyze the output *.lst file.

19. Save the file (don't forget to save it as Unicode or UTF-8!). You are done! Open the 'main-app.swf' in GFxPlayer and you'll see the default text as 'This is' and 'English!'. Use Ctrl-N to switch to Korean and you'll see how English text will be replaced by Korean with using fonts from fonts_kr.swf (since we embedded Korean glyphs only in fonts_kr.swf).

3.2 Custom Asset Generation

As its name implies, custom asset generation relies on creation of custom SWF/GFX files for each target language distribution. If a game uses the "UIScreen.fla" file, for example, a different version of a SWF can be generated manually for each game distribution, with Korean "UIScreen.swf" being different from English "UIScreen.swf". The different versions of the file can be placed into different file system directories or not distributed at all in the target market.

The primary benefit of custom assets use is that they allow the absolute minimum number of glyphs to be embedded in each file. Although we do not generally recommend this approach, custom asset generation can be considered when memory is extremely limited (as in less than 600K available for the UI) and art assets meet the following criteria:

- The number of UI asset files is relatively small and their text content is simple.
- Multiple UI files are rarely loaded together (if they are, they would benefit from the font sharing feature provided by other approaches).
- Dynamic text field updates are limited and use a small number of embedded characters.
- UI does not require Asian language IME support.

If developers choose to adopt custom asset generation for a game title, we recommend that they read the "Authoring multilanguage text with the Strings panel" topic in Flash Studio documentation and consider using the Strings panel (Ctrl + F11) for their international text. Artists will need to set the "Replace strings" setting to "manually using stage language" in order for Flash text substitution to work correctly with current version of Scaleform.

4 Part 3: Configuring Font Sources

Each text field in Flash has a font name associated with it that is either stored directly or encoded in its HTML tags. When text field is displayed, the font used for rendering is obtained up by searching the following font sources:

1. Locally embedded fonts.
2. Font symbols imported from separate SWF/GFX files.
3. SWF/GFX files installed through GFx::FontLib, searched either by font name or through imported font substitution.
4. System font provider, such as GFx::FontProviderWin32, if one is installed by the user.

The use of embedded and imported fonts has been described in the beginning of this document. For the most part, embedded fonts work identically to Flash and thus require no custom configuration. For the purposes of font lookup, imported font symbols act similar to the embedded fonts.

The three installable states that configure non-embedded font lookup are GFx::FontLib, GFx::FontMap and GFx::FontProvider. As described earlier in the document, GFx::FontLib and GFx::FontMap are used to look up SWF/GFX-loaded fonts for imported font substitution or device font emulation. The use of the system font provider will be covered in detail below.

The system font provider is installed with the GFx::Loader::SetFontProvider call; it allows font data to come from an alternative non-SWF file source. Font provider can only be used with a dynamic cache and is searched only if the font data is not embedded in the SWF or the font library. Currently, two font providers are included with Scaleform:

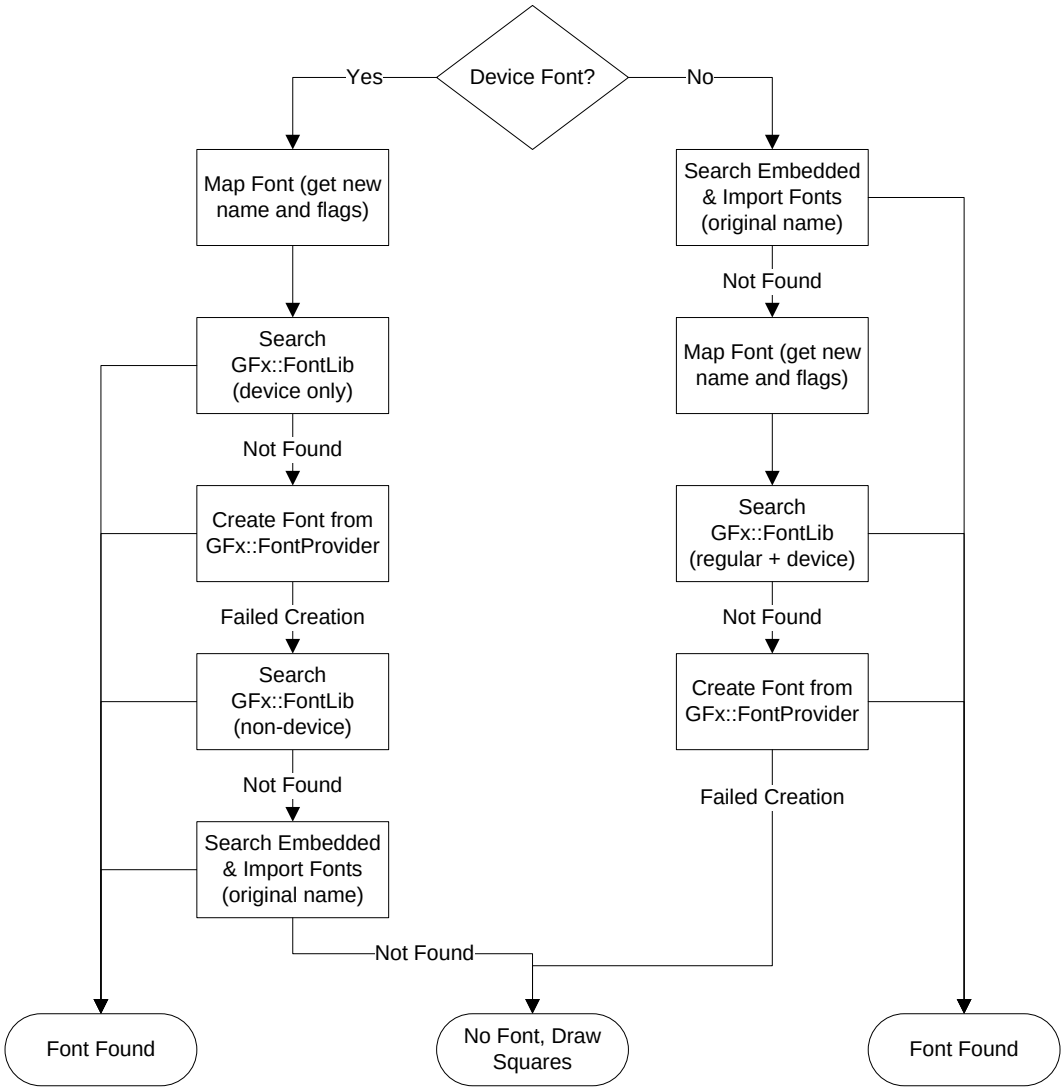
- GFx::FontProviderWin32 – Relies on Win32 APIs to obtain font glyph data.
- GFx::FontProviderFT2 – Uses the FreeType-2 library by David Turner to read and interpret stand-alone font files.

In this part we describe the font lookup order in detail and provide code examples of how different font sources can be configured.

4.1 Font Lookup Order

The flowchart on the following page illustrates the order of font lookup in Scaleform . As can be seen from the chart, the lookup behavior is modified by the Device Font flag, set if the “Use device fonts” rendering method is selected in text field properties.

In Adobe Flash, setting the device font flag causes the corresponding font to be obtained from the system in preference of identically named embedded fonts (in this case, the later are used as a fall-back). Scaleform replicates this behavior, although its installed font library is searched before the system font providers. Such setup does not cause conflicts since most console-portable games will rely on shared font libraries, while games which choose to use system providers can leave the font library in the un-initialized (null) state.



As can be seen from the diagram, embedded fonts are searched first if the Device Fonts are *not* used for a text field and last otherwise. Furthermore, embedded fonts are always looked up based on the original font name used in a text field, while a font map can be applied to substitute font names looked up from GFx::FontLib and GFx::FontProvider.

4.2 GFx::FontMap

GFx::FontMap is a state used to substitute font names, allowing alternative fonts to be used during internationalization if the required characters are not available in the development font. The font map is used with both imported font substitution and device font emulation. In the first case, it converts font symbol identifiers into font names. In the second case, it maps original font names into the translated alternative.

In Part 3, font maps were created with the following lines in the font configuration file:

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

In the example above, the mapping statements are used to map font import identifiers, such as "$TitleFont" to the actual fonts font names, which in this case are embedded in the font library file.

Equivalent font map setup is achieved with the following C++ statements.

```
#include "GFx/GFx_FontLib.h"
. . .
Ptr<GFx::FontMap> pfontMap = *new GFx::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);
```

In this case, all three fonts are mapped to the same font name; in particular "$NormalFont" and "$SmallFont" also share the same font style, saving memory used by the font library file. Different styles can be specified with a third argument to MapFont, forcing the mapping to use a particular embedding. If the argument is not specified, MFF_Original value is used, indicating that font lookup should retain the original style specified for the text field. The size of the font can be changed by a factor set by the parameter scaleFactor. By default, the scaleFactor is set to 1.0f. This parameter is useful if the font is not visible well enough and needs to be slightly increased.

Developers should be aware that GFx::FontMap is a binding state, meaning that movie instances created with it will use the state even if it is later changed in the loader. If a different font map is set, a different GFx::MovieDef will be returned for it by GFx::Loader::CreateMovie for the same file name. This is also true for all other font configuration states.

4.3 GFx::FontLib

As discussed in Part 3, GFx::FontLib state represents an installable font library that (1) provides alternatives for fonts imported from the default "fonts_en.swf" and (2) provides fonts for device font emulation. The font library is searched before system font providers. Its use should be clear from the following example.

```
#include "GFx/GFx_FontLib.h"
. . .
Ptr<GFx::FontLib> fontLib = *new GFx::FontLib;
loader.SetFontLib(fontLib);
fontLib->SetSubstitute("<default_font_lib_swf_or_gfx_file>");

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

The idea is to create and load the movies as usual, but instead of playing them, the movies are used as font storages. It is possible to load as many movies as needed. If the same font is defined in different movies, only the first one will be used.

The first parameter to AddFontsFrom is a movie definition that will serve as a source for fonts. The second parameter to AddFontsFrom is a pin flag, which should be set if the loader should AddRef to the movie in memory. This flag is only necessary when users do not keep smart pointers to the loaded movies (m1 and m2 in the example). If a pin flag is false, font binding data such as exported or packed textures may be released early and have to be reloaded/regenerated when the font is used.

Similar to the font map, GFx::FontLib is a binding state, which is referenced by the created movies until they die.

4.4 GFx::FontProviderWin32

The GFx::FontProviderWin32 font provider is available with Win32 API, where it can rely on the GetGlyphOutline() function to retrieve vector data. It can be used as outlined below:

```
#include "GFx/GFx_FontProviderWin32.h"
. . .
Ptr<GFx::FontProviderWin32> fontProvider = *new GFx::FontProviderWin32(::GetDC(0));
loader.SetFontProvider(fontProvider);
```

The constructor of GFx::FontProviderWin32 takes the handler of Windows Display Context as an argument. In most cases the use of the screen DC is appropriate (::GetDC(0)).

The fonts will be created as necessary, when requested.

4.4.1 Using Natively Hinted Text

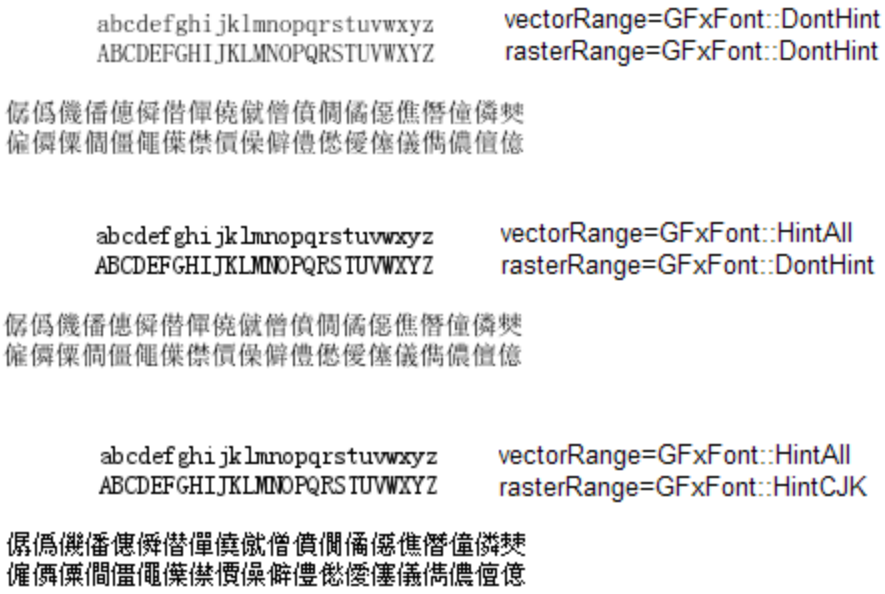
In general, font hinting is a very non-trivial problem. Scaleform provides an auto-hinting mechanism, but it does not work well enough for Chinese, Japanese, and Korean (CJK) characters. Besides, most of well-designed CJK fonts contain raster images for glyphs of certain sizes, because at small sizes proper hinting of CJK becomes extremely complex. Duplicating vector glyphs with raster images is a good and practical solution. System font providers based on font APIs (GFx::FontProviderWin32 and GFx::FontProviderFT2) can produce glyphs in both, vector and raster representations. The font providers have an interface to control the native hinting. The interfaces are slightly different, but have the same principle. There are 4 parameters:

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

Parameters vectorRange and rasterRange control the character range in which the hinting is used. The values are:

`Font::DontHint` – do not use native hinting,
`Font::HintCJK` – use native hinting for Chinese, Japanese, and Korean characters,
`Font::HintAll` – use native hinting for all characters.

The rasterRange has a priority over the vectorRange. Parameters maxVectorHintedSize and maxRasterHintedSize define the maximal font size in pixels at which the native hinting is used.
The picture below contains an example of the font SimSun, with different options.



As you can see, native vector hinting does not help much for this font, while using the raster images change the text dramatically. However, for certain fonts, such as "Arial Unicode MS" vector hints make sense.

Note that unlike Adobe Flash, Scaleform supports arbitrary affine transformations for device fonts. Rotation and skewing certainly blur the glyphs, but it still remains quite appropriate for animated text.

By default, both, Win32 and FreeType providers use the following values:

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

Particular interfaces for configuring the hinting are described below.

4.4.2 Configuring Native Hinting

By default GFx::FontProviderWin32 uses native raster hinting for CJK characters and does use vector native hinting (). It is possible to change this behavior by calling:

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

or

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

Note that if function SetHintingAllFonts() is called after SetHinting() it will not change the hinting behavior for this particular font. The exact functions prototypes are:

```
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);
```

Parameter name can use UTF-8 encoding.

4.5 GFx::FontProviderFT2

This font provider uses the FreeType-2 library by David Turner. Its use is similar to GFx::FontProviderWin32, except for the necessity of mapping the font names and attributes to actual font files.

First, we recommend developers read the FreeType manual to properly configure and build the library. It is the developers' responsibility to decide to use static or dynamic linking, as well as proper runtime configuration settings such as font drivers, memory allocators, external file streams, and so on.

For Windows MSVC compilers with static linking use the following libraries:
freetype<ver>.lib – for Release Multi-threaded DLL code generation,
freetype<ver>_D.lib – for Debug Multi-threaded DLL,
freetype<ver>MT.lib – for Release Multi-threaded (static CRT),
freetype<ver>MT_D.lib – for Debug Multi-threaded (static CRT).

Where "<ver>" is the version of FreeType, for example, for version 2.1.9 it's 219.

Also, make sure the directories freetype2/include and freetype2/objs are available in the paths for additional includes and libraries respectively.

In order to use FT2 font provider in addition to steps mentioned above, the following files need to be added to the build:
Src\Render\FontProvider\Render_FontProviderFT2.cpp,
Src\Render\FontProvider\Render_FT2Helper.cpp ,
Src\GFx\GFx_FontProviderFT2.cpp

Creating of the GFx::FontProviderFT2 object looks as follows:

```
#include "GFx/GFx_FontProviderFT2.h"
. . .
Ptr<GFx::FontProviderFT2> fontProvider = *new GFx::FontProviderFT2;
<Map Font to Files or Memory>
loader.SetFontProvider(fontProvider);
```

The constructor has one argument:

```
FontProviderWin32(FT_Library lib=0);
```

It is the FreeType library handle. If zero (default value), the provider will initialize FreeType internally. The ability to specify an existing initialized external handler is provided in case the application already uses FreeType and wants to share the handle between Scaleform and other systems. Note that when using the external handler it's the developer's responsibility to properly release the library (call FT_Done_FreeType). Also, the application must guarantee that the life time of the handle is ***longer*** than the life time of GFx::Loader. Using the external handler is also supported in cases where it is desirable to reconfigure the run-time call-backs of FreeType (memory allocators and such). Note, that this configuration can be done with the internal initialization too. The `fontProvider->GetFT_Library()` function returns the used `FT_Library` handler and it is guaranteed that no calls to FreeType will be invoked by the provider before actual use of the fonts. As a result, there is a chance to reconfigure the FreeType at run-time after creating GFx::FontProviderFT2.

4.5.1 Mapping the FreeType Fonts to Files

Unlike Win32 API, FreeType does not provide mapping typeface names (and attributes) to actual font files. Thus, this mapping must be provided externally. GFx::FontProviderFT2 has a simple mechanism of mapping the fonts to files and memory.

```
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);
```

Argument “fontName” specifies font typeface name, for example, “Times New Roman”. Argument “fileName” specifies the path to the font file, for example, “C:\\WINDOWS\\Fonts\\times.ttf”. The “fontFlags” can have values “Font::FF_Bold”, “Font::FF_Italic”, or “Font::FF_BoldItalic”. Value “Font::FF_BoldItalic” actually equals to “Font::FF_Bold | Font::FF_Italic”. The “faceIndex” is what is passed to FT_New_Face. In most cases this parameter is 0, but not always, depending on the type and the content of the font file. Note that this function does not actually open the font files; it only forms the mapping table. The font files will be open when they are actually requested. Unlike the Win32 font provider, the FreeType uses just function’s arguments to configure the native hinting.

4.5.2 Mapping Fonts to Memory

FreeType allows fonts to be mapped to memory. It may make sense, for example, if a single font file contains many typefaces, or if the font files are already loaded by the application.

```
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);
```

A simple (and somewhat dirty) example of mapping fonts to memory is below.

```
    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);
    }
```

Memory is not released in this example (and, yes, eventually it will result in a memory leak). It’s because the mapping table keeps only a naked constant pointer to the font data. It is the developer’s responsibility to properly release it. The application must guarantee that the life time of this block of memory is ***longer*** than the life time of GFx::Loader. It is designed to provide as much freedom as possible in the font mapping mechanism. For example, the application may already use FreeType with pre-loaded memory fonts whose allocation and destruction is handled externally from Scaleform.

An example of using the FreeType font mapping in Windows may look as follows.

```
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);
```

Note that it is only an example; in a real application it is a bad idea to specify the absolute hard-coded file paths. Normally, it should come from a configuration file, or by automatically scanning the font typefaces. Specifying the explicit typeface names may seem like overkill, because the fonts typically contain these names, but it avoids file parsing which can be an expensive operation. The function MapFontToFile() only stores this information and does not open the file unless it's requested. As a result, a large number of mapped fonts can be specified, while only a few of them are actually used. In this case no extra file operations will be performed.

5 Part 4: Configuring Font Rendering

Scaleform can render text characters in two ways. First, font glyphs can be rasterized into a texture and later drawn using batches of textured triangles (two per character). Alternatively, font glyphs can be tessellated into triangle meshes and rendered as vector shapes.

For most text in an application, textured triangles should always be used for performance reasons. The glyph textures are generated through either dynamic or static caching. Dynamic glyph caching is more flexible and produces higher quality images, while static caching has the advantage of being computable off-line.

Depending on the target platform and title needs, developers can choose from the following font rendering options:

1. Use the dynamic cache, allowing for both fast animation and high-quality output. The dynamic cache will use a fixed amount of texture memory and will reduce load time as all glyphs do not need to be rasterized and/or loaded.
2. Use the static cache, loading pre-generated mip-mapped textures from disk. Developers can use the GFxExport tool to generate packed glyph textures ahead of time, stripping out the font vector data so that it does not need to be loaded.
3. Use the static cache, auto-generating textures at load time with GFx::FontPackParams. This is similar to the previous method, except textures do not need to be loaded from disk.
4. Use vector shapes to render text glyphs directly.

Using dynamic cache is recommended on higher end systems such as PC, Xbox 360, PS3 and in many cases Nintendo Wii. The dynamic cache will ensure that load times are minimized and the fonts are rendered with the highest quality possible for the given resolution. The use of dynamic cache is also required if developers plan to utilize system or external font support through either GFx::FontProviderWin32 or GFx::FontProviderFT2.

Using the static cache with textures exported by the GFxExport tool is a good option for CPU and memory constrained platforms such as PSP and PS2, where runtime rasterization of glyphs may be too expensive. Static cache can also be used if developers need to avoid dynamic texture updates in their renderer implementation. However, as we optimize vector data memory use, dynamic cache may become a good option even on the lower performing systems. We recommend that developers experiment with their game data to determine the best solution.

Although Scaleform can be configured to do so, vector shapes are rarely used stand-alone for text rendering. Instead, glyph tessellation is generally applied for large glyphs only, combined with texture based methods

for efficient rendering of small text. Developers can refer to the Controlling Vectorization section for more details on how to control or disable this option.

5.1 Configuring Glyph Cache

Font rendering in Scaleform is configured though either Render::GlyphCacheConfig or GFx::FontPackParams state objects. By default, glyph cache is automatically created by the Renderer2D object on the render thread and initialized for dynamic glyph caching. The glyph packer is only used for static texture initialization; pack parameters are null by default. With this setup, all of the created movies will automatically use the dynamic font cache, unless they come from GFX files pre-processed to have static textures.

Scaleform 4.0 Upgrade Note

Dynamic glyph cache configuration was changed significantly between Scaleform 3.x and 4.0 versions. While GFx 3.3 relied on the GFxFontCacheManager object maintained by the GFxLoader and configured on the main thread, GFx 4.0 replaced it with the GlyphCacheConfig interface maintained by the Renderer2D and configured on the render thread. See section 5.2 for more details on configuring the cache in Scaleform 4.0 and higher.

The glyph cache system is always used when raster glyphs are rendered from a texture. Internally, the cache manager is responsible for maintaining the text batch vertex arrays, which are created when both static and dynamic texture caching is used. When dynamic caching is enabled, the cache manager is responsible for allocating cache textures, updating them with rasterized characters and keeping textures in sync with the batch vertex data. When only static caching is used, cache manager still needs to maintain text vertex arrays, but it does not need to allocate or update dynamic textures.

The static cache is represented by pre-rasterized bitmap textures with tightly packed glyphs. Static texture rasterization and packing can either be done at load time based on GFx::FontPackParams or off-line with the 'gfxexport' tool. In both cases, static textures are associated with fonts. Depending on how it was loaded, an embedded font will or will not have a set of static textures containing its font glyphs. If the font does have static textures, they will always be used for rendering that font's glyphs; otherwise, the dynamic cache will be used if enabled.

In addition to depending on the type of font textures in use, the rendering method also depends on the glyphs destination pixel size. More formally, the following logic is used:

```
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;
}
```

As outlined above, vector rendering is used if texture caches are not available or if the glyphs are too big to be rendered from a texture. The cache approach is chosen based on availability of packed glyph textures in the font. The details of how both of the approaches can be setup are provided in the following sections.

5.2 Using the Dynamic Font Cache

The static cache is efficient when the font and character set is restricted, for example, Basic Latin, Greek, Cyrillic, etc. However, for most Asian languages the static cache may consume far too much system and video memory and result in slow loading. In addition, it may happen when using too many different typefaces; in other words, when the total number of embedded glyphs is big (say, 10000 or more). In these cases it makes sense to use the dynamic cache mechanism. Besides, the dynamic cache provides more capabilities, such as optimization for readability that drastically improves the quality. The dynamic cache is enabled and allocates its buffers by default; to disable this you can call:

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

In the above call `renderer` is a Render::Renderer2D object, is maintained on the render thread. This call will set the number of dynamic textures used by glyph cache to zero, effectively disabling it. To avoid default temporary buffer allocation, this call should be done before render HAL is initialized.

The dynamic cache rasterizes glyphs and updates the respective textures on demand when drawing text. It uses a simple LRU (Least Recently Used) cache scheme, but with a smart adaptive glyph packing into textures performed “on the fly”.

You can configure the texture parameters in the following way:

```
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);
```

The above values are used by default.

`TextureWidth, TextureHeight` - the size of the caching textures. Both values are rounded to the nearest greater power of two.

`MaxNumTextures` - the maximal number of textures used for caching.

`MaxSlotHeight` - the maximal height of the glyphs. The actual pixel glyph height cannot exceed this value. Bigger glyphs are rendered as vector shapes.

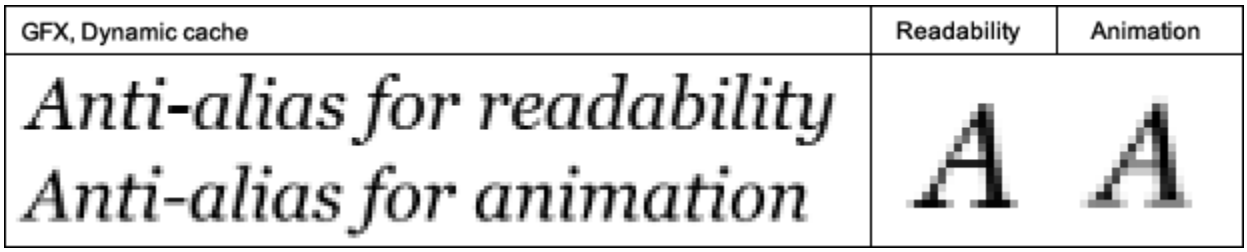
`SlotPadding` - the margin value used to prevent clipping and overlapping of the glyphs. Value 2 is good enough in most practical cases.

`TexUpdWidth, TexUpdWidth –` the size of the image used to update the texture. Typically 256x512 (128K of system memory) is good enough in all practical cases. It's possible to reduce it to 256x256 or even 128x128, but in this case the texture update operation will happen more frequently.

Dynamic glyph cache tries to pack the glyphs into the textures as tight as possible. This occurs dynamically, preserving the "Least Recently Used" cache strategy. To understand how the font glyph caching works, consider a very simple memory allocator which is allowed to allocate only 4 KB blocks of memory within a given space of 1MB. The allocator can allocate or de-allocate only these 4 K blocks. Obviously, the allocator is capable to have at most 256 simultaneous allocations. But in most cases, the requested sizes are much less than 4 K. They could be 16 bytes, 100 bytes, 256 bytes, and so on, but no more than 4K. In this situation you try to invent some mechanism to handle smaller memory blocks within those 4K blocks and obtain a bigger capacity of allocations within the given space of 1 MB. But the guaranteed minimum still remains the same – those very 256 simultaneous allocations. A similar mechanism works in the dynamic glyph cache, but in 2-dimensional texture space. In other words, glyph cache guarantees the capacity to store squares of MaxSlotHeight+2*SlotPadding. But it also tries to pack the glyphs much tighter, especially the small ones. In average cases this results in increasing the dynamic font cache capacity 2-5 times (for very small glyphs, increasing the capacity may be in factor of 10s). Typically, 60-80% of the texture space is used for the payload, not depending on the glyph sizes.

The total cache capacity (i.e., the maximum number of different glyphs simultaneously stored in cache) depends on the average glyph size. For the typical game UI we can roughly estimate that the above parameters allow for caching from 500 to 2000 different glyphs. The maximal number of cached glyphs must be enough to handle the visible part of any single text field. In this case, if the number of ***different*** glyphs in the ***visible part*** of a ***single*** text field exceeds the cache capacity, the remaining glyphs are drawn as vector shapes.

As was mentioned before, the dynamic cache allows for extra capabilities and provides support for "Anti-alias for readability" and "Anti-alias for animation" options. These options are the properties of the text field that the Flash designer can select in the text dialog panel. Text optimized for readability looks more sharp and readable. Although it also can be animated, this animation is more expensive because it results in more frequent texture update operations. Besides, the glyphs are automatically snapped to the pixel grid to reduce blurriness with the pixel auto-fitting operation, which means the text lines are also snapped to pixels. Visually it has a jitter effect when animating, especially changing scale. The figure below demonstrates the difference between these options.



When "Anti-alias for readability" is used, the glyph rasterizer performs an automatic fitting procedure (also known as auto-hinting). Scaleform does not use any kind of glyph hints from Flash files or any other font sources. All it requires to render text is glyph outlines. It provides identical visual appearance of text independently of the font source. The Scaleform auto-hinter requires certain information from the font, namely the height of Latin letters with flat top. It is necessary to correctly snap glyphs with overshoots (such as O, G, C, Q, o, g, e, and so on). Typically, fonts do not provide such information, so that, it has to be deduced somehow. For that purpose Scaleform uses Latin letters with the flat top. They are:

- any of H, E, F, T, U, V, W, X, Z for capital letters,
- any of z, x, v, w, y for lower case letters.

In order to use the auto fitter, the font must contain at least one of the above for capital letters and at least one for lower case. If the font does not contain them, Scaleform will produce a log warning:

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

If the warning appears, the Flash designer should just embed these letters. Typically it's enough to add just "Zz" in the "Embedding Characters" dialog form, or embed "Basic Latin".

5.3 Using font compactor – gfxexport

The purpose of the command line gfxexport tool is to pre-process SWF files, generating GFX files which are distributed and loaded into the game. During preprocessing, the tool can strip images out of the SWF file, extracting them into external files. External files can be stored in a number of useful formats, such as DDS and TGA. When –fc option is selected gfxeport also will compress font vecrtor data. This is lossy compression so you may need to experiment with parameters to find best compromise between memory consumption and font quality.

Command Line Option	**Behavior**
-fc	Enable font compactor.
-fcl <size>	Set nominal glyph size. Small nominal size will result in smaller data size but less accurate glyph. The default value is 256. In most cases nominal size 256 can save about 25% of memory (compared to the nominal size of 1024 that is typically used in Flash) without visible quality degradation. However for really big glyphs you may want to increase nominal size.
-fcm	Merge edges for compacted fonts. A Boolean flag that tells, whether or not the FontCompactor should merge the same contours and glyphs. When merging the data can be more compact and save 10-70% of memory, depending on fonts. But if the font contains too many glyphs, the hash table may consume additional memory, 12 (32-bit) or 16 (64-bit) bytes per each unique path, plus 12 (32-bit) or 16 (64-bit) bytes per each unique glyph.

5.4 Preprocessing Font Textures – gfxexport

When –fonts option is specified, gfxexport can rasterize and export packed font textures, saving them in a user specified format. When loading the GFX file in Scaleform Player, external textures will be automatically loaded and used as a static cache during text rendering.

We do not recommend using exported textures for typical GFx application. However it may be beneficial for low end mobile platforms and in some special cases.

Using gfxexport to generate font textures has the following advantages:

- Saving external texture allows glyph vector data to be striped, saving memory. Glyph data is, however, replaced by the static textures that are loaded instead.
- Loading of texture files may be faster than generating them at load time on CPU-constrained systems.

- Texture files can be converted into compact game-specific formats, loadable by overriding the GFx::ImageCreator. This may be beneficial if you implement your own Render::Renderer that supports those formats directly.

Using pre-generated textures does, however, have disadvantages of lower quality text rendering and potentially longer load times compared to using the dynamic cache. Static text uses mip-maps and thus cannot be hinted; this means that it does not honor the "Anti-alias for readability" setting.

The following command line statement will pre-process 'test.swf' into the 'test.gfx' file, generating additional texture files for all of the embedded fonts.

```
gfxexport –fonts –strip_font_shapes test.swf
```

The –strip_font_shapes option will exclude the embedded font vector data from the resulting gfx file. Although this will save memory, it will make it impossible to fall back to vector rendering for large characters, causing them to degrade in quality when stretched beyond the nominal glyph size of the font texture.

The following table lists the font related options for gfxexport. Options are provided for controlling texture size, nominal glyph size and target file format. Most of these options correspond with glyph packer parameters described in the next section, since glyph packer is used by gfxexport when generating font textures.

Command Line Option	**Behavior**
-fonts	Export font textures. If not specified, font textures will not be generated (allowing for either dynamic cache or packing at load time to be done instead).
-fns <size>	Nominal size of texture glyph in pixels; default value of 48 is used if not specified. Nominal size if the maximum size of one glyph in a texture. Smaller characters are rendered at runtime by using tri-linear mip-map filtering.
-fpp <n>	Space, in pixels, to leave around the individual glyph image. Default value is 3.
-fts <WxH>	The dimensions of the textures that the glyphs get packed into. Default size is 256x256. To specify square texture only one dimension can be specified, e.g.: '-fts 128' is 128x128. '-fts 512x128' specifies rectangle texture.
-fs	Force separate textures for each font. By default, fonts share textures.
-strip_font_shapes	Do not write font glyph shape data into the resulting GFX file.
-fi <format>	Specifies output format for font textures where <format> is one of TGA8 (grayscaled), TGA24 (grayscaled), TGA32 or DDS8. By default, if image

Command Line Option	Behavior
	format (-i option) is TGA then TGA8 is used for font textures; otherwise DDS A8.

Warning: If you plan to use only packed static fonts or glyph packer in your game, you should disable dynamic glyph cache texture allocation as described in section 5.2. If this is not done, the default texture will still be allocated and will remain unused.

5.5 Configuring the Font Glyph Packer

The Font Glyph Packer rasterizes and packs the embedded glyphs when loading the file. Also, it is possible that the fonts are pre-rasterized by the GFxExport. As was mentioned before, the Font Cache Manager can use simultaneously both, dynamic and static mechanisms. For example, it is possible to configure the Font Glyph Packer in such a way that fonts with large character sets will use the dynamic cache, while fonts with Basic Latin only will be pre-rasterized and used statically.

As was described earlier, the general strategy is to use the pre-rasterized static textures when available, otherwise use the dynamic cache if enabled.

The font glyph packer is disabled by default, which means Scaleform will use the dynamic cache for all embedded fonts unless pack parameters are explicitly created and set on the loader. This can be done with the following statement:

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

However, when using GFxExport (and the respective .GFx files) the glyphs can be pre-rasterized. The above call only means "do not pack any glyphs when loading"; if the glyphs are pre-packed with GFxExport they will be used as the static cache.

In cases when the static cache is desirable (small total number of embedded glyphs) it can be configured in the following way.

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

`SetUseSeparateTextures()` controls packing. If it's true, the packer will use separate textures for every font. Otherwise, it tries to pack all the glyphs as compactly as possible. Using separate textures may reduce the

number of switches made between textures, and so, the number of draw primitives. But it typically increases the amount of system used and video memory. By default it's false.

`SetGlyphCountLimit()` controls the maximal number of glyphs to the packed. By default it's 0, which means "no limit". If the total number of embedded glyphs in a font exceeds this limit, the font will not be packed. The parameter makes sense when using Asian languages together with Latin-based or others. If you set this limit to 500, most of the Asian fonts will be cached dynamically (if the dynamic cache is enabled), while using the static textures for fonts with a small number of glyphs.

`SetTextureConfig()` controls the texturing parameters, which are:

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

The above values are used by default.

`NominalSize` - the nominal size in pixels of the anti-aliased glyphs stored in the texture. This parameter controls how large the very largest glyphs will be in the texture; most glyphs will be considerably smaller. This is also the parameter that controls the tradeoff between texture RAM usage and the sharpness of large text. Note that it's called "NominalSize". Unlike the dynamic cache, where the glyphs are stretched to the glyph slots, the static cache packs the glyphs of different size, using the actual bounding boxes. The NominalSize is exactly the same value as if the text height is set in pixels. It also means the dynamic cache has slightly better "resolution capacity" than the static one.

`PadPixels` - how much space to leave around the individual glyph image. This should be at least 1. The bigger it is, the smoother the boundaries of minified text will be, but the more texture space is wasted.

`TextureWidth, TextureHeight` - the dimensions of the textures that the glyphs get packed into. These values must be powers of two.

Several use case scenarios and their meanings are listed below:

1) Everything is set by default. The dynamic cache is enabled; the Font Glyph Packer is not used. The dynamic cache is used in all cases except for pre-rasterized glyphs textures loaded from preprocessed .GFX files.

2)
```
Ptr<FontPackParams> packParams = *new FontPackParams();
Loader.SetFontPackParams(packParams);
...
renderer->GetGlyphCacheConfig()->SetParams(Render::GlyphCacheParams(0));
```
The font glyph packer is always used in all cases. The dynamic cache is disabled.

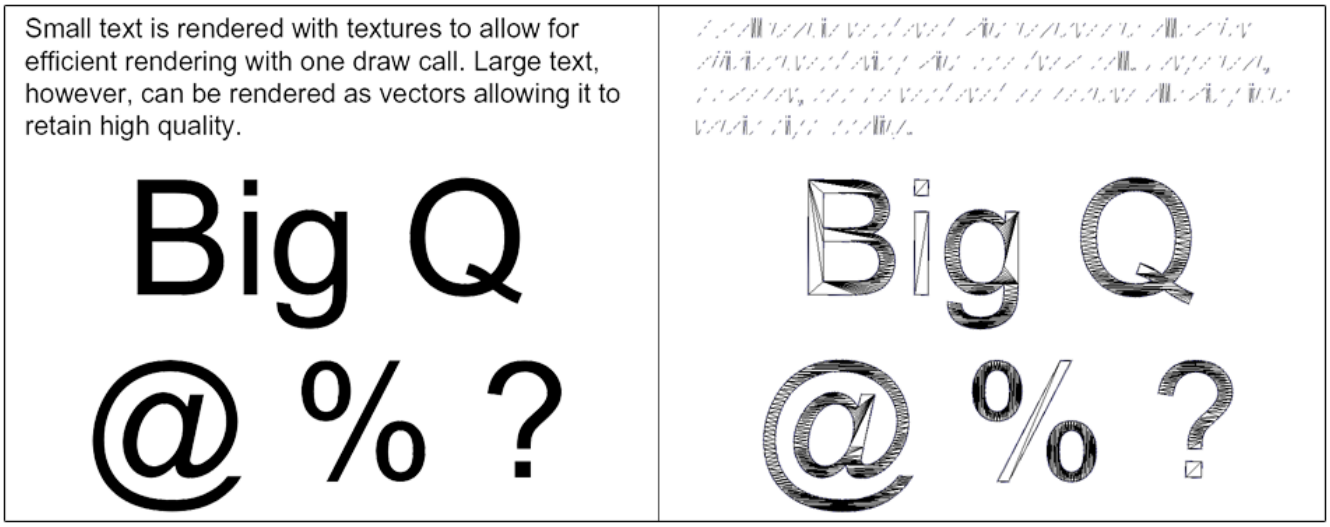
3)
```
Ptr<FontPackParams> packParams = *new FontPackParams();
Loader.SetFontPackParams(packParams);
Loader.GetFontPackParams()->SetGlyphCountLimit(500);
```
The Font Glyph Packer and the dynamic cache both are used. Here fonts with 500 embedded glyphs or less use pre-rasterized static textures, the other ones use the dynamic cache.

Also, it is necessary to mention that several of these capabilities are available only with the dynamic cache. For example, text optimized for readability, with automatic pixel grid fitting (so called auto-hinting) works only with the dynamic cache. Any effects, such as blur, shadows and glow are also available with the dynamic cache only. In general, the glyph packer and static cache are good for low budget systems with poor performance.

5.6 Controlling Vectorization

As described earlier in this document, tessellated vector shapes are used in Scaleform as a fallback when rendering large glyphs that do not fit into the texture cache. Since there is no limit on the size of individual text characters in Flash, the amount of memory required for rendering them through textures becomes prohibitive beyond a certain point. One way to address this problem is to limit glyph bitmap size and use bilinear filtering from that point on. Unfortunately, doing so will cause rendering quality to quickly degrade.

To allow rendering large high-quality text, Scaleform is capable of switching glyph rendering to triangulated shapes and then leveraging the edge anti-aliasing technology. In most cases, the switch will not be noticeable by the user, as glyph shapes will retain their smooth edges when text characters grow in size. The following figure demonstrates the difference between texture and triangle mesh rendered text. In Scaleform Player, the executable Ctrl+W key toggles the wire-frame mode, allowing users to see how the rendering is done.



Although triangle-rendered glyphs look good, they require more processing time during rendering due to the increased triangle and draw primitive counts. When the dynamic cache is used the maximum texture glyph height is determined by the MaxSlotHeight value of the cache, modifiable through Render::GlyphCacheConfig::SetParams. If the glyph to be displayed fits within the texture slot it is rendered through a texture, otherwise vector rendering is used. Since the rendering technique used is determined per glyph, a single line of text can contain both bitmap and vector symbols, when its pixel height is close to the maximum cache slot height. Due to sub-pixel precision, the differently rendered symbol types are almost indistinguishable from each other.

The static cache works differently. In a static cache all glyphs are pre-rasterized based on the specified nominal font size and scaled using a tri-linear filter. Since the nominal font size is fixed, the text rendering approach selection is done based on the nominal font size and not the height of the individual glyphs.

For both dynamic and static cache, developers can control the point at which the texture to vector rendering switch occurs by modifying the GlyphCacheParams:: MaxRasterScale value. MaxRasterScale assigns the multiplier of maximum texture slot size, in pixels, after which the switch to vectors will occur. The default value for MaxRasterScale is 1.0f. The value of 1.25, for example, would mean that the player will render text using textures unless the on-screen glyph size becomes greater than 1.25 times the nominal glyph size stored in textures. With the default nominal size of 48 pixels, vector rendering will be used for glyphs bigger than 60 pixels on screen.

If the MaxRasterScale value is set high enough vector rendering will never be used. Vector rendering is also disabled for GFX files produced by executing the gfxexport tool with the -strip_font_shapes option. In the later case, font glyph data no longer exists in the file so vectorization is not possible.

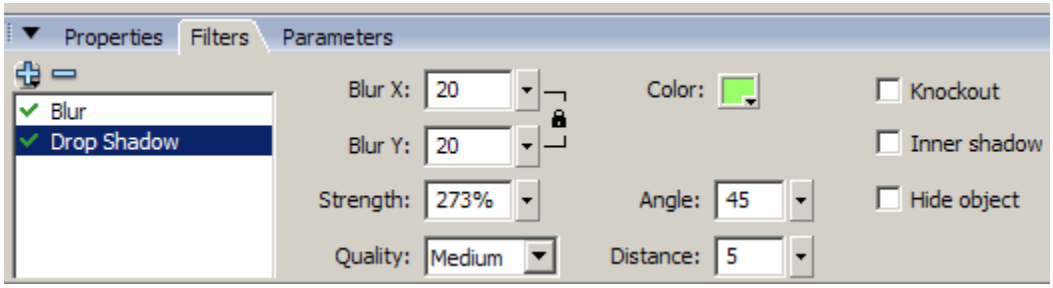
6 Part 5: Text Filter Effects and ActionScript Extensions

When the dynamic glyph cache is enabled, Scaleform 3 and higher version supports Blur, Drop Shadow, and Glow filter effects applied to text. Text filter support requires dynamic cache; when using glyph packer or static font textures filter effects are not supported. In the current version of the player filters will only work when applied directly to text fields, they will have no effect when applied to movie clips.

Although Scaleform text filters work similar to Flash there are a number of differences. In Adobe Flash, filters are applied consecutively to the raster result of the entire text field. This means that there can be Blur, Drop Shadow, and Glow filters applied one after another. This method has a good degree of flexibility, but it is computationally expensive. In contrast to Flash, filter support in Scaleform is limited, but very fast. Combined with the dynamic adaptive glyph cache, filters work almost as fast as regular text. Despite limited filter support, Scaleform offers very good capabilities for making soft shadow and glow effects.

6.1 Filter Types, Available Options and Restrictions

In Flash Studio, filters can be added the text field one at a time.



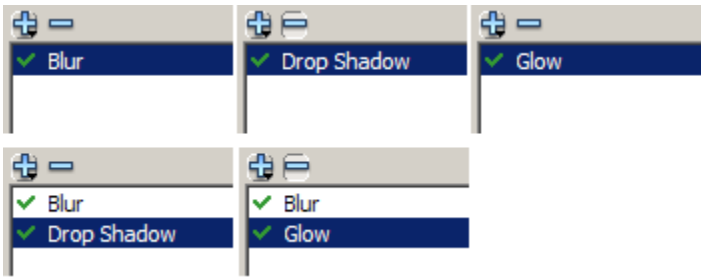
The major difference between Flash and Scaleform filters is that Scaleform creates and stores in the cache bitmap copies of blurred glyphs, where Flash applies the filters to the resulting text field.

Scaleform supports only Blur and Drop Shadow filters. The Glow filter is actually a sub-set of the Drop Shadow filter. Unlike Flash the filters are not applied consecutively, instead, they work independently, as two different layers. In Scaleform, only one Drop Shadow or Glow filter is taken into account, whichever appears last in the filter list. Plus, an optional Blur filter can be applied to the text itself. The general algorithm is as follows.

- Iterate through the filter list;
- If “Glow” or “Drop Shadow” store the shadow filter parameters;
- Each “Glow” or “Drop Shadow” overrides the one found previously in the list;
- If “Blur” store the blurring filter parameters;

- Each “Blur” overrides the one found previously in the list.

As a result, the combinations of filters that make sense are:

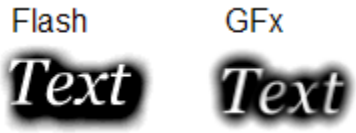


If you add Drop Shadow and Glow filters to the list, only the last one will be taken into account.

Rendering in Scaleform, also differs from Flash. First, the “shadow” or “glow” layer is rendered. Then the text itself is rendered over the shadow, with an optional “Blur” filter applied to the original glyphs. If the “shadow” or “glow” filters have “Knockout” or “Hide Object” flags, the text (the second layer) is not rendered. “Inner Shadow” and “Inner Glow” are not support by Scaleform at this time.

Due to the nature of the algorithm, filters in Scaleform have certain restrictions. The maximal Blur X and Blur Y values are restricted to 15.9 pixels. The maximal shadow or glow strength is also restricted to 1590%.

The “Knockout” option also works differently. In Flash the original image is subtracted from the shadow as a whole, preserving the shadow offset. In Scaleform it is done “glyph-by-glyph”, ignoring the offset. Also, if the shadow or glow radius is big the glyph images may overlap and “obscure” each other:

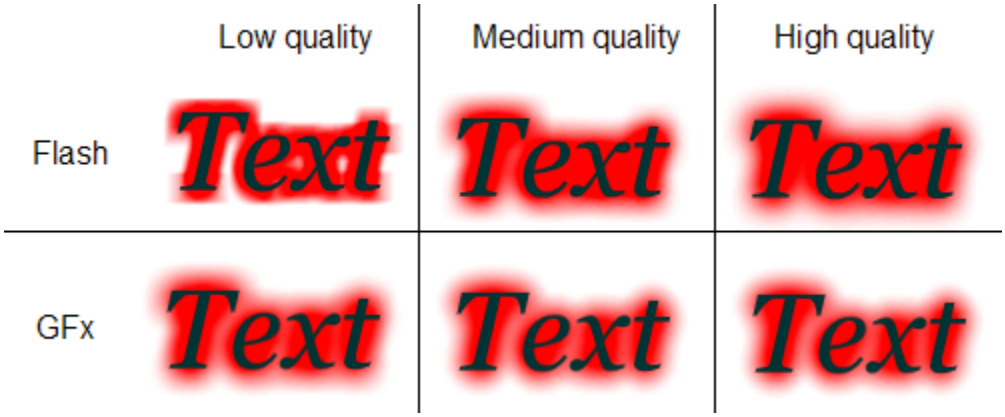


As a result, the “Knockout” option is basically only useful for the “Glow” filter (or for the “Drop Shadow” with zero offset) with a small filter radius.

6.2 Filter Quality

Flash uses a simple box filtering to blur the images. “Quality” controls the number of passes the filter performs, which heavily affects the resulting image. For example, the blurring filter 3x3 with low quality simply calculates average values of 3x3 pixel areas. Medium quality means the same filter is applied twice; and high quality applies this filter 3 times. It means the quality heavily affects the visual radius of the filter. Because of different approaches, the visual result is different, but similar. In fact, a simple one-pass box filter produces too poor a result.

Unlike Flash, Scaleform uses a fair Gaussian Blur filter and a smart recursive algorithm, whose speed does not depend on the filter radius. There are only two levels of quality in Scaleform: Low and High and they produce very similar visual results. The low quality is used only when "Quality: Low" is set in the filter panel. Otherwise Scaleform uses the high quality filter. The difference between them is the high quality filter can operate with fractional values of radii (sigma), while the low quality filter has integer radii. In most cases Scaleform simulates the fractional radius with appropriate scaling of the glyphs, but in case the text field is anti-aliased for readability the low quality shadow may look slightly inaccurate. The general recommendation is to use the high quality filter for small text optimized for readability.



As you can see, in Flash the difference is very visible; while in Scaleform, it is subtle. Only "Low quality" differs from the others in Scaleform. "Medium quality" and "High quality" generate identical results.

The low quality filter works faster, but the glyph cache system eliminates the difference. Still, it is definitely better to use the low quality filters on low performance systems, especially if there is no hardware floating point available.

The high quality filter requires floating point calculations and uses a recursive implementation described here: http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Smoothin.html.

6.3 Filter Animation

It is possible to animate the shadow effects using the Flash timeline as well as Scaleform ActionScript extensions. However, it is necessary to understand the cost of the animation operations. When changing the radius, strength, or quality, Scaleform has to re-generate the glyph images and store them in the cache. It will result in more frequent cache update. In fact, changing values mentioned above is equivalent to increasing the multiplicity of the alphabet. Each version has to be stored in the cache. Instead, changing the color (including alpha), angle, and the distance does not affect the performance. For example, if you want to

fade in and out the shadow or glow, it's better to animate the color translucency (alpha) instead of changing the radius and/or strength.

6.4 Using filters from ActionScript

Scaleform support standard filter classes (such as DropShadowFilter, BlurFilter, ColorMatrixFilter, BevelFilter) for dynamic/input text fields for both AS2 and AS3. Refer to Flash documentation for further details.

7 Translator

Translator class is used for localization of dynamic text fields. Using the Translator use may translate the text from one language to another, control word wrapping and control right-to-left text rendering.

7.1 Installing Translator

To install Translator user should:

- Inherit from Translator class and override GetCaps method to return 'Cap_BidirectionalText' flag, for example:

```
#include "GFx_Loader.h"

class Translator : public SF::GFx::Translator
{
public:
        ...
};
```

- Create an instance of the custom Translator class and set it to the Loader object:

```
SF:GFx::Loader loader;
...
loader.SetTranslator(Ptr<SF:GFx::Translator>(*new Translator()));
```

7.2 Translation of Text

Translation is performed through the Translate virtual function which is expected to fill in a buffer with a translated string. Developers usually override Translate method of this class.

The translation interface will be used to look up dynamic text in all of the text fields. For this reason, it is often wise to differentiate translatable text fields in the SWF file from the dynamic ones by prefixing the content with a special tag, such as a '$'. If this is done, translation implementation can translate all of the prefixed strings, while leaving the non-prefixed ones unchanged.

Scaleform translation interface can both receive and generate HTML formatted strings. By default, HTML tags are stripped out before calling Translate, but this behavior might be changed by returning bit Cap_ReceiveHtml set from GetCaps virtual function.

Translate method receives a pointer to TranslateInfo class. This class provides data exchange between Scaleform core and users code. Method TranslateInfo::GetKey() returns original text in UTF-8 encoding. Methods TranslateInfo::SetResult and TranslateInfo::SetResultHtml should be used to return plain and

HTML translated texts respectively. These methods exist in two versions each for Unicode (UCS-2) and for UTF-8 encodings.

```
class TranslatorImpl : public Translator
{
    virtual unsigned GetCaps() const
    {
        // Kill ending new line that gets generated if HTML text field content is used to
        // create a translation key.
        return Cap_StripTrailingNewLines;
    }

    virtual bool Translate(TranslateInfo* ptranslateInfo)
    {
        // Translate '$hi' into 'Hello!'. Users can perform string lookup
        // and substitution here.
        const char* pkey = ptranslateInfo->GetKey();
        if (pkey[0] == '$')
        {
            String key(pkey);
            if (pkey == "$hi")
            {
                ptranslateInfo->SetResult(L"Hello!");
            }
        }
    }

};

// Use class when setting up a loader or a movie view.
Ptr<TranslatorImpl> ptranslator = *new TranslatorImpl();
loader.SetTranslator(ptranslator);
```

7.3 Controlling Word Wrapping

It is not a secret that different languages have different rules for word wrapping text. For most of the languages the algorithm is quite simple: it wraps at the nearest to the line's end space. However, Asian languages, such as Chinese, Japanese and Korean and others have their own rules.

Scaleform has built-in word wrapping rules for Chinese, Japanese and Korean languages, but it is also possible to implement any custom rules with using GFx::Translator API.

7.3.1 Word Wrapping for Asian Languages (Chinese, Japanese, Korean)

To use built-in Chinese, Japanese or Korean word wrapping it is necessary to inherit from GFx::Translator and to set WWMode to corresponding WWT_<> value (see below).

A constructor of the inherited class might have a parameter 'wwMode' that specifies the actual word wrapping mode. Or, the desired word wrapping mode can just be passed to the base class constructor. Possible values are:

```
    enum WordWrappingTypes
    {
        WWT_Default       = 0,
        WWT_Hyphenation,
        WWT_Custom,

        WWT_Korean,
        WWT_Japanese,
        WWT_Chinese
    };
```

WWT_Default
Custom word wrapping is off, OnWordWrapping will not be invoked, the default simple algorithm to break at whitespaces will be used.

WWT_Hyphenation
Turns on a simple hyphenation; for demo purposes only.

WWT_Chinese
Turns on Chinese word wrapping rules.

WWT_Japanese
Turns on Japanese prohibition rule.

WWT_Korean
Turns on Korean-specific word wrapping rules.

WWT_Custom
Turns on custom user-defined word wrapping. User should implement his own version of OnWordWrapping virtual method.

Here is an example of the Translator with the ability to specify the desired word wrapping rules:

```
#include "GFx_Loader.h"
using namespace Scaleform;

class TranslatorImpl : public GFx::Translator
{
public:
    // The 'wwMode' parameter can be one of the WWT_<> value.
    // Let say, by default we need Japanese word
    // wrapping here...
    TranslatorImpl(unsigned wwMode = GFx::Translator::WWT_Japanese) :
      GFx::Translator(wwMode) {}
};
```

To create this translator you need to provide word wrapping mode, for example, for Korean:

```
#include "GFx_Loader.h"
using namespace Scaleform;
....

GFx::Loader Loader;
....
Loader.SetTranslator(Ptr<GFx::Translator>(*new TranslatorImpl(GFx::Translator::WWT_Korean)));
```

It is possible to change word wrapping mode later, by assigning appropriate value to WWMode member of the GFx::Translator class:

```
class TranslatorImpl : public GFx::Translator
{
public:
    ....
    void SetWWMode(unsigned wwmode) const
    {
        WWMode = wwmode;
    }
};
```

7.3.2 Custom Word Wrapping

If the existing algorithms of word wrapping are not enough then it is possible to implement custom word wrapping algorithm. To do this, the GFx::Translator exposes the virtual method "OnWordWrapping" that should be overloaded:

```
virtual bool OnWordWrapping(LineFormatDesc* pdesc);
```

OnWordWrapping is a virtual method, a callback, which is invoked once a necessity of word wrapping for any text field is detected. This method is invoked only if custom word wrapping is turned on by using the GFx::Translator(wwMode) constructor, where wwMode should be set to GFx::Translator::WWT_Custom value:

```
class TranslatorImpl : public GFx::Translator
{
public:
    // Turning custom word wrapping on.
    TranslatorImpl() :
      GFx::Translator(GFx::Translator::WWT_Custom) {}

    // A callback that is invoked at word wrap event.
    virtual bool OnWordWrapping(LineFormatDesc* pdesc);
};
```

The parameter of the OnWordWrapping method is a pointer to the LineFormatDesc structure for calculating the word wrap position. The LineFormatDesc structure provides information of the line text to be formatted like the length of the text, text position etc:

```
struct LineFormatDesc
{
    const wchar_t*  pParaText;
    UPInt           ParaTextLen;
    const float*    pWidths;
    UPInt           LineStartPos;
    UPInt           NumCharsInLine;
    float           VisibleRectWidth;
    float           CurrentLineWidth;
    float           LineWidthBeforeWordWrap;
    float           DashSymbolWidth;
    enum
    {
```

```
        Align_Left      = 0,
        Align_Right     = 1,
        Align_Center    = 2,
        Align_Justify   = 3
    };
    UInt8           Alignment;

    UPInt           ProposedWordWrapPoint;

    bool            UseHyphenation;
};
```

Here is the description of the structure's members:
const wchar_t pParaText;*
[in] text of the current paragraph, wide-characters are used.

UPInt ParaTextLen;
[in] length of the paragraph text, in characters.

UPInt NumCharsInLine;
[in] number of characters currently in the line.

const float pWidths;*
[in] an array of line widths, in pixels, before the character at the corresponding index. The size of the array is NumCharsInLine + 1. Note, this is not the array of character widths. For example, there is a line that contains three characters: ABC. The NumCharInLine will be equal 3, the size of the pWidths will be 4; the pWidth[0] will be always 0 (since there are no characters before the 'A'), the pWidth[1] will contain the width of 'A' symbol, pWidths[2] will contain the width of 'A' PLUS the width of 'B', and, finally, pWidths[3] will contain total width of the line (width of 'A' PLUS width of 'B' PLUS width of 'C').

UPInt LineStartPos;
[in] the text position of the first character in the line. ParaTextLen[LineStartPos] might be used to get the value of this character.

float VisibleRectWidth;
[in] width, in pixels, of the textfield rectangle (the inner text area, excluding borders). This width might be used in calculation of word wrapping position: the total width of line should not exceed this width.

float CurrentLineWidth;
[in] current line width, in pixels.

float LineWidthBeforeWordWrap;
[in] line width before the ProposedWordWrapPoint (see below), in pixels. For example, if line is "ABC DEF" and ProposedWordWrapPoint = 3 (space) then LineWidthBeforeWordWrap will contain the width of "ABC" (w/o space) part of the line.

float DashSymbolWidth;
[in] supplementary member, width of the hyphen symbol, in pixels. It might be used to calculate hyphenation.

UInt8 Alignment
[in] The alignment of the line. Can be one of the following values:

```
struct LineFormatDesc
{
    ....
    enum
    {
        Align_Left      = 0,
        Align_Right     = 1,
        Align_Center    = 2,
        Align_Justify   = 3
    };
    ....
};
```

UPInt ProposedWordWrapPoint;
[in,out] an index in the line of the proposed word wrap position. For example, if the line text is "ABC DEF" and only "ABC DE" fits in VisibleRectWidth then the ProposedWordWrapPoint will be equal to 3. Note, this is the index in line, not in text (pParaText), not in line. Use LineStartPos to calculate the proposed word wrapping position in the text. The user's OnWordWrapping method should change this member if it is necessary to change the word wrapping position according to custom rules.

bool UseHyphenation;
[out] the user's implementation of the OnWordWrapping method may set this to true to indicate to put hyphen symbol at the word wrapping position. This might be useful for implementing hyphenation.
Note, all the members marked as "[in]" are used as input values only and shouldn't be modified. Only members markes as "[out]" or "[in,out]" might be modified.

7.3.3 How it Works

When Scaleform detects necessity of word wrapping and if custom word wrapping is turned on, then Scaleform fills the LineFormatDesc structure and passes it to OnWordWrapping translator's method. This structure already has so called "proposed word wrapping position". If the user decides that it is not necessary to perform any custom logic for this particular word wrap event she may return "false" from OnWordWrapping and that position will be used. Otherwise, the user should modify the proposed word wrapping position (stored in ProposedWordWrapPoint member) and the OnWordWrapping implementation should return "true".

7.3.4 Examples of Custom Word Wrapping Logic

Here is an example of the Translator with the custom word wrapping implementation. The OnWordWrapping method implements word wrapping at spaces, dots, commas and dashes.

```
class TranslatorImpl : public GFx::Translator
{
public:
    // Turning custom word wrapping on.
    TranslatorImpl() :
        GFx::Translator(GFx::Translator::WWT_Custom) {}

    // an example of custom word wrapping
    bool OnWordWrapping(LineFormatDesc* pdesc)
    {
        // check if we deal with Asian languages; if so, use
        // the default implementation.
        if (WWMode != WWT_Custom)
            return Translator::OnWordWrapping(pdesc);
        if (pdesc->NumCharsInLine == 0)
            return false; // skip empty lines
        // implementing word wrapping at spaces, dots, commas and dashes.
        // start from NumCharsInLine.
        // search for desired word breaker backward
        UPInt linePos = pdesc->NumCharsInLine - 1;
        UPInt textPos = pdesc->LineStartPos + linePos;
        for (SPInt tpos = SPInt(textPos), lpos = SPInt(linePos);
            tpos >= 0 && lpos >= 0;
            --tpos, --lpos)
        {
            int isSpace = SFiswspace(pdesc->pParaText[tpos]) ? 1 : 0;
            if (isSpace ||
                pdesc->pParaText[tpos] == '.' ||
                pdesc->pParaText[tpos] == ',' ||
                pdesc->pParaText[tpos] == '-')
            {
                // found our spacer;
                // check if width fits our textfield.
                // we will want to exclude the whitespace from the
                // widths, but include other spacers (so, they fit
                // the line).
                if (pdesc->pWidths[lpos + (1 - isSpace)] <= pdesc->VisibleRectWidth)
                {
                    // it fits, modify the ProposedWordWrapPoint and return true;
                    // otherwise, continue.
                    pdesc->ProposedWordWrapPoint = lpos + (1 - isSpace);
                    return true;
                }
            }
        }
        // nothing was found, return false.
        return false;
    }
};
```

An example of hyphenation:

```
class TranslatorImpl : public GFx::Translator
{
public:
    // Turning custom word wrapping on.
```

```
    TranslatorImpl() :
        GFx::Translator(GFx::Translator::WWT_Custom |
                        GFx::Translator::WWT_Hyphenation) {}

    // an example of custom word wrapping with hyphenation
    bool OnWordWrapping(LineFormatDesc* pdesc)
    {
        if ((WWMode & WWT_Hyphenation))
        {
            if (pdesc->ProposedWordWrapPoint == 0)
                return false;
            const wchar_t* pstr = pdesc->pParaText + pdesc->LineStartPos;
            // determine if we need hyphenation or not. For simplicity,
            // we just will put dash only after vowels.
            UPInt hyphenPos = pdesc->NumCharsInLine;
            // check if the proposed word wrapping position is at the space.
            // if so, this will be the ending point in hyphenation position search.
            // Otherwise, will look for the position till the beginning of the line.
            // If we couldn't find appropriate position - just leave the proposed word
            // wrap point unmodified.
            UPInt endingHyphenPos = (SFiswspace(pstr[pdesc->ProposedWordWrapPoint - 1])) ?
                pdesc->ProposedWordWrapPoint : 0;
            for (; hyphenPos > endingHyphenPos; --hyphenPos)
            {
                if (Text::WordWrapHelper::IsVowel(pstr[hyphenPos - 1]))
                {
                    // check if we have enough space for putting dash symbol
                    // we need to summarize all widths up to
                    // hyphenPos + pdesc->DashSymbolWidth
                    // and this should be less than view rect width
                    float lineW = pdesc->pWidths[hyphenPos - 1];
                    lineW += pdesc->DashSymbolWidth;
                    if (lineW < pdesc->VisibleRectWidth)
                    {
                        // ok, looks like we can do hyphenation
                        pdesc->ProposedWordWrapPoint = hyphenPos;
                        pdesc->UseHyphenation = true;
                        return true;
                    }
                    else
                    {
                        // oops, we have no space for hyphenation mark
                        continue;
                    }
                    break;
                }
            }
        }
        return false;
    }
};
```

An example of Translator with handling Asian languages:

```
class TranslatorImpl : public GFx::Translator
{
public:
    // Turning custom word wrapping on.
    TranslatorImpl(unsigned wwMode) :
        GFx::Translator(wwMode) {}

    // an example of custom word wrapping with Asian langauges
    bool OnWordWrapping(LineFormatDesc* pdesc)
```

```
    {
        if (WWMode == WWT_Default)
            return false;
        if ((WWMode & (WWT_Asian | WWT_NoHangulWrap | WWT_Prohibition)) &&
             pdesc->NumCharsInLine > 0)
        {
            UPInt wordWrapPos = Text::WordWrapHelper::FindWordWrapPos
                (WWMode,
                pdesc->ProposedWordWrapPoint,
                pdesc->pParaText, pdesc->ParaTextLen,
                pdesc->LineStartPos,
                pdesc->NumCharsInLine);
            if (wordWrapPos != SF_MAX_UPINT)
            {
                pdesc->ProposedWordWrapPoint = wordWrapPos;
                return true;
            }
            return false;
        }
        return false;
    }
};
```

8 Right to Left Text Support

Since version 4.3, Scaleform supports rendering right-to-left (RTL) text. RTL text rendering is used for Arabic and Hebrew languages. There are several limitations to the RTL text support in Scaleform:

- Editing and selection of RTL text is not supported;
- Some text position-related functions do not work correctly, such as getCharBoundaries();
- Arabic shaping (changing letter's glyph **depending on where the letter is located**) is not implemented, but it is possible for customer to implement his own algorithm via Translator::OnBidirectionalText callback (see below for details).
- Arabic ligaturing is not implemented, but it is possible for customer to implement his own algorithm via Translator::OnBidirectionalText callback (see below for details).
- Mirroring is not implemented, but it is possible for customer to implement his own algorithm via Translator::OnBidirectionalText callback (see below for details). Currently, mirroring can be implemented only by replacing one glyph by another (for example ']' can be replaced by '['). If there is no corresponding mirrored glyph in the font then currently mirroring for such glyphs is not possible (for example, the integral math symbol).

8.1 Enabling RTL Text Rendering

To enable RTL text rendering the following should be done:

- Scaleform should be compiled with option GFX_ENABLE_BIDIRECTIONAL_TEXT enabled in GFxConfig.h (it is so by default);
- Inherit from Translator class and override GetCaps method to return 'Cap_BidirectionalText' flag, for example:

```
#include "GFx_Loader.h"

class Translator : public SF::GFx::Translator
{
public:
       virtual unsigned GetCaps() const { return Cap_BidirectionalText; }
};
```

- Create an instance of the custom Translator class and set it to the Loader object:

```
SF:GFx::Loader loader;
...
loader.SetTranslator(Ptr<SF:GFx::Translator>(*new Translator()));
```

8.2 Translator::OnBidirectionalText

Scaleform contains basic implementation of RTL text re-ordering. It is able to re-order mixed RTL and ordinary text correctly. However, the algorithm is not fully Unicode-complaint. User can re-implement re-ordering algorithm by overriding the Translator::OnBidirectionalText virtual method. This method can be also used to implement re-shaping, ligaturing and mirroring (with some limitations discussed earlier). The signature of OnBidirectionalText is the following:

```
virtual bool OnBidirectionalText(const wchar_t* text, UPInt textLen, wchar_t* newText,
                                 unsigned* indexMap, bool* mirrorBits);
```

This method is called once per paragraph when bidirectional text is detected (Arabic or Hebrew) and if support of bidirectional text is enabled (see above). The 'paragraphs' are text lines separated by new-line symbols.

This method receives 'text' (a wchar_t pointer to the paragraph text) and 'textLen' (the number of characters in the 'text') as input parameters.
The 'newText', 'indexMap' and 'mirrorBits' parameters are just pre-allocated buffers to fill out. The sizes of these buffers are textLen*sizeof(type).

The 'newText' should receive changed text: it should be re-ordered and optionally re-shaped/ligatured/mirrored. To implement re-shaping / ligaturing user should replace codes from 'text' by the codes corresponding to the desired form. New codes should be put into 'newText' buffer. For example, the letter with **code 0x621 might be replaced by code 0xFE87**, if this letter is in the middle of the word. Similarly, the character '[' might be replaced by ']' for mirroring.

The 'indexMap' array should contain mappings between old (original) and new (re-ordered) indices. The index in this array represents indices of characters in the original 'text'. The values at the indices represent new locations of the characters. For example, if original character at index 0 was relocated to index 10, then indexMap[0] = 10. It is very important to fill this array correctly, otherwise ASSERT may be thrown in Text::DocView::CheckConsistency method.

'mirrorBits' is an array of booleans, where each element represents corresponding position in the re-ordered text; if the element is set to 'true' then the glyph should be rendered mirrored. *Not supported yet.*

This method should return 'true' if method was successful and the core should use contents of the buffers; false, if no changes to the text field should be applied.