

# Autodesk® Scaleform®

## 폰트 및 텍스트 설정 개요

이 문서는 Scaleform 4.2 에서 사용되는 폰트 및 텍스트 렌더링을 다루며 국제화를 위해 예술 자산 및 Scaleform C++ API 모듈을 어떻게 설정하는지에 대한 세부사항을 제공한다.

저자: Maxim Shemanarev, Michael Antonov

버전: 2.2

최종 편집: 2012 년 6 월 21 일

# Copyright Notice

## Autodesk® Scaleform® 4.2

© 2012 Autodesk, Inc. All rights reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

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

The following are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries: 123D, 3ds Max, Algor, Alias, AliasStudio, ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Homestyler, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, Beast (design/logo) Built with ObjectARX (design/logo), Burn, Buzzsaw, CAiCE, CFdesign, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Creative Bridge, Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWFX, DXF, Ecotect, Evolver, Exposure, Extending the Design Team, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, Freewheel, GDX Driver, Green Building Studio, Heads-up Design, Heidi, Homestyler, HumanIK, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Instructables, Instructables (stylized robot design/logo), Inventor, Inventor LT, Kynapse, Kynogon, LandXplorer, Lustre, MatchMover, Maya, Mechanical Desktop, MIMI, Moldflow, Moldflow Plastics Advisers, Moldflow Plastics Insight, Moondust, MotionBuilder, Movimento, MPA, MPA (design/logo), MPI (design/logo), MPX, MPX (design/logo), Mudbox, Multi-Master Editing, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, Pixlr, Pixlr-o-matic, PolarSnap, Powered with Autodesk Technology, Productstream, ProMaterials, RasterDWG, RealDWG, Real-time Roto, Recognize, Render Queue, Retimer, Reveal, Revit, RiverCAD, Robot, Scaleform, Scaleform GFx, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Sparks, SteeringWheels, Stitcher, Stone, StormNET, Tinkerbox, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, U-Vis, ViewCube, Visual, Visual LISP, Vtour, WaterNetworks, Wire, Wiretap, WiretapCentral, XSI.

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

### Disclaimer

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

연락처:

---

문서	폰트 및 텍스트 설정 개요
주소	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
홈페이지	<a href="http://www.scaleform.com">www.scaleform.com</a>
이메일	<a href="mailto:info@scaleform.com">info@scaleform.com</a>
직통전화	(301) 446-3200
팩스	(301) 446-3199

# 목차

<b>1</b>	<b>소개: Scaleform 에서의 폰트 .....</b>	<b>4</b>
1.1	플래시 텍스트와 폰트 기본 .....	5
1.1.1	텍스트 필드 .....	6
1.1.2	문자 내장 .....	8
1.1.3	내장 폰트를 통한 메모리 사용 .....	9
1.1.4	폰트 메모리 사용 제어 .....	9
1.2	게임 UI 용 폰트 결정 .....	10
<b>2</b>	<b>제 1 부: 게임 폰트 라이브러리 생성 .....</b>	<b>13</b>
2.1	폰트 기호 .....	13
2.1.1	폰트 기호의 불러오기 및 내보내기 .....	14
2.1.2	내보낸 폰트 및 문자 내장 .....	16
2.2	gfxfontlib.swf 생성 단계 .....	17
<b>3</b>	<b>제 2 부: 국제화 접근 선택 .....</b>	<b>19</b>
3.1	불러온 폰트 교체 .....	19
3.1.1	국제 텍스트 설정 .....	22
3.1.2	Scaleform Player 에서의 국제화 .....	23
3.1.3	장치 폰트 에뮬레이션 .....	24
3.1.4	폰트 라이브러리를 생성하기 위한 단계별 가이드 .....	26
3.2	커스텀 자산 생성 .....	28
<b>4</b>	<b>제 3 부: 폰트 소스 설정 .....</b>	<b>30</b>
4.1	폰트 록 업 순서 .....	31
4.2	Gfx::FontMap .....	32
4.3	Gfx::FontLib .....	33

4.4	GfX::FontProviderWin32 .....	34
4.4.1	네이티브 힌트 텍스트 사용 .....	34
4.4.2	네이티브 힌트 설정.....	36
4.5	GfX::FontProviderFT2 .....	37
4.5.1	FreeType 폰트를 파일로 매핑 .....	38
4.5.2	폰트를 메모리로 매핑 .....	39
<b>5</b>	<b>제 4 부: 폰트 렌더링 설정.....</b>	<b>41</b>
5.1	문자 캐시(Glyph cache) 구성하기.....	42
5.2	동적 폰트 캐시 사용 .....	43
5.3	폰트 압축기- gfxexport 사용하기.....	46
5.4	폰트 텍스처 전처리 - gfxexport.....	46
5.5	폰트 글리프 벡터 설정.....	48
5.6	벡터화 제어 .....	51
<b>6</b>	<b>제 5 부: 텍스트 필터 효과 및 ActionScript 확장.....</b>	<b>53</b>
6.1	필터 형태, 가용한 옵션 및 제약.....	53
6.2	필터 품질.....	55
6.3	필터 애니메이션 .....	56
6.4	ActionScript 에서 필터 사용하기.....	57

## 1 소개: Scaleform 에서의 폰트

Scaleform 는 HTML 포맷으로 변환이 가능한 높은 품질의 텍스트를 제공하는 새로운 유연한 폰트 시스템을 탑재하고 있다. 새로운 시스템을 사용하여, 로컬화를 위해 폰트를 교체하고 로컬에 내장된 텍스트, 공유 GFX/SWF 파일, OS 폰트 데이터 또는 FreeType 2 라이브러리를 포함하는 다른 소스로부터 폰트를 취할 수 있다. 폰트 렌더링 품질 또한 크게 향상되었으며 이를 통해 개발자들이 애니메이션 성능, 메모리 사용 및 텍스트 준비 상태 등을 크게 고려하지 않게 되었다.

Scaleform 폰트 특성 중 다수는 플래시 스튜디오에서 물려 받은 기능성에 의존하며 여기에는 폰트 문자 셋을 SWF 파일에 내장하고 가용한 경우 시스템을 활용하는 것을 포함한다. 그러나 플래시 스튜디오는 주로 개별 파일을 개발하기 위해 고안된 프로그램이므로 폰트 로컬화에 있어선 제한적이다. 특히 플래시는 내장된 폰트나 번역 테이블을 파일간 쉽게 공유 불가능한데 이는 효과적인 메모리 사용 및 게임 자산 개발에 있어 중요한 특성이다. 이에 더해, 플래시는 파일에 내장되지 않은 기타 문자의 폰트 교체를 처리하는데 있어서 시스템 의존적이며 이는 시스템 폰트를 사용할 수 없는 게임 콘솔에선 옵션으로 작용하지 않는다.

Scaleform 은 Gfx::Translator 클래스를 모든 번역 가능한 텍스트를 위한 중앙화 콜 백으로서 사용하고 Gfx::FontLib 클래스를 동적 폰트 매핑에 사용함으로써 이러한 문제를 해결했다. 추가적인 인터페이스를 제공하여 폰트에 사용되는 캐시 메커니즘을 제어하고, 로컬화 과정에서 폰트 이름을 교체하며 적용 가능한 경우 OS 및 FreeType2 폰트 록 업이 가능하다.

Scaleform 폰트 시스템을 효과적으로 사용하기 위해, 개발자들은 플래시 스튜디오와 Scaleform 런타임 모두에 대한 이해가 필요하다. 플래시의 측면을 알게 되면 개발자들은 메모리 사용을 최소화하면서 원하는 품질로 효과적인 렌더링을 할 수 있는 지속적인 탐색 콘텐츠를 개발할 수 있게 된다. Scaleform 폰트 루틴 옵션을 알게 되면 주어진 플랫폼에 개발자들은 가능한 최상의 품질, 성능 및 메모리 사용 특성을 선택할 수 있게 된다.

이 문서는 플래시와 Scaleform 사용 경험이 있는 개발자들에게 게임 UI 폰트 설정에서의 ins and out 을 가르치기 위한 것이다. 나머지 소개 부분은 다음과 같이 이뤄진다.

- “플래시 텍스트와 폰트 기본”에선 플래시에서의 텍스트 및 폰트 사용에 대한 기본 사항을 다룬다. 플래시에 익숙한 개발자들은 TextField 와 문자 내장을 설명하는 첫 두 부분을 건너뛸 수 있다.
- “게임 UI 를 위한 폰트 결정”은 플래시로 게임 UI 를 제작할 때 개발자들이 내릴 필요가 있는 기본 폰트와 관련된 결정을 다룬다. 나머지 문서의 구조에 대해 설명하기 때문에 이 문서를 읽는 모든 개발자들이 읽길 바란다.

## 1.1 플래시 텍스트와 폰트 기본

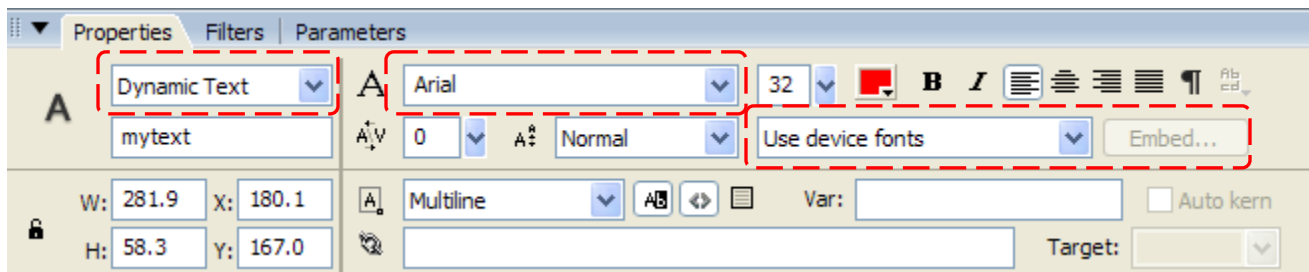
플래시 스튜디오에서 아티스트들은 마우스를 이용해서 텍스트 필드를 그리고 해당 텍스트를 입력함으로써 가시적으로 생성한다. 텍스트 필드에 적용되는 폰트는 아티스트 시스템에 설치된 폰트 목록 중 이름을 선택함으로써 이뤄지며 필요한 폰트가 없는 시스템에서의 콘텐츠 재생을 위해 가능한 옵션을 내장하고 있다.

폰트 이름을 직접적으로 사용하는 것에 더해, 아티스트는 라이브러리에서 폰트 기호를 생성하고 이를 기호명을 기준으로 해서 간접적으로 사용할 수 있다. 문자 내장과 결합되어, 폰트 기호는 휴대가 가능하고 일관적인 게임 UI 자산 개발을 위한 강력한 개념 프레임워크를 생성한다. 나머지 부분에선 텍스트 필드 및 문자 내장에 대한 개념을 간단히 살핀다. 더욱 자세한 설명을 원하는 개발자들은 플래시 스튜디오의 문서 부분을 참고한다.

플래시 폰트 기호 시스템은 편리하고 설정이 쉽다. 그러나 국제화를 위해 폰트를 공유하고 교체하는 데는 사용이 어렵다. 이러한 한계를 어떻게 해결하는가에 대한 사항이 제 1 부 – 게임 폰트 라이브러리 생성에 나온다.

### 1.1.1 텍스트 필드

아티스트들은 Text Tool 을 선택하고 해당 단계에서 사각형 영역을 그림으로써 플래시에서 텍스트 필드를 생성할 수 있다. 텍스트 형태, 폰트, 크기 및 스타일을 포함한 텍스트의 다양한 속성은 Text Field 속성 패널에서 설정할 수 있으며 이는 플래시 스튜디오 하단에 위치하고 있다. 패널은 아래와 같이 생겼다.



텍스트 필드의 옵션이 다양하지만 여기에서 논의할 중요한 특성이 원으로 표시되어 있다. 텍스트 필드에 있어 가장 중요한 옵션은 좌측 상단에 있는 텍스트 형태로 다음 중 하나의 값을 가질 수 있다.

- 정적 텍스트
- 동적 텍스트

- 입력 텍스트

게임 개발에 있어 동적 텍스트 속성은 가장 일반적으로 사용되는 필드 형태가 될 것인데, 그 이유는 ActionScript 를 통한 콘텐츠 수정을 지원하고, 동시에 사용자가 인스톨하는 Gfx::Translator 클래스를 통해 폰트 교체 국제화가 가능하기 때문이다. 정적 텍스트는 이러한 특성이 없으므로 기능성에 있어 벡터 아트와 유사하다. 입력 텍스트는 편집 가능한 텍스트 상자가 필요할 때마다 사용될 수 있다.

원으로 표시된 기타 두 개의 부분은 속성 시트 상단에 있는 폰트 이름과 폰트 렌더링 기법이다. 폰트 이름은 (a) 개발자의 시스템에 설치된 폰트 중 하나를 사용하거나 (b) 영화 라이브러리에서 생성되는 폰트 기호 이름 중 하나를 사용해서 선택할 수 있다. 위의 예에선 폰트 이름으로 "Arial"을 선택한다. 또한 Bold 및 Italic 토글 버튼을 통해 폰트 스타일을 설정할 수 있다.

국제화를 위해, 폰트 렌더링 기법은 가장 중요한 설정인데 그 이유는 SWF 파일 내 문자 내장을 제어하기 때문이다. 플래시 8 에서 다음 값 중 하나로 설정이 가능하다.

- 장치 폰트 사용
- 비트맵 텍스트 (안티 앨리어싱 없음)
- 애니메이션용 안티 앨리어싱
- 가독성을 위한 안티 앨리어싱

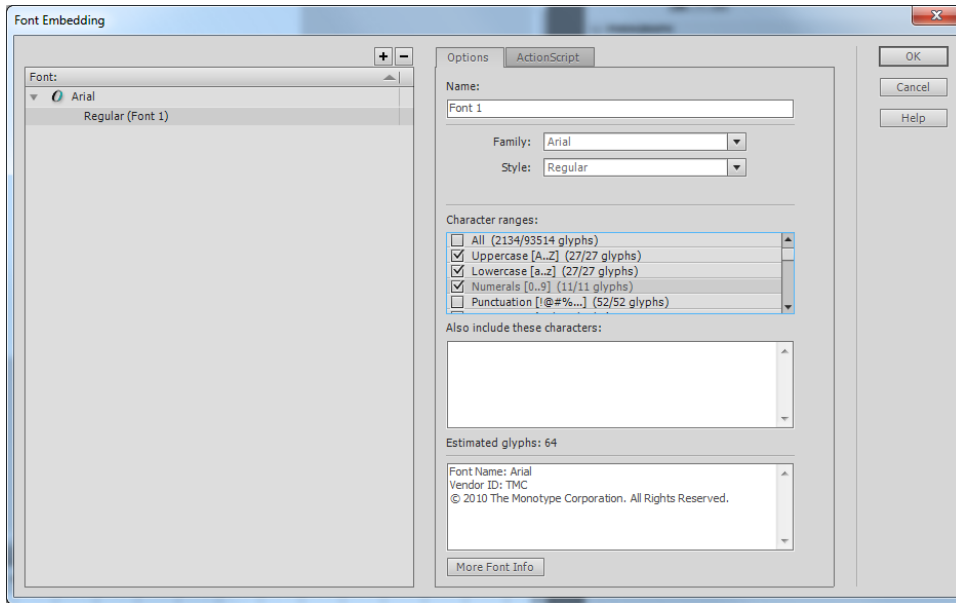
"장치 폰트 사용"을 선택하면, OS 에서 취한 폰트 데이터를 이용하여 시스템 별 폰트 렌더링을 사용한다. 장치 폰트 사용의 장점은 결과로 나오는 SWF 파일의 크기가 작고 재생 시 시스템 메모리를 적게 사용한다는 점이다. 그러나 타깃 시스템에 요청한 폰트가 없는 경우, 즉 플래시 플레이어에 대체 폰트를 선택하는 경우 문제가 발생할 수 있다. 이러한 대체 폰트는 원래의 폰트와 같아 보이지 않을 수 있으며/있거나 많은 경우 필요한 글리프 전체를 갖지 않을 수도 있다. 예를 들어, OS 폰트 라이브러리가 없는 게임 콘솔의 경우, 텍스트가 나타나지 않는다. Scaleform 에션 사용 불가능한 글리프는 사각형으로 렌더링된다.

시스템 폰트를 사용하는 대신, "애니메이션용 안티 앨리어싱"과 "가독성을 위한 안티 앨리어싱"을 설정은 내장된 폰트 문자에 의존하며, 각각 애니메이션 성능 및 높은 최적화로 렌더링한다. 이러한 옵션 중 하나를 선택할 때마다, Embed 버튼이 활성화되며, 이를 통해 사용자는 폰트에 내장된 문자 범위를 선택할 수 있게 된다.



## 1.1.2 문자 내장

임베드된 폰트 렌더링이 기능하려면, 파일에서 하나 이상의 TextField에 지정한 폰트에 대한 캐릭터가 임베드되어야 합니다(폰트를 내보내기 한 경우에는 textfield가 필요하지 않습니다). 만약 필요한 문자가 없을 경우, 장치 폰트를 사용하며 이 때 위에서 언급한 모든 제약 조건이 발생한다 (나중에 언급할 Scaleform에 Gfx::FontLib이 의존하지 않을 경우). 폰트 문자 내장을 위해, 사용자는 Embed 버튼을 눌러 아래와 같이 문자 내장 대화창을 표시할 필요가 있다.



내장 대화창에서, 사용자는 현재 선택한 폰트와 스타일의 텍스트 필드에 대해 내장된 해당 문자 범위를 선택할 수 있다. 동일한 폰트 스타일을 사용하는 모든 텍스트 필드는 동일한 내장 문자셋을 사용하므로 텍스트 필드 중 단 하나에 대한 내장을 지정하는 데 충분하다.

사용자는 임베딩의 관점에서, 폰트의 볼드 및 이탤릭 속성이 개별적으로 다뤄진다는 것을 알아야 합니다(위 스크린샷의 "스타일" 콤보박스를 참고하십시오). 예를 들어 Arial과 Arial Bold 스타일을 사용하는 경우, 독립적으로 내장을 시켜 파일의 메모리 footprint를 증가시켜야 한다. 그러나 폰트 크기를 바꾸는 것은 폰트 사이즈의 증가 또는 메모리 오버헤드 없이 이뤄질 수 있다. 폰트 및 문자 내장에 대한 자세한 내용은 플래시 스튜디오 내 폰트 사용에 관한 내용을 참고한다.

내장 문자는 휴대용 폰트 사용에 있어 유일한 방법이다. 폰트 문자가 내장될 때 폰트의 벡터 표현이 SWF/GFX 파일에 저장되며 차후 사용 시 메모리에 로딩된다. 글리프 데이터가 파일의 일부이기 때문에 사용중인 시스템에 설치된 폰트에 상관 없이 항상 올바르게 렌더링된다. Scaleform에서 내장 폰트는

게임 콘솔 (Xbox 360, PS3, Wii) 및 PC (윈도우, 맥, 리눅스)에서 제대로 작동한다. 내장 폰트 사용에서 있어 피할 수 없는 단점은 파일 크기 및 메모리 사용 증가라는 측면이다. 특히 아시아 언어에 있어 크기 증가가 중요한 요소이기 때문에 개발자들은 미리 게임 폰트 사용에 대한 계획을 수립하고 가능한 경우 항상 폰트를 공유할 필요가 있다.

### 1.1.3 내장 폰트를 통한 메모리 사용

문자 내장이 Scaleform 내에서 갖는 메모리 영향에 관한 사항을 이해하기 위해 내장 문자의 수가 증가할 때의 메모리 사용을 나타내는 다음의 표를 참고한다.

내장 문자 수	비압축 SWF 크기	Scaleform 런타임 크기
1 문자	1 KB	450 K
114 문자 – 라틴 + 기호	12 KB	480 K
596 문자 – 라틴 + 키릴	70 KB	630 K
7,583 문자 – 라틴 및 일본어	2,089 KB	3,500 K
18,437 문자 – 라틴 및 전통 중국어	5,131 KB	8,000 K

위의 표는 “Arial Unicode MS” 폰트를 내장해서 생성된 것이다. 위에서 밝힌 바와 같이, 유럽 문자셋을 포함하는 경우 개발자들은 500 문자당 150K 의 메모리 사용을 포함할 수 있다. 이는 로컬화 된 분포에 있어 서로 다른 폰트 스타일에 대해선 충분하다. 그러나 아시아 언어의 경우 메모리 사용량은 관련 글리프의 수치가 크기 때문에 훨씬 커진다.

### 1.1.4 폰트 메모리 사용 제어

큰 문자 셋을 내장하면 수 메가 바이트의 메모리를 소모할 수 있기 때문에 개발자들은 사전에 폰트 사용에 관한 계획을 수립할 필요가 있으며 그 이유는 메모리 사용량을 줄이기 위한 것이다. 폰트 메모리 제어에 사용될 수 있는 몇 가지 기법이 있다.

1. *사용하는 폰트와 폰트 스타일을 UI 아트 자산 개발 전 미리 정의한 수준으로 제한한다.* 볼드 및 이탤릭 폰트 스타일은 플래시에서 별도의 문자셋으로 내장되므로 전혀 별개의 폰트로 처리하여 필요할 때만 저장해야 한다. 그러나 서로 다른 폰트 크기는 추가적인 오버헤드를 유발하지 않으므로 메모리 증가를 야기하지 않고 사용이 가능하다. 향후, 볼드 및 이탤릭체 용으로 저장해야 하는 추가적인 폰트 문자를 피할 수 있는 가상의 볼드 및 이탤릭 옵션을 제공할 것이다.
2. *GfX::FontLib 과/또는 불러오기를 사용해서 파일간 폰트를 공유한다.* 각 파일에 동일한 문자를 저장하는 것은 자산 크기, 메모리 사용(차후 파일을 동시에 로딩하는 경우) 및 로딩 시간에 있어 비정상적인 증가를 유발할 수 있다. 가능한 경우, 공유되는 별개의 *SWF/GFX* 파일에 내장 폰트를 저장하는 것이 가장 좋으며 이를 통해 메모리에서의 복제 또는 재 로딩이 필요없게 된다. *GfX::FontLib* 의 사용에 관해 이 문서 후반에서 다룰 것이다.
3. *아시아 문자셋에 대해 사용되는 문자만 내장한다.* 아시아에서 로컬화 된 게임에선 해당 언어에 관한 모든 문자의 내장을 피할 수 있으며 대신에 게임 텍스트에서 사용되는 문자만 내장할 수 있다. 번역 후, 모든 게임 스트링을 스캔해서 필요한 특유의 문자셋을 생성하고 그 후 내장한다. IME 를 통해 게임에서 임의의 동적 텍스트 입력을 요구하지 않는 경우, 문자를 제산함으로써 상당한 공간 절약을 도모할 수 있다.
4. *GfXExport 폰트 압축(-fc 옵션) 사용을 고려해 보십시오.* 보통 확인 가능한 품질 감소 없이 10% ~ 30%의 크기 감소를 달성할 수 있습니다(섹션 5.3 참조).

국제화가 잘 된 게임을 보면, 사용자들은 이러한 모든 기법을 결합한 형태를 고르게 되며 GfX::FontLib 을 통해 사용 폰트를 제한하고 공유하게 된다. 아시아 언어에 있어, 사용중인 문자만을 내장하거나 IME 가 필요로 하는 경우, 하나의 완전한 폰트를 내장하게 되면 상당한 메모리 절약을 야기할 수 있다.

## 1.2 게임 UI 용 폰트 결정

Scaleform 에서 게임 사용자 인터페이스를 생성할 때, 개발자들은 폰트 사용, 설정 및 국제화 접근 방식에 대한 다수의 결정을 내릴 필요가 있다. 이 문서에서는 이러한 결정을 네 가지의 독립된 부분으로 다룬다.

- 제 1 부: 게임 폰트 라이브러리 생성 - 폰트 라이브러리 생성 시 몇 개의 폰트 스타일을 사용해야 하는 지를 결정하는 데 도움을 준다.
- 제 2 부: 국제화 접근 선택 - 국제화에 친화적인 게임 자산을 생성하기 위한 여러 접근 방식을 설명한다.
- 제 3 부: 폰트 소스 설정 - 폰트 록 업 순서와 Scaleform 내에서 가용한 폰트 자원을 어떻게 설정할 수 있는 지 설명한다.
- 제 4 부: 폰트 렌더링 설정 - Scaleform 텍스트 렌더링 접근과 이에 대한 상이한 설정을 설명한다.

게임 폰트 라이브러리 생성은 게임 인터페이스가 의존해야 할 폰트 스타일의 선택과 관련이 있으며 이는 게임 UI 자산을 개발하기 전 일반적으로 실시해야 하는 것이다. 예술적 측면에서, 표준 폰트 라이브러리의 사용은 상이한 폰트를 모든 UI, 파일에서 동일한 목적으로 일관되게 사용해야 함을 확인하는 데 있어 중요하다. 기술적 측면에서, 감시 중인 내장 폰트의 수를 유지하여 어플리케이션의 메모리 한계에 맞도록 하는 것이 중요하다.

게임 폰트를 결정한 후라면 어떻게 국제화를 할 것인지를 결정해야 한다. 제 2 부에선 국제화를 위해 사용할 수 있는 세 가지 가능한 모델을 다룬다.

- *불러온 폰트 교체*, Gfx::FontLib 을 통해 공유가 이뤄지고 플레이어에서 로딩된 별도의 언어별 폰트 파일과 불러오기에 의존한다.
- *장치 폰트 에뮬레이션*, 위의 내용과 유사하나 불러온 폰트 대신 장치 폰트에 의존하며 기존 적절한 기준 시스템 폰트 지원을 필요로 한다 (현재 게임 콘솔에선 불가능).
- *커스텀 자산 생성*, 플래시에서 구성된 번역 장치를 사용하여 각 목표 시장에 대한 맞춤형 SWF/GFX 파일을 생성한다.

개발자는 이 중 한가지 또는 몇 가지를 결합하여 개발 과정 전반에서 따를 수 있다.

Scaleform 내 다수의 소스로부터 폰트를 사용할 수 있기 때문에 올바른 설정이 매우 중요하다. 이러한 폰트 설정을 돕기 위해, 제 3 부: 폰트 소스 설정은 Gfx::FontLib 및 폰트 제공자를 어떻게 설정해야 하는가에 대한 예를 포함하여 Scaleform 폰트 록 업 프로세스를 자세히 설명하고 있다. 폰트 제공자는 사용자가 설정 가능한 플래스로서 시스템 폰트와 FreeType2 폰트를 지원하기 위해 사용된다. 아니면 내장 없이 Gfx::FontLib 대신 폰트에 접근하기 위해 사용할 수 있다.

이 문서의 제 4 부인 폰트 렌더링 설정은 Scaleform 에서 사용되는 폰트 렌더링 접근 방식을 설명하며 가용한 폰트 텍스처 국제화의 세 가지 선택에 초점을 둔다. GfxExport 를 통해 전처리간의 렌더링 이전 텍스처, 로딩 시간에 생성된 정적 텍스처 포장 및 요청 시 업데이트 된 동적 텍스처 사용 등이 그 세 가지 선택이다. 이 문서는 각각의 접근 방식에서 가용한 설정 옵션을 논의하며 이를 통해 개발자들은 어플리케이션 및 목표 플랫폼에 대해 올바른 결론을 내릴 수 있게 된다.

## 2 제 1 부: 게임 폰트 라이브러리 생성

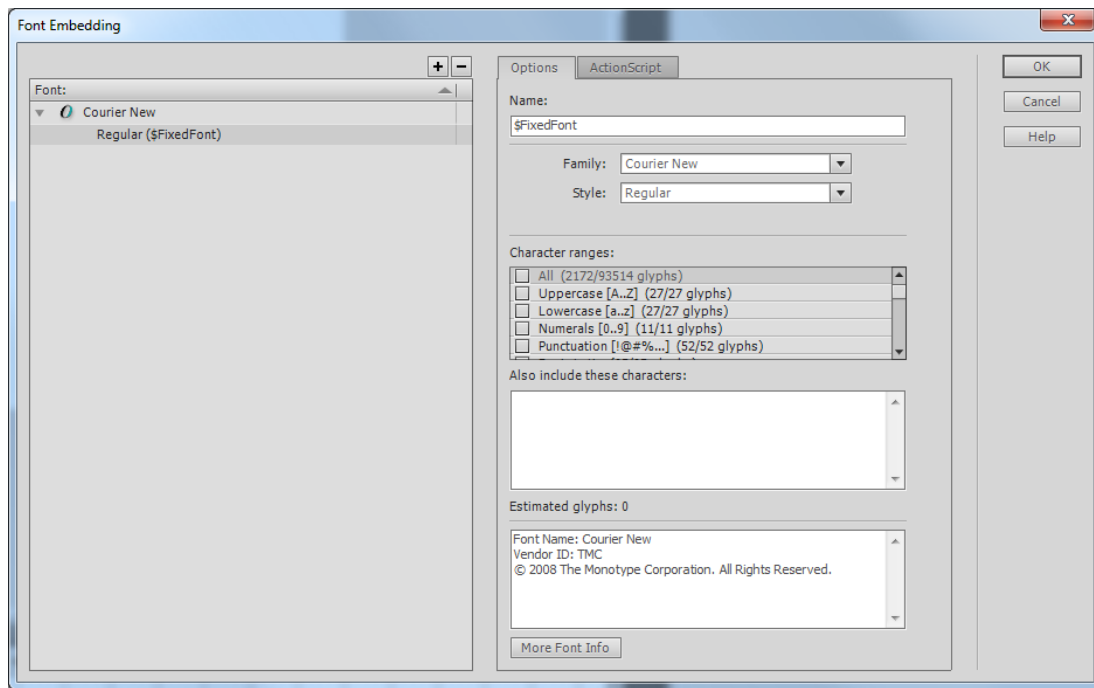
게임 UI에 관한 작업을 시작할 때 아티스트가 처음으로 해야 할 일 중 하나는 게임 전반에 사용될 잠재적인 폰트 집합을 선택하는 것이다. 예를 들어, 모든 제목에 사용할 폰트 형태를 선택하고, 게임 내 나머지 텍스트에서 사용할 다른 폰트를 선택하는 등의 방법이 있다. 이러한 결정을 내린 후, 매우 신중한 고려 없이 게임 UI 화면의 어느 곳에서도 추가적인 폰트 형태를 사용해선 안 된다. 이러한 제한을 두는 세 가지 이유가 있다.

1. 동일한 폰트를 사용함으로써, 아티스트는 게임 내 모든 콘텐츠가 사용자에게 일관적으로 나타남을 확신한다. 만약 다른 화면에서 다른 폰트를 사용하면 읽고 이해하는 데 더 어려운 산만한 UI를 만들게 된다.
2. 국제화 간, 개발 폰트는 종종 목표 언어에 맞는 문자를 갖는 다른 폰트로 교체할 필요가 있다. 폰트를 미리 결정해 놓으면 이러한 작업을 쉽게 할 수 있다.
3. 다수의 UI 화면에서 공유가 이뤄지는 고정된 형태의 폰트 집합을 갖게 되면 데이터를 메모리에서 공유할 수 있고, 이로 인해 메모리 사용 및 화면 로딩 시간을 상당히 절약할 수 있다. 폰트 데이터가 많은 메모리를 점유할 수 있기 때문에, 이는 매우 중요한 기술적인 고려사항이다.

다음 페이지에 기재된, Scaleform으로 가져오기 한 폰트의 대체 접근법을 사용하면 언어당 하나의 파일 세트로 이루어진 폰트 라이브러리를 생성하여 게임을 위한 폰트 세트 선택 작업이 정형화됩니다("fonts\_en.swf", "fonts\_kr.swf" 등). . 아티스트는 플래시 파일 라이브러리 내 명명 폰트 기호를 생성하고 이를 해당 SWF로 보냄으로써 이 파일을 생성할 수 있다. 라이브러리 파일 생성 후, 그 폰트 기호는 다른 플래시 파일에서 불러와서 개발간 사용이 가능하다. 다음 내용은 폰트 기호 생성에 관한 세부사항 및 게임 폰트 라이브러리 생성 방법에 대해 다루고 있다.

### 2.1 폰트 기호

서두에서 우리는 폰트가 텍스트 필드에 어떻게 적용되고 그 문자가 휴대용 재생에 있어 어떻게 내장될 수 있는지를 다뤘다. TextField 속성에서 시스템 폰트의 사용에 더해, 플래시 스튜디오는 아티스트가 라이브러리에 마우스 오른쪽을 클릭한 후 "New Font..." 항목을 선택함으로써 새로운 폰트 기호를 정의할 수 있도록 하며 이 내용은 다음 화면과 같다.



이렇게 생성된 폰트 기호는 FLA 파일 라이브러리에 추가되며 차후 일반적인 시스템 폰트와 유사한 텍스트 필드에 적용될 수 있다. 예를 들어, 게임 폰트를 "\$FixedFont"라고 명명하면 귀하의 TextField에서 유효한 폰트로서 해당 명칭을 선택할 수 있어야 한다.

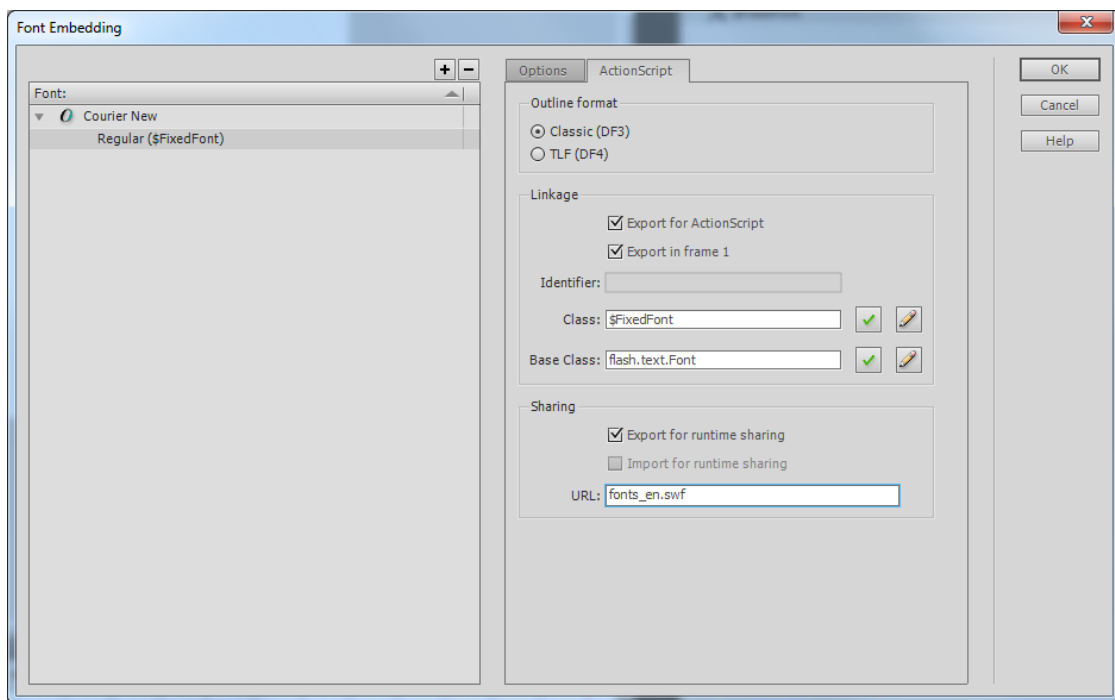
폰트 이름을 라이브러리에 추가하게 되면 고유의 방법으로 사용된 폰트를 확인한다는 장점이 있다. 라이브러리 폰트가 폰트 목록에 표시될 때, "\$ArialGame\*"과 같이 \*가 옆에 나타난다. 아티스트가 라이브러리에 있는 파이 이름만 사용하는 경우 실수로 UI 파일에서 기타 폰트를 참조하지 않음을 확인할 수 있게 된다. 이에 더해, 만약 모든 폰트의 기호가 '\$' 기호로 시작하는 이름을 갖는다면, TextField 속성 내 폰트 목록에 항상 나타나며 개발에 도움을 준다.

다수의 웹사이트에선 폰트 기호 이름에 대한 접두사로서 '\_' 기호를 사용하도록 권장하고 있다. 밑줄 기호가 분명한 것처럼 보이지만, 이를 사용하게 되면 "Font rendering method" 선택 상자가 깨지게 되는데, 그 이유는 플래시 스튜디오가 '\_'로 시작하는 모든 폰트명을 빌트인 폰트로 인식하기 때문이다. 이러한 문제를 피하기 위해 위의 예에선 '\$' 표시를 대신 사용했다.

### 2.1.1 폰트 기호의 불러오기 및 내보내기

기타 라이브러리 입력과 마찬가지로, FLA 파일 내에 위치한 폰트 기호는 그 내용을 복사하거나 불러오기/내보내기 메커니즘에 따라 기타 파일에서 사용이 가능하다. 라이브러리 기호를 복사하기 위해, 소스 FLA 라이브러리에 오른쪽 마우스를 클릭하고 “Copy”를 선택한다. 그 후, 목표 FLA 파일 라이브러리로 이동한 후 오른쪽 마우스를 클릭하여 “Paste”를 선택한다. 기호의 복사본이 생성되고 복사된 모든 데이터가 두 번째 파일에 담기게 된다.

데이터 중복을 피하기 위해, 내보내기/불러오기 메커니즘을 플래시에서 사용할 수 있다. 라이브러리 심볼을 내보내기 하려면, 오른쪽 마우스 버튼으로 클릭하여 “속성”을 선택하고 “ActionScript” 탭을 선택하십시오. 다음 속성 시트가 표시됩니다.



“OK” 버튼을 누르면 Flash 에서 클래스에 대한 정의가 없다는 경고 메시지를 표시하지만, 이 경고 메시지는 무시해도 안전합니다.

심볼을 내보내기 할 때에는 내보내기 식별자를 부여하게 되며, 여기에 “런타임 공유를 위해 내보내기”, “ActionScript 를 위해 내보내기” 및 “첫 번째 프레임으로 내보내기” 플래그를 붙이게 됩니다.

기호 이름과 동일한 불러오기 확인자를 설정하는 게 좋다. 이 심볼을 가져오기 할 때에는 URL 로 사용할 SWF 파일 경로를 지정해야 합니다; 폰트 심볼을 Gfx::FontLib 오브젝트를 통해 대체하려 할 때는 URL 필드를 사용자의 기본 폰트 라이브러리로 설정해야 합니다. 본 문서에서는 기본 이름으로 “fonts\_en.swf”를 사용하지만, 임의의 다른 이름도 사용할 수 있습니다. **폰트 대체를 사용하려면 Gfx 에서 기본 폰트 라이브러리를 설정해야 함을 명심하십시오.**



## 예제

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

fonconfig.txt 에서 GfxPlayer 사용자에게 대한 기본 폰트 라이브러리를 설정할 수 있습니다.

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

첫 번째 fontlib 모양을 기본 라이브러리로 사용하게 됩니다.

기호를 내보내면, 라이브러리 Linkage 컬럼에 "Export: \$identifier" 라벨이 붙게 된다. 만약 Copy/Paste 또는 드래그 및 드롭 옵션이 불러온 라이브러리 기호에 적용되면 더 이상은 복사본을 만들지 않으며 오히려 목표 파일에 불러오기 링크를 생성한다. 이는 목표 파일의 크기가 더 작으며 메모리에 로딩될 때 소스 SWF 에서 불러온 데이터를 끌어온다는 것을 의미한다. Scaleform 에서 불러온 SWF 데이터는 여러 파일로부터 불러올 때 한 번만 로딩되며 이는 잠재적으로 시스템 메모리 사용량을 크게 절약한다.

폰트 기호명은 TextField.htmlText 문자열 또는 TextFormat 내의 폰트명의 일부로 반환되지 않는다.

"Arial"과 같은 원래 시스템의 폰트명을 대신 반환한다. 이와 유사하게, 기호명은 내보낸 폰트에 대해 불러온 이름이 일치하지 않으면 ActionScript 를 통해 할당될 수 없다 (권장하지 않음).

### 2.1.2 내보낸 폰트 및 문자 내장

버전 CS4 까지는 Flash Studio 로 내보내기 한 폰트에 대해 임베드할 캐릭터를 지정할 수 없었습니다. 대신 임베드할 캐릭터 세트를 시스템 로컬 설정으로 결정하였습니다. 윈도우에서 이는 제어판<sup>W</sup>지역 및 언어 옵션<sup>W</sup>고급 설정에 있는 "비 유니코드 시스템용 언어"를 통해 제어된다. 만약 언어를 "영어"로 설정한다면 243 개의 글리프만이 모든 폰트를 위해 사용된다. "한국어"를 선택한 경우 11,920 개의 문자를 내보낸다. 이러한 방법은 게임 개발에 편리하지 않으므로, 'gfxfontlib.swf'를 이용한 전체적 접근법을 만들었습니다.

다행히 Flash CS5 부터는 내보내기 할 문자를 지정할 수 있게 되어 'gfxfontlib.swf'은 더 이상 필요하지 않습니다. 대신 개발자가 언어별 특정 폰트 라이브러리(예컨대 'fonts\_en.swf', 'fonts\_jp.swf' 등)를 생성하고 어느 문자를 각 라이브러리에 임베드할지를 직접 지정할 수 있습니다.

그러나 CS4 Flash Studio(또는 이전 버전)를 사용하는 경우에는 'gfxfontlib.swf' 접근법을 사용해야 합니다(현재 버전에서 권장하지는 않지만, 여전히 지원함). 이 접근법에 대한 보다 자세한 내용은 아래 및 Gfx 4.0 에 대한 본 문서의 이전 개정판을 참조하십시오.

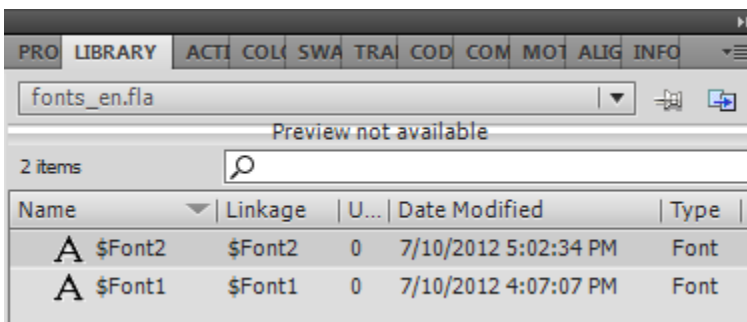
## 2.2 gfontlib.swf 생성 단계

폰트 심볼을 제대로 이해했다면, 이제 폰트 라이브러리 파일을 생성할 준비가 되었습니다. 재반복을 위해 이들 파일을 생성하여 언어별 소스 폰트를 제공합니다. 라이브러리 폰트를 매핑하고 대체하는 방법에 대한 자세한 내용은 다음 섹션에 기재됩니다.

특정 언어에 대한 폰트 라이브러리 파일을 생성하려면, Flash Studio 에서 새 FLA 를 생성하고 그 라이브러리에 폰트 심볼을 배치합니다. . 특히, 다음과 같은 단계를 거칠 수 있다.

1. 모든 게임 장면에서 사용할 폰트를 정의하고 그 목적에 따른 명칭을 부여한다. '\$TitleFont'와 '\$NormalFont'는 각각 게임 타이틀 및 일반 크기의 폰트를 사용하는 데 있어 좋은 이름이 될 수 있다.
2. 라이브러리로 새로운 FLA 파일을 생성한다.
3. 1 단계에서 정의한 모든 폰트에 대해 새로운 폰트 기호를 생성한다.
4. 불러온 폰트 교체를 사용하려는 경우, 각 폰트에 대한 연결 속성을 설정하여 폰트 기호명과 동일한 확인자로 내보내도록 한다.
5. 어느 문자를 임베딩할 것인지를 지정합니다.
6. 생성 파일을 fonts\_<lang>.swf 로 저장하십시오(여기서 'lang'은 언어, 예를 들어 'en' – 영어, 'ru' – 러시아어, 'jp' – 일본어, 'cn' – 중국어, 'kr' – 한국어 등을 가리킴). 어느 이름이든 사용할 수 있지만, 이 이름을 나중에 FontMap 구성에 사용해야 합니다.

설명 목적으로 추가 텍스트 필드를 사용하려는 것이 아니라면, 폰트 라이브러리 단계에 어떠한 textfield 도 추가할 필요가 없습니다. . 폰트를 제외하고 기타 기호 형태를 폰트 라이브러리 파일에 추가해선 안 된다. FLA 를 완료한 후, 라이브러리는 다음과 유사한 형태로 나타나야 한다.



폰트 라이브러리 완료와 함께, 그 기호는 어플리케이션의 사용자 인터페이스 화면에서의 사용이 준비된다. 불러오는 파일에서 라이브러리 폰트를 사용하기 위해, 폰트 기호를 타깃 영화 파일에 드롭하거나 앞에서 언급한 Copy/Paste 기법을 사용하면 된다.

## 3 제 2 부: 국제화 접근 선택

UI 국제화 단계에서 교체할 필요가 있는 두 가지 주요 항목은 텍스트 필드 문자열과 폰트이다. 텍스트 필드 문자열은 언어 변환을 위해서 교체되며 목표 언어에서의 대응 문장으로 개발 텍스트를 교체한다. 원래 개발 폰트가 필요한 모든 문자를 포함하지 않기 때문에 목표 언어에 대한 올바른 문자셋 제공을 위해서 폰트를 교체한다.

이 문서에서는 예술 자산의 국제화를 위해 사용할 수 있는 세 가지 다른 접근 방식을 설명한다.

1. *불러온 폰트 교체.*
2. *장치 폰트 에뮬레이션.*
3. *커스텀 자산 생성.*

불러온 폰트 교체는 폰트 국제화를 위한 권장 사항이며 default 'fonts\_en.swf' 라이브러리로 제공되는 폰트 기호 교체를 위해 Gfx::FontLib 과 Gfx::FontMap 객체에 따른다.

장치 폰트 에뮬레이션은 불러온 폰트 교체와 유사하나 불러온 기호 대신에 실제 폰트를 제공하기 위한 폰트 매핑에 의존하며 이는 설정에선 약간 더 쉬우나 몇 가지 제약사항이 있다.

불러온 폰트 교체와 장치 폰트 에뮬레이션 모두 텍스트 문자열 번역을 위해 사용자가 생성한 Gfx::Translator 객체에 의존한다.

커스텀 자산 생성은 폰트 매핑이나 번역 객체를 사용하지 않는다는 점에서 위의 두 접근 방식과는 다르며 대신에 각 목표 언어용 UI 자산 파일을 생성하는 플래시 스튜디오의 특성에 의존한다. 이러한 접근은 소수의 정적 UI 자산이 있는, 메모리가 극도로 제한된 플랫폼에서 선택이 가능하다.

### 3.1 불러온 폰트 교체

불러온 폰트 교체는 결합 텍스트 필드가 폰트를 대체하는 플래시 폰트 기호의 불러오기/내보내기에 의존한다. 본 방법을 사용하려면, 먼저 기본 폰트 라이브러리 파일(예를 들어 파트 1 에 기재된 대로 "fonts\_en.swf")을 생성한 뒤 전체 게임 UI 파일에서 해당 파일에서 내보내기 한 폰트 심볼을 사용합니다.

. UI 파일을 어도비 플래시 플레이어에서 테스트할 때, "fonts\_en.swf" 파일에서 불러오며, 이는 해당 콘텐츠가 개발 언어에서 올바르게 렌더링 될 수 있도록 한다. 그러나 이러한 자산을 Scaleform 에서 실행할 때, 국제화 구성을 대신 로딩할 수 있으며 이를 통해 번역 및 폰트 대체가 가능하다.

*Loader::SetDefaultFontLibName (const char\* filename)* C++ 방법을 호출하여 'filename' 파라미터로 파일명(경로가 아닌 파일명만!)을 전달해야 함을 명심하십시오. 예를 들어, 기본 개발 언어가 한국어라면 다음과 같이 호출해야 합니다.

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

이 방법은 메인 SWF 파일에 대한 CreateInstance 를 호출하기 전에 호출해야만 합니다.

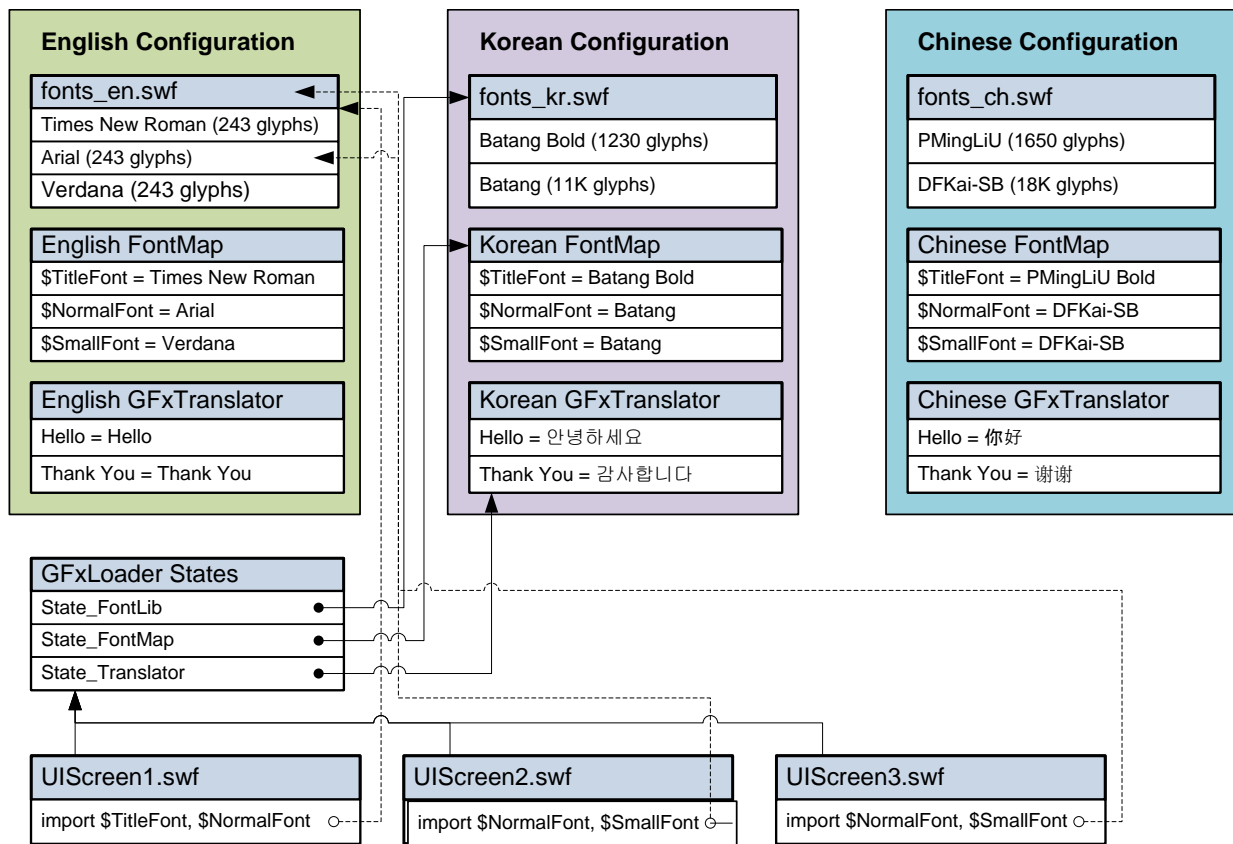
근본적으로, 게임의 국제화 설정은 다음과 같은 데이터 셋으로 구성된다.

게임 내의 문장을 번역된 문장과 매핑시키는 스트링 번역 테이블

번역에 사용되는 문자용 글리프를 제공하는 폰트 집합

설정에서 사용 가능한 "\$NormalFont"와 같이 폰트 기호명을 매핑시키는 폰트 대체표

Scaleform 에서 이러한 컴포넌트는 Gfx::Loader 에서 설치 가능한 Gfx::Translator, Gfx::FontLib 및 Gfx::FontMap 상태 객체로 나타난다. Gfx::Loader::CreateMovie 로의 호출과 함께 영화를 로딩할 때, 로더가 제공하는 폰트 설정 상태를 따르며 텍스트 렌더링을 위해 해당 상태에 있는 폰트에 의존한다. 서로 다른 폰트 결합을 생성하기 위해, 사용자는 로더에 있는 새로운 상태 결합을 사용하고 CreateMovie 를 다시 호출할 필요가 있다.



위의 그림은 잠재적인 국제화 폰트 설정에 관한 내용을 나타낸다. 이 설정에서, "UIScreen1.swf"부터 "UIScreen3.swf"라는 세 개의 사용자 인터페이스 파일은 "fonts\_en.swf" 라이브러리에서 폰트 집합을 불러온다. 불러온 파일 링크를 점선 화살표로 나타냈다. 그러나 불러온 파일에서 폰트를 취하는 대신 Scaleform 는 Gfx::Loader 에서 설정된 Gfx::FontLib 과 Gfx::Fontmap 상태를 기준으로 폰트를 교체할 수 있다. (1) 가져오기 한 파일이 Loader::SetDefaultFontLibName() 방법으로 지정한 것과 일치하고, (2) UI 화면을 불러오기 전에 null 이 아닌 Gfx::FontLib 상태가 loader 에 설정되었다면, 자동으로 대체되게 됩니다.

국제화 번역을 제공하기 위해, 위에선 영어, 한국어 및 중국어의 세 가지 설정을 사용한다. 각각의 설정은 한국어의 경우 "fonts\_kr.swf"와 같은 자신의 폰트 파일과 불러온 폰트 기호명과 폰트 파일에 내장된 실제 폰트를 매칭시키는 폰트 맵을 포함한다. 여기에서, 로더 상태는 한국어로 설정되어 있으며 이는 한국어 폰트와 번역 테이블을 사용하겠다는 것을 의미한다. 그러나 로더 상태는 영어나 중국어 설정으로 쉽게 설정할 수 있으며 이 경우 사용자 인터페이스는 해당 언어로 번역을 실시하게 된다. 이러한 번역을 위해 UIScreen 파일을 수정할 필요는 없다.

### 3.1.1 국제 텍스트 설정

불러온 폰트 교체로 게임을 국제화 할 때, 개발자들은 우선 국제화 예술 자산과 필요한 언어 설정을 생성할 필요가 있다. 예술 및 폰트 자산은 일반적으로 SWF 포맷 (탐재 전 GFX 로 변환 가능)으로 저장되며 폰트 맵과 번역 테이블은 GfX::FontMap 과 GfX::Translator 객체의 생성을 지원하는 게임별 데이터에 저장될 수 있다. Scaleform 샘플에서, 이러한 정보 제공을 위해 "fontconfig.txt" 설정 파일에 따른다. 그러나 게임 개발자들은 생산을 위해 더욱 발전된 계획을 수립해야 한다.

공식적으로 국제화 된 설정의 생성은 다음과 같은 단계로 이뤄질 수 있다.

1. 파트 1 에 기재된 바와 같이 기본 라이브러리 파일(예를 들면 "fonts\_en.swf")을 생성하고 이를 사용자의 게임 자산 생성에 사용합니다.
2. 해당 GfX::Translator 객체 생성을 위해 사용하는 각 언어에 관한 번역 문자열 표를 생성한다.
3. 해당 언어에 어떤 폰트 매칭을 사용할 지 결정한다. GfX::FontMap 생성에 사용할 수 있는 포맷으로 이 맵을 저장한다.
4. 각각의 대상 언어에 대해 "fonts\_kr.swf"와 같이 언어별 특정 폰트 파일을 생성합니다.

국제화 설정을 생성한 후, 개발자들은 게임에 국제화 된 UI 화면을 로딩할 수 있다. 목표 언어를 선택했으면, 다음과 같은 단계를 밟을 필요가 있다.

1. 해당 언어에 관한 번역 객체를 생성하고 GfX::Loader 에 설정한다. 귀하의 클래스를 GfX::Translator 로부터 유도하고 Translate 가상 함수를 override 함으로써 번역을 구현한다. 이 클래스를 override 함으로써, 개발자들은 자신이 선택한 어떠한 포맷으로 번역 데이터를 나타낼 수 있다.
2. GfX::FontMap 객체를 생성하고 GfX::Loader 에 설정한다. MapFont 함수를 호출하여 필요한 폰트 매핑을 추가한다. GfX::FontMap 의 사용 예가 제 3 부에 있다.
3. GfX::FontProvider 및/또는 GfX::FontPackParams 와 같이 게임에 필요한 다른 모든 폰트 관련 상태를 설정합니다. 필요하다면 렌더 스레드에서 GlyphCacheConfig 를 통해 동적 캐시를 구성하십시오.
4. GfX::FontLib 객체를 생성하고 GfX::Loader 에 설정한다.
5. 기본 폰트 라이브러리(swf 콘텐츠에 사용한 것)를 위해 SetDefaultFontLibName(파일명) 방법을 호출합니다.

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

GFx::FontLib 을 생성한 후, 목표 언어에서 사용되는 폰트 소스 SWF/GFX 파일에 로딩하고 GFx::FontLib::AddFromFile 을 호출함으로써 라이브러리에 추가한다. 이러한 호출은 폰트 불러오기 또는 장치 폰트로서 설명 파일에서 사용이 가능하도록 폰트를 내장시킨다.

6. GFx::Loader::CreateMovie 를 호출하여 사용자 인터페이스 파일을 로딩한다. 폰트맵인 fontlib 과 번역 상태가 자동으로 사용자 인터페이스에 적용된다.

### 3.1.2 Scaleform Player 에서의 국제화

국제화를 나타내기 위해, Scaleform Player 는 국제화 프로파일 파일인 "fontconfig.txt"의 사용을 지원한다. SWF/GFX 파일을 드래그하여 플레이어에 놓으면, 가능한 경우 'fontconfig.txt' 설정이 파일의 로컬 디렉토리에서 자동으로 로딩된다. 또한 명령줄에서 /fc 옵션을 사용해서 로딩이 가능하다. 국제화 프로파일을 로딩하면, 사용자는 CTRL + N 키를 사용해서 언어 설정을 변경할 수 있다. F2 키를 누르면 현재 설정이 플레이어 HUD 의 하단에 나타난다.

국제화 프로파일은 8 비트 ASCII 또는 UTF-16 포맷 중 하나로 저장 가능하며 폰트 설정 및 그 속성의 나열을 통해 생성된 선형 구조를 갖는다. 다음의 국제화 프로파일을 사용해서 앞에서 설명한 게임 설정을 설명할 수 있다.

```
[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"
```

샘플에서 보는 바와 같이, [FontConfig "name"] 헤더로 시작하는 세 가지 설정 항목이 있다. 해당 설정을 선택한 경우 설정 명칭이 Scaleform Player HUD 에 나타난다. 각 설정 내에서, 설정에 적용된 설명을 사용한다. 가능한 내용이 다음의 표에 나와 있다.

설정 설명	의미
fontlib "fontfile"	특정 SWF/GFX 폰트 파일을 Gfx::FontLib 에 로딩한다. 첫 번째 fontlib 모양을 기본 폰트 라이브러리로 사용하게 됩니다.
map "\$UIFont" = "PMingLiU"	게임 UI 화면에서 사용하는 폰트를 폰트 라이브러리가 제공하는 목표 폰트로 매핑하는 Gfx::FontMap 에 항목을 추가한다. 불러온 폰트 교체와 함께, \$UIFont 는 "gfxfontlib.swf"에서 UI 파일로 불러온 폰트 기호의 명칭이어야 한다.
tr "Hello" = "□好"	소스 문자열에서 목표 언어와 동일한 번역문을 추가한다. 개발자는 문자열 번역 테이블의 사용을 위해 더욱 향상된 솔루션을 사용해야 한다.

사용자는 설정 파일의 기능을 Scaleform 를 사용해 국제화를 어떻게 이룰 수 있는가에 대한 샘플로서 제공한다는 점을 주지해야 한다. 설정 파일의 구조는 향후 통지 없이 변경될 수 있다. 그러나 Scaleform 국제화 기능이 발전함에 따라 이를 XML 로 변환할 계획이다. 개발자들은 폰트 상태를 어떻게 설정할 수 있는가에 대한 예로 FontConfigParser.h/cpp 에 있는 소스 코드를 참고할 수 있다.

더욱 완성된 불러온 폰트 교체 샘플이 Bin\Data\AS2\Samples\FontConfig and Bin\Data\AS3\Samples\FontConfig 디렉토리 내 Scaleform 2.x 에 탑재되어 있다. 해당 디렉토리는 Scaleform Player 로 드롭 가능한 "sample.swf" 파일과 다양한 언어로 번역을 실시하는 "fontconfig.txt" 파일을 포함한다. 개발자들은 해당 디렉토리에 있는 파일을 사용하여 국제화 과정을 더욱 잘 이해할 수 있다.

### 3.1.3 장치 폰트 에뮬레이션

장치 폰트 에뮬레이션은 불러온 폰트 기호를 사용하지 않는 대신 시스템 폰트를 교체하기 위해 Gfx::FontMap 을 사용한다는 점에서 불러온 폰트 교체와는 다르다. UI 생성간, 아티스트는 "Arial" 및 "Verdana"와 같은 개발 시스템 폰트를 선택하며 모든 UI 자산에 관한 장치 폰트로서 직접 사용하게

된다. Scaleform Player 에서 구동 시, 폰트 설정을 사용하여 시스템 폰트명을 목표 언어 내 대체 폰트로 매핑할 수 있다. 이러한 매핑의 예로서, "Arial"은 한국어 사용자 인터페이스에서 "바탕"체로 매핑이 가능하다. 결국 "바탕"체는 Gfx::FontLib 의 사용을 통해 "font\_kr.swf"에서 로딩이 가능하다.

장치 폰트 에뮬레이션은 개발자가 Gfx::FontLib 에 따르지 않고 Gfx::FontProviderWin32 를 통해 시스템 폰트를 직접 사용하거나 Gfx::FontProviderFT2 를 통해 폰트 파일을 직접 사용하는 경우 유용하다. 이러한 접근 방식이 설정상 간단히 보일 수 있겠지만 장치 폰트는 불러온 폰트 교체에 비해 다수의 제한사항이 있기 때문에 유연성에서 떨어진다.

- 장치 폰트는 플래시에서 변환이 불가능하다. 어도비 플래시 플레이어는 회전 변환이 적용되는 경우 장치 폰트 텍스트 필드를 표시할 수 없게 된다. 이에 더해, 플레이어는 장치 폰트 텍스트의 크기를 제대로 표시할 수 없고 여기에 적용한 마스크를 무시한다. 이러한 모든 특성이 Scaleform 에션 제대로 작동하더라도 플래시의 제한된 지원을 UI 예술 자산의 테스트를 더 어렵게 만든다.
- 장치 폰트는 "애니메이션용 안티 앨리어싱" 설정을 하도록 가상으로 설정이 불가능하다. Scaleform Player 는 ActionScript 내 TextField.antiAliasType 속성을 통해 정의되지 않으면 항상 가독성을 위해 장치 폰트 텍스트를 안티 앨리어싱한다.
- 장치 폰트를 사용하는 경우 아티스트가 모든 UI 자산의 재편집을 하지 않고 개발 폰트를 수정하는 것이 불가능하다. 폰트 맵의 사용은 Scaleform 측면에서 가능하나 어도비 플래시 플레이어에서 테스트하는 경우 이를 적용하지 않는다. 가져오기 한 폰트 대체를 대신 사용할 때는, 기본 폰트 lib 파일("fonts\_en.swf")을 단순히 편집 및 재생성하면 다른 개발 폰트를 선택할 수 있습니다.
- ActionScript 에서 TextFormat 으로 작업할 때 기호명을 불러오지 않으면 해당 기호명을 인식하지 못한다. 이는 개발자들이 "\$TitleFont"와 같은 기호명 대신 "Arial"과 같은 폰트명을 직접 사용해야 한다는 것을 의미한다.

장치 폰트 에뮬레이션과 함께 "fonts\_en.swf" 파일이 꼭 필요하진 않지만 개발간 폰트 기호 저장공간으로서 여전히 유용하다. 아티스트가 "fonts\_en.swf"를 사용하는 경우 폰트 기호를 내보내선 안 된다. 대신 이러한 기호를 목표 UI 파일로 복사하여 매핑이 일어난 장치 폰트에서 앨리어스의 역할을 하도록 할 수 있다.

### 3.1.4 폰트 라이브러리를 생성하기 위한 단계별 가이드

여기에서는 2 가지 언어 및 fontconfig.txt 를 갖는 폰트 라이브러리를 사용하는, 단순 SWF 를 생성하는 단계를 설명합니다.

1. 새 FLA 를 생성하고, ActionScript 2.0 또는 3.0 을 선택합니다. 이것을 'main-app.fl'a라고 부릅니다. 이것이 폰트 라이브러리를 사용하는 메인 응용 프로그램 SWF 가 됩니다.
2. 다른 FLA 를 생성하고, 이전 단계에서 선택한 것과 동일한 ActionScript 를 선택합니다. 이것을 'fonts\_en.swf'라고 부릅니다. 이것이 기본 폰트 라이브러리가 됩니다.
3. '라이브러리' 윈도우로 이동하고, 오른쪽 마우스 버튼을 클릭하여 '새 폰트...'를 선택하고 이름을 \$Font1 로 지정하고, "Family"를 "Arial"로, "Style"을 "Regular"로 선택합니다.
4. "캐릭터 범위"에서 임베드하려는 캐릭터를 선택합니다. 영어라면 'Basic Latin'을 선택하면 됩니다.
5. 'ActionScript' 탭으로 전환합니다. 'ActionScript 를 위해 내보내기', '프레임 1 을 위해 내보내기', '런타임 공유를 위해 내보내기'를 체크합니다. 'URL' 입력 필드에 'fonts\_en.swf'를 타이핑합니다. 'OK' 버튼을 클릭합니다. ActionScript 3.0 을 사용할 때는 Flash Studio 에서 '이 클래스에 대한 정의를 찾을 수 없습니다...'라는 경고를 표시할 것입니다. 이를 무시하고 다시 'OK'를 클릭하십시오.
6. 다른 폰트에 대해서는 단계 3 ~ 5 를 반복하고, 이를 '\$Font2'로 명명하여 다른 'Family' 및/또는 'Style'을 선택하십시오. 'Family'와 'Style'이 모두 동일한 경우, Flash Studio 는 첫 번째 것의 복제본인 두 번째 폰트를 폐기할 수 있습니다. 이번에는 'Times New Roman'을 선택했다고 합시다.
7. 하나 이상의 FLA 를 동일한 ActionScript 설정으로 생성합니다. 사용하려는 언어에 따라 이것을 'fonts\_kr.fl'a 또는 'fonts\_ru.fl'a 또는 임의의 다른 이름으로 부릅니다. 한국어라고 치면, 'fonts\_kr.fl'a가 됩니다.
8. '라이브러리' 윈도우로 이동하고, 오른쪽 마우스 버튼을 클릭하여 '새 폰트...'를 선택하고 이름을 '\$Font1'로 지정하지만, 이번에는 "Family"를 다른 한국어 폰트, 예를 들어 "바탕"으로, "Style"을 "Regular"로 선택합니다.
9. "캐릭터 범위"에서 임베드하려는 캐릭터를 선택합니다. 한국어라면, '한국어 한글(모두)'일 수 있습니다.

10. 'ActionScript' 탭으로 전환합니다. 'ActionScript 를 위해 내보내기', '프레임 1 을 위해 내보내기', '런타임 공유를 위해 내보내기'를 체크합니다. 'URL' 입력 필드에 'fonts\_kr.swf'를 타이핑합니다. 'OK' 버튼을 클릭합니다. ActionScript 3.0 을 사용할 때는 Flash Studio 에서 '이 클래스에 대한 정의를 찾을 수 없습니다...'라는 경고를 표시할 것입니다. 이를 무시하고 다시 'OK'를 클릭하십시오.
11. 다른 폰트에 대해서는 단계 8 ~ 10 을 반복하고, 이를 '\$Font2'로 명명하여 'Family'를, 예를 들어 '바탕체'로 선택하십시오.
12. 'fonts\_en.swf' 및 'fonts\_kr.swf'를 모두 배포합니다.
13. 이제 main-app fla 로 돌아올 시간입니다. 그러나 먼저 기본 폰트 라이브러리 파일(즉 'fonts\_en.swf')로 이동하여, 라이브러리로 이동하고, 거기에서 '\$Font1' 및 '\$Font2'를 모두 선택하여 이들을 클립보드로 복사합니다(Ctrl-C 나 오른쪽 마우스를 클릭 -> '복사하기').
14. 'main-app fla'로 전환하여, 라이브러리로 이동하고 폰트 심볼을 붙입니다(Ctrl-V 나 오른쪽 마우스를 클릭 -> '붙이기'). 폰트에 '가져오기:' 접두어가 붙은 것이 보이게 됩니다.
15. 'main-app fla' 단계에서 텍스트 필드를 생성합니다. 이것을 'Classic Text', 'Dynamic Text'로 만듭니다. 드롭다운 'Family' 목록에서 '\$Font1\*'을 선택합니다. 텍스트 필드의 콘텐츠로 '\$TEXT1'을 타이핑합니다(이것은 번역을 위한 ID 로 사용됨).
16. 두 번째 텍스트 필드를 생성하고, '\$Font2\*'를 선택하여, 여기에 '\$TEXT2'를 타이핑합니다.
17. 'main-app.swf'를 배포하면 끝납니다.
18. 이제 전체 swf 를 저장한 디렉토리에 fontconfig.txt 를 생성해야 합니다. Unicode 또는 UTF-8 로 작업할 수 있는 '메모장' 또는 다른 텍스트 에디터를 열어서 다음과 같이 타이핑합니다.

```
[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 "$TEXT1" = "이것은"
tr "$TEXT2" = "한국어이다!"
```

**중요:** 폰트에 한국어 이름을 사용하였지만("바탕" 및 "바탕체"), Flash 가 SWF 에서 영어 이름을 사용할 수 있습니다("Batang" 및 "BatangChe"). 이것이 fontconfig.txt 에서 영어 폰트 이름을 사용하는 이유입니다. 어느 폰트 이름이 SWF 에서 생성되었는지를 확인하려면 *gfxexport -fntlst <swfname>* 명령을 사용하여 출력 \*.lst 파일을 분석할 수 있습니다.

19. 파일을 저장합니다(반드시 Unicode 또는 UTF-8 로 저장해야 함!). 이제 끝났습니다!

GFxPlayer 에서 'main-app.swf'를 열면 'This is' 및 'English!'를 기본 텍스트로 보게 됩니다. Ctrl-N 을 사용하여 한국어로 전환하면 영어 텍스트가 어떻게 fonts\_kr.swf 의 폰트를 사용하여 한국어로 치환되는지를 보게 됩니다(한국어 문자를 fonts\_kr.swf 에만 임베드했으므로).

## 3.2 커스텀 자산 생성

그 명칭에서 알 수 있듯, 커스텀 자산 생성은 각 목표 언어 배포에 있어 커스텀 SWF/GFX 파일의 생성에 의존한다. 예를 들어 어떤 게임이 "UIScreen.fla" 파일을 사용하는 경우 SWF 의 서로 다른 버전을 게임 배포를 위해 영어로 된 "UIScreen.swf"와 다른 한국어 "UIScreen.swf"로 수동으로 생성할 수 있다. 서로 다른 파일 버전을 서로 다른 파일 시스템 디렉토리에 넣거나 목표 시장에 아예 배포하지 않을 수 있다.

커스텀 자산의 주요 이점은 각 파일에 내장할 글리프의 수를 최소한으로 맞춘다는 것이다. 우리가 일반적으로 이러한 접근 방식을 권장하진 않지만 커스텀 자산 생성은 메모리가 극도로 제한적이고 (UI 에 대해 600K 이하의 메모리가 가용한 경우) 예술 자산이 다음의 기준을 충족하는 경우 고려될 수 있다.

UI 자산 파일의 수가 상대적으로 작고 텍스트 콘텐츠가 단순한 경우

다수의 UI 파일이 같이 로딩되는 경우가 거의 없을 때 (그런 경우, 기타 접근방식에서 제공하는 폰트 공유 특성을 통해 이익을 얻을 수 있다)

동적 텍스트 필드 업데이트가 제한되고 소수의 내장 문자를 사용할 때  
UI 는 아시아 언어 IME 지원을 필요로 하지 않는다.

만약 개발자가 게임 타이틀에서의 사용을 위해 커스텀 자산 생성을 선택하는 경우, 플래시 문서에서 "문자열 패널이 있는 다중 언어 텍스트 제작"이라는 부분을 참고하여 국제화 텍스트를 위해 문자열 패널 (CTRL + F11)을 사용할 것을 권장한다. 아티스트는 현재 Scaleform 버전에서 플래시 텍스트 대체를 제대로 작동시키기 위해 "Replace string"을 "스테이지 언어를 수동으로 사용"으로 설정할 필요가 있다.

## 4 제 3 부: 폰트 소스 설정

플래시의 각 텍스트 필드는 직접 저장되거나 HTML 태그로 인코딩되는 방법으로 폰트명을 갖는다.

텍스트 필드를 표시할 때 렌더링에 사용되는 폰트는 다음과 같은 폰트 소스를 검색함으로써 획득한다.

1. 로컬에 내장된 폰트
2. 별도의 SWF/GFX 파일로부터 불러온 폰트 기호
3. Gfx::FontLib 을 통해 설치된 SWF/GFX 파일, 폰트명 또는 불러온 폰트 교체를 통해 검색이 이뤄짐
4. 사용자가 설치한 경우 Gfx::FontProviderWin32 와 같은 시스템 폰트 제공자

내장 및 불러온 폰트의 사용이 이 문서 첫 부분에 나와 있다. 대부분 내장 폰트는 플래시와 동일하게 작동하므로 별도의 커스텀 설정이 필요하지 않다. 폰트 록 업을 위해 불러온 폰트 기호는 내장 폰트와 비슷한 역할을 한다.

내장되지 않은 폰트 록 업을 설정하는 세 가지 설치 가능한 상태는 Gfx::FontLib, Gfx::FontMap 및 Gfx::FontProvider 이다. 이 문서 첫 부분에서 다룬 바와 같이, Gfx::FontLib 과 Gfx::FontMap 은 불러온 폰트 교체 또는 장치 폰트 에뮬레이션을 위해 SWF/GFX 에 로딩된 폰트를 록 업하기 위해 사용한다. 시스템 폰트 제공자의 사용에 관한 자세한 내용은 아래에서 다룬다 .

시스템 폰트 제공자는 Gfx::Loader::SetFontProvider 호출로 설치된다. 이를 통해 폰트 데이터를 비 SWF 파일 소스로부터 불러올 수 있다. 폰트 제공자는 동적 캐시와만 같이 사용할 수 있고 폰트 데이터가 SWF 또는 폰트 라이브러리에 내장되지 않은 경우에만 검색이 이뤄진다. 현재, 두 개의 폰트 제공자가 Scaleform 에 포함되어 있다.

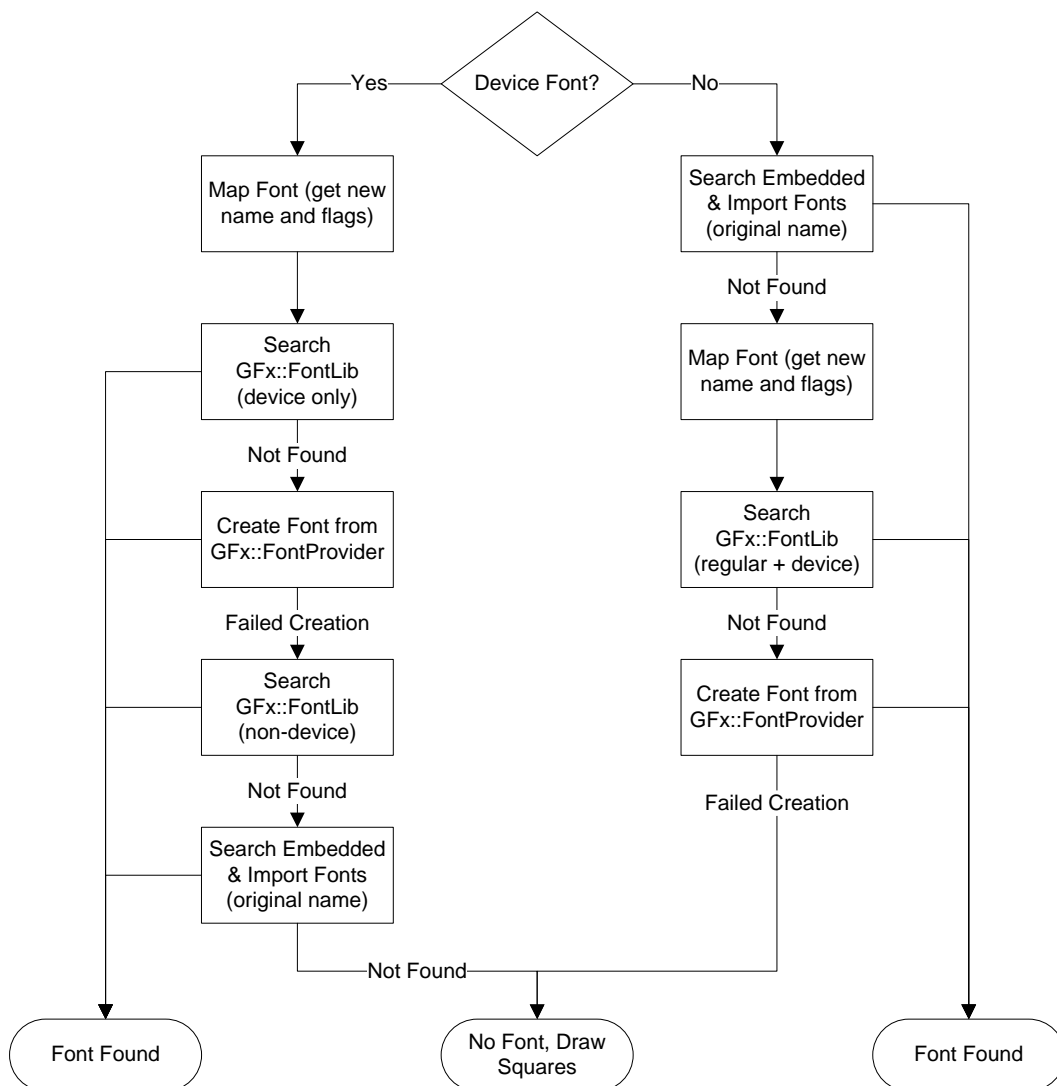
- Gfx::FontProviderWin32 – 폰트 글리프 데이터 획득을 위해 Win32 에 따름
- Gfx::FontProviderFT2 – 스탠드 얼론 폰트 파일을 읽고 해석하기 위해 David Turner 가 개발한 FreeType-2 라이브러리 사용

여기에선 폰트 록 업 순서를 자세히 다루고 상이한 폰트 소스를 어떻게 설정할 수 있는지에 대한 코드 예제를 다룬다.

## 4.1 폰트 록업 순서

다음 페이지의 순서도는 Scaleform 에서 폰트 록업의 순서를 보여주고 있다. 그림에서 보는 바와 같이, 록업 속성은 장치 폰트 플래그를 통해 수정되며 "장치 폰트 사용"을 선택하는 경우 렌더링 기법을 텍스트 속성에서 선택한다.

어도비 플래시에서, 장치 폰트 플래그 설정은 해당 폰트를 내장 폰트와 동일한 이름을 갖는 폰트를 시스템으로부터 획득할 수 있도록 한다 (이 경우, 후자는 fall-back 으로 사용). Scaleform 는 이러한 속성을 그대로 사용하나 설치된 폰트 라이브러리를 시스템 폰트 제공자에 앞서 검색한다. 이러한 설정으로 인해 충돌이 발생하지 않는데 그 이유는 대부분의 콘솔-휴대용 게임은 공유된 폰트 라이브러리에 따르나, 시스템 제공자를 사용하기 위해 선택한 게임은 폰트 라이브러리를 초기화되지 않은 (null) 상태로 놓을 수 있기 때문이다.





다이어그램에서 보는 것과 같이, 장치 폰트가 텍스트 필드에 사용되지 않을 경우 내장 폰트를 우선 검색하고 그렇지 않은 경우 마지막에 검색한다. 이에 더해, 내장 폰트는 텍스트 필드에서 사용한 원래 폰트명을 기준으로 항상 록 업되는 반면 폰트 맵은 Gfx::FontLib 과 Gfx::FontProvider 로부터 록 업이 이뤄지는 폰트명 교체를 위해 적용이 가능하다.

## 4.2 Gfx::FontMap

Gfx::FontMap 은 폰트명 교체를 위해 사용되는 상태로 필요한 문자를 개발 폰트에서 사용할 수 없을 때 국제화 과정간 대체 폰트를 사용할 수 있도록 한다. 폰트 맵은 불러온 폰트 교체 및 장치 폰트 에뮬레이션 모두와 사용된다. 전자의 경우 폰트 기호 확인자를 폰트명으로 변환한다. 후자의 경우, 원래의 파일명을 변환이 이뤄진 폰트명으로 매핑시킨다.

제 3 부에서 폰트 설정 파일 내에 다음과 같은 라인을 추가해 폰트 맵을 생성한다.

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

위의 예에서, 매핑 문장을 사용하여 "\$TitleFont"와 같은 폰트 불러오기 확인자를 실제 폰트명과 매핑시키며 이 경우 폰트 라이브러리 파일에 내장된 것이다.

다음과 같은 C++ 문장을 통해 동일한 폰트 맵 설정이 이뤄진다.

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

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

이 경우 세 폰트 모두 동일한 폰트명으로 매핑된다. 특히 "\$NormalFont"와 "\$SmallFont"는 동일한 폰트 스타일을 공유하며 이를 통해 폰트 라이브러리 파일이 사용하는 메모리를 절약한다. 세 번째 문장으로 서로 다른 스타일을 MapFont 에 정의할 수 있으며 이로 인해 매핑이 특별한 내장을 사용하게 된다. 만약 설명을 정의하지 않으면, MFF\_Original 값이 사용되는데 이는 폰트 록 업은

텍스트 필드에서 정의한 원래 스타일을 유지해야 한다는 것을 의미한다. Font 의 사이즈는 scaleFactor 변수에 의해 정해지는데 Default 값은 1.0f 이다. 이 변수는 Font 가 잘 보이지 않을 때나 약간 크기를 키울 때 용이하다.

개발자는 Gfx::FontMap 이 결합 상태, 즉 이와 생성된 영화 인스턴스는 차후 로더 안에서의 변경이 있더라도 해당 상태를 사용한다는 점을 의미한다. 만약 다른 폰트 맵을 설정하면, 상이한 Gfx::MovieDef 가 동일한 파일명에 대해 Gfx::Loader::CreateMovie 를 통해 반환된다. 이는 또한 기타 폰트 설정 상태에 대해 참이다.

### 4.3 Gfx::FontLib

파트 3 에서 설명한 바와 같이, Gfx::FontLib 상태는 (1) 기본 "fonts\_en.swf"에서 가져오기 한 폰트에 대한 대안을 제공하고 (2) 장치 폰트 에뮬레이션을 위한 폰트를 제공하는, 설치 가능한 폰트 라이브러리를 나타냅니다. 폰트 라이브러리는 시스템 폰트 제공자 이전에 검색된다. 다음의 예를 통해 그 사용을 분명히 한다.

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

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

중요한 것은 평상시와 같이 영화를 생성 및 로딩하는 것이지만 이를 재생하는 대신 영화를 폰트 저장공간으로 사용한다. 필요한 만큼의 영화를 로딩하는 것이 가능하다. 만약 동일한 폰트를 서로 다른 영화에서 정의한다면 첫 번째 영화에서만 사용된다.

AddFontsFrom 의 첫 번째 매개변수는 폰트 소스의 역할을 하는 영화의 정의이다. AddFontsFrom 의 두 번째 매개변수는 핀 플래그로서 로더가 메모리 내 영화로 AddRef 를 하는 경우 반드시 설정해야 한다. 이 플래그는 사용자가 로딩된 영화에 스마트 포인터를 유지하지 않을 경우에만 필요하다 (예를 들어 m1, m2). 만약 핀 플래그가 거짓인 경우, 내보내거나 패킹 된 텍스처와 같은 폰트 결합 데이터를 조기에 배출하여 폰트를 사용할 때 재로딩/재생성해야 한다.

폰트 맵과 유사하게, Gfx::FontLib 은 결합 상태이며 사라질 때까지 생성된 영화를 통해 참고된다.

## 4.4 Gfx::FontProviderWin32

Gfx::FontProviderWin32 폰트 제공자는 Win32 API 와 가용하며, 벡터 데이터 회수를 위해 GetGlyphOutline()에 의존할 수 있다. 다음과 같이 사용이 가능하다.

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

Gfx::FontProviderWin32 의 구성자는 내용으로서 Windows Display Context 의 핸들러를 취한다. 대부분의 경우 화면 DC 의 사용이 적합하다 (::GetDC(0)).

요청이 있을 때 필요한 경우 폰트를 생성한다.

### 4.4.1 네이티브 힌트 텍스트 사용

일반적으로 폰트 힌트는 매우 중요한 문제이다. Scaleform 는 자동 힌트 메커니즘을 제공하나 중국어, 일본어 및 한국어 (CJK) 문자에 대해선 제대로 작동하지 않는다. 이에 더해, 설계가 제대로 된 대부분의 CJK 폰트는 특정 크기의 글리프에 대해 래스터 이미지를 포함하는데 그 이유는 CJK 의 적절한 힌트가 있는 작은 크기의 폰트는 극도로 복잡하기 때문이다. 래스터 이미지로 벡터 글리프를 복제하는 것은 양호하고 실질적인 해결책이다. 폰트 API (Gfx::FontProviderWin32 와 Gfx::FontProviderFT2)를 기준으로 하는 시스템 폰트 제공자는 벡터와 래스터 이미지 모두를 사용해 글리프를 생성할 수 있다. 폰트 제공자는 네이티브 힌트를 제어하기 위해 인터페이스를 갖는다. 인터페이스는 약간씩 다르나 동일한 원리를 갖는다. 네 가지 매개변수가 있다.

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

Parameters vectorRange 와 rasterRange 매개변수는 힌트를 사용하는 문자 범위를 제어한다. 값은 다음과 같다.



```
maxVectorHintedSize = 24;
maxRasterHintedSize = 24;
```

힌트를 설정하는 특정 인터페이스는 아래에 있다.

#### 4.4.2 네이티브 힌트 설정

정의에 따라 Gfx::FontProviderWin32 는 CJK 문자에 대해 네이티브 래스터 힌트를 사용하며 벡터 네이티브 힌팅 ()을 사용한다. 다음을 호출하여 속성 변경이 가능하다.

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

또는

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

SetHinting()을 호출한 후 SetHintingAllFonts()를 호출하면 이러한 특정 폰트에 대해 힌트 속성이 변경되지 않음을 확인한다. 정확한 형태의 함수는 다음과 같다.

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

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

fontName 매개변수는 UTF-8 인코딩을 사용할 수 있다.

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

인자의 형태는 다음의 타입으로 정할 수 있다.

```
GGO_BITMAP (by default),
GGO_GRAY2_BITMAP,
GGO_GRAY4_BITMAP,
GGO_GRAY8_BITMAP
```

이 값들은 Gfx::FontProviderWin32.h 내부의 <windows.h>에 정의되어 있다.

감마 인자는 0~255 범위의 픽셀 값인 unsigned byte 의 배열이며, GGO\_GRAY2\_BITMAP 위한 선택 사항이다. 그러나 GGO\_GRAY4\_BITMAP 과 GGO\_GRAY8\_BITMAP 은 정의 되어야 한다.

GGO\_GRAY2\_BITMAP 은 0~4 범위의 픽셀들을 발생시키며, GGO\_GRAY4\_BITMAP 은 0~16, GGO\_GRAY8\_BITMAP 은 0~64 의 범위 값을 갖는다. 그러므로 감마 배열은 5, 17 과 65 의 값을 각각 포함하여야 한다. 첫 번째 값은 0 이어야 하며, 마지막 값은 255 이어야 한다. 다른 값들은 0~255 의 감마 곡선을 보간 해야 한다. 간단한 선형 보간의 경우의 예로는 선형 감마인 GGP\_GRAY2\_BITMAP 은 0, 63, 127, 191, 255 입니다.

## 4.5 Gfx::FontProviderFT2

이 폰트 제공자는 David Turner 를 통해 FreeType-2 를 사용한다. Gfx::FontProviderWin32 와 사용법이 유사하나 폰트명과 속성을 실제 폰트 파일에 매핑해야 한다는 점이 다르다.

우선 개발자들이 FreeType 매뉴얼을 숙지하여 라이브러리를 적절히 설정하고 구성해야 함을 권장한다. 정적 또는 동적 링크를 사용하고 폰트 드라이버, 메모리 할당, 외부 파일 스트림 등과 같은 적절한 런타임 설정에 관한 결정을 내리는 것은 개발자의 책임이다.

정적 링크가 있는 윈도우 MSVC 컴파일러에 대해 다음과 같은 라이브러리를 사용한다.

freetype<ver>.lib – 다중 스레드 DLL 코드 생성을 위해

freetype<ver>\_D.lib – 디버그 다중 스레드 DLL 을 위해

freetype<ver>MT.lib – 다중 스레드 배포를 위해 (정적 CRT)

freetype<ver>MT\_D.lib – 디버그 다중 스레드를 위해 (정적 CRT)

Where "<ver>"는 FreeType 의 버전이며, 버전 2.1.9 의 경우 219 로 표시한다.

또한 freetype2\include 와 freetype2\objs 디렉토리는 추가적인 include 와 라이브러리 각각에 대해 경로상에서 가용하다는 점을 확인한다.

Gfx::FontProviderFT2 객체의 생성은 다음과 같다.

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

구성자는 다음과 같은 구조를 갖는다.

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

이는 FreeType 라이브러리 처리이다. 만약 0 인 경우 (기본값), 제공자는 FreeType 을 내부에서 초기화한다. 기존의 초기화 된 외부 핸들러를 정의할 수 있는 능력은 어플리케이션이 이미 FreeType 을 사용하고 Scaleform 와 기타 시스템 사이에서 핸들을 공유하려는 경우 제공된다. 외부 핸들러를 사용할 때 라이브러리를 적절히 배포하는 것은 개발자의 책임임을 주지한다 (FT\_Done\_FreeType 을 호출). 또한 어플리케이션은 핸들의 수명이 Gfx::Loader 의 수명보다 **길다**는 점을 보장해야 한다. 외부 핸들러의 사용은 또한 FreeType 의 런타임 콜 백을 재구성하는 게 바람직한 경우 지원된다 (메모리 할당자 등). 이러한 설정은 또한 내부 초기화로 이뤄질 수 있음을 주의한다. fontProvider->GetFT\_Library() 함수는 사용한 FT-Library 를 반환하며 FreeType 에 대한 호출이 폰트의 실제 사용 이전 제공자를 통해 호출되지 않았음을 보장한다. 그 결과, Gfx::FontProviderFT2 의 생성 후 런타임 시 FreeType 을 재설정할 수 있다.

#### 4.5.1 FreeType 폰트를 파일로 매핑

Win32 API 와는 다르게 FreeType 은 실제 폰트 파일로 서체 명 (그리고 속성)으로의 매핑을 제공하지 않는다. 그러므로 이러한 매핑을 외부에서 제공해야 한다. Gfx::FontProviderFT2 는 폰트를 파일과 메모리로 매핑하는 단순한 메커니즘을 갖고 있다.

```
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);
```

"typeface"는 예를 들면 "Times New Roman"과 같은 폰트의 서체명을 정의한다. "fileName"은 예를 들면 "C:\\WINDOWS\\Fonts\\times.ttf"와 같은 폰트 파일로의 경로를 정의한다. "fontFlags"는 "Font::FF\_Bold", "Font::FF\_Italic" 또는 "Font::FF\_BoldItalic" 값을 갖는다. "Font::FF\_BoldItalic"값은 실제로 "Font::FF\_Bold | Font::FF\_Italic"와 동일하다. "faceIndex"는 FT\_New\_Face 에 전달된다. 대부분의 경우 이 매개변수는 0 이지만 항상 0 은 아니며 이는 폰트 파일의 내용과 형태에 따른다. 이러한 함수가 실제로 폰트 파일을 여는 것은 아니며 단지 매핑 테이블을 구성한다는 점을 주의한다. 폰트 파일은 실제로 요청이 들어온 경우 열린다. Win32 폰트 제공자와는 달리, FreeType 은 네이티브 힌트를 구성하기 위해 함수 인자만을 사용한다.

## 4.5.2 폰트를 메모리로 매핑

FreeType 은 폰트가 메모리로 매핑될 수 있도록 한다. 예를 들어 하나의 폰트 파일이 많은 서체를 포함하거나 폰트 파일이 이미 어플리케이션을 통해 로딩된 경우라면 타당하다.

```
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);
```

폰트를 메모리로 매핑하는 단순한 (그리고 어느 정도는 지저분한) 예가 다음과 같다.

```
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);
}
```

이 예에서 메모리를 배포하진 않는다 (그리고 결국에는 메모리 누출을 야기한다). 매핑 테이블은 단순히 naked 된 상수 포인터를 폰트 데이터에 유지하기 때문이다. 이를 제대로 배포하는 것은 개발자의 책임이다. 어플리케이션은 반드시 메모리 블록의 수명이 Gfx::Loader 의 수명보다 **길어야 한다는** 점을 보장해야 한다. 폰트 매핑 메커니즘에 최대한의 자유를 주도록 한다. 예를 들어, 어플리케이션은 할당 및 파괴가 Scaleform 를 통해 외부에서 처리되는 미리 로딩된 폰트가 있는 FreeType 을 이미 사용할 수 있다.

윈도우에서 FreeType 폰트 매핑의 사용 예는 다음과 같다.

```
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);

```

위의 내용은 단순히 예임을 주의한다. 실제 활용에서 하드 코딩된 파일의 절대 경로를 지정하는 것은 좋지 않다. 일반적으로 설정 파일을 통해 작업하거나 자동으로 폰트 서체를 검색해서 확인해야 한다. 서체명을 명시적으로 정의하는 것은 폰트가 일반적으로 이러한 서체명을 포함하고 있기 때문에 지나칠 수 있으나 메모리 소비가 큰 파일 파싱을 피할 수 있다. MapFontToFile() 함수는 이러한 정보만을 저장하며 요청을 받은 경우가 아니라면 파일을 열지 않는다. 그 결과, 매핑된 폰트 다수를 정의할 수 있는 반면 이 중 일부만이 실질적으로 사용된다. 이 경우 추가적인 파일 작업을 실시할 필요가 없다.

## 5 제 4 부: 폰트 렌더링 설정

Scaleform 는 두 가지 방법으로 텍스트 문자를 렌더링 할 수 있다. 첫 번째로, 폰트 글리프를 텍스처 내로 래스터화 한 후 텍스처가 있는 삼각형의 배치를 사용해서 그리는 것이다 (문자 당 2 개). 대안으로 폰트 글리프를 삼각형 메시로 넣은 후 벡터 도형으로 렌더링할 수 있다.

어플리케이션 내 대부분의 텍스트에서 성능상의 이유로 항상 텍스처 삼각형을 사용해야 한다. 글리프 텍스처는 동적 또는 정적 캐싱을 통해 생성된다. 동적 글리프 캐싱이 더욱 유연하고 품질이 높은 이미지를 생성하며 정적 캐싱은 오프 라인으로 계산이 가능하다는 장점이 있다.

목표 플랫폼 및 제목에서의 필요한 부분에 따라 개발자는 다음과 같은 폰트 렌더링 옵션을 선택할 수 있다.

1. 빠르고 품질이 좋은 애니메이션을 만드는 동적 캐시를 사용한다. 동적 캐시는 고정량의 텍스처 메모리를 사용하므로 모든 글리프를 래스터로 만들고/만들거나 로딩할 필요가 없을 때 로딩 시간을 절감하게 된다.
2. 디스크에서 미리 생성된 mip-mapped 텍스처를 로딩하는 정적 캐시를 사용한다. 개발자는 gfxexport 툴을 사용하여 패킹된 글리프 텍스처를 이미 생성하여 폰트 벡터 데이터를 제거함으로써 로딩시킬 필요가 없도록 할 수 있다.
3. Gfx::FontPackParams 로 로딩 시간에 자동 생성되는 텍스처인 정적 캐시를 사용한다. 이는 앞에 언급한 방법과 유사하나 텍스처가 디스크에서 로딩될 필요는 없다.
4. 텍스트 글리프를 직접 렌더링하기 위해 벡터 도형을 사용한다.

PC, Xbox 360 PS3 및 많은 경우 닌텐도 Wii 와 같이 최종 시스템의 사용이 높은 경우 동적 캐시를 권장한다. 동적 캐시는 로딩 시간을 최소화하고 폰트를 주어진 해상도에 최대 가장 높은 품질로 렌더링 하는 것을 보장한다. 동적 캐시의 사용은 또한 개발자가 시스템을 활용하거나 Gfx::FontProviderWin32 또는 Gfx::FontProviderFT2 중 하나를 통해 외부 폰트 지원을 계획하는 경우 필요하다.

gfxexport 툴을 통해 내보내는 텍스처를 갖는 정적 캐시 사용은 PSP 및 PS3 와 같이 CPU 와 메모리가 제한되고 글리프의 런타임 래스터에 메모리가 너무 많이 사용되는 플랫폼에 좋은 옵션이다. 또한 개발자가 렌더링 구현에 있어 동적 텍스처 업데이트를 피하려는 경우 정적 캐시를 사용할 수 있다.

그러나 벡터 데이터 메모리 사용을 최적화함에 따라 저사양 시스템에서도 동적 캐시가 좋은 선택이 될 수 있다. 최상의 솔루션을 선택하는 데 있어 게임 데이터를 개발자들이 다루도록 권장한다.

Scaleform 를 이런 식으로 설정할 수 있으나 벡터 도형은 텍스트 렌더링을 위해 스탠드 얼론에선 거의 사용되지 않는다. 대신 크기가 큰 글리프에 대해서만 글리프 채우기를 일반적으로 사용하며 작은 크기의 텍스트를 효과적으로 렌더링하기 위해 텍스처 기반의 기법을 사용한다. 개발자는 이러한 옵션을 어떻게 제어하고 사용 불가능하게 하는지에 대한 자세한 사항을 벡터화 제어를 통해 참고할 수 있다.

## 5.1 문자 캐시(Glyph cache) 구성하기

Scaleform 에서 폰트를 렌더링하려면 `Render::GlyphCacheConfig` 나 `Gfx::FontPackParams` 상태 객체를 통해 이를 구성해야 합니다. 문자 캐시는 기본적으로 렌더 스레드에 있는 `Renderer2D` 객체가 자동으로 만들고 동적 문자 캐싱을 위해 초기화합니다. 글리프 패커는 정적 텍스처 초기화에서만 사용됩니다. 기본적으로 패킹 매개변수는 `null` 이다. 이러한 설정과 함께, 정적 텍스처를 갖기 위해 Scaleform 파일을 미리 처리하지 않는 경우 생성된 모든 영화는 자동으로 동적 폰트 캐시를 사용한다.

### GfX 4.0 업그레이드 정보

Scaleform 3.x 버전에서 4.0 버전으로 업그레이드하면서 동적 문자 캐시를 구성하는 방법이 상당히 바뀌었습니다. GfX 3.3에서는 `GfxLoader`가 관리하는 `GfxFontCacheManager` 객체에 의존하고 이를 주 스레드에서 구성했지만, GfX 4.0에서는 이를 `Renderer2D`가 관리하는 `GlyphCacheConfig` 인터페이스로 대체되었으며 렌더 스레드에서 구성하도록 변경되었습니다. Scaleform GfX 4.0에서 캐시를 구성하는 자세한 정보를 보려면 섹션 5.2를 참조하십시오.

텍스처에서 래스터 문자를 렌더링할 경우 항상 문자 캐시 시스템이 사용됩니다. 내부적으로, 캐시 관리자는 텍스트 배치 정점 배열을 유지할 책임이 있는데 이는 동적 및 정적 캐시 모두를 사용할 때 생성된다. 동적 캐싱이 가능할 때, 캐시 관리자는 캐시 텍스처 할당, 래스터화 된 문자와의 업데이트 및 배치 정점 데이터로 텍스처를 동기화하는 책임이 있다. 정적 캐싱만을 사용할 때, 캐시 관리자는 여전히 텍스트 정점 배열을 유지할 필요가 있으나 동적 텍스처를 할당하거나 갱신할 필요는 없다.

조밀하게 패킹된 글리프로 기존에 래스터 된 비트맵 텍스처를 통해 정적 캐시를 표현한다. 정적 텍스처의 래스터화와 패킹은 `Gfx::FontPackParams`를 기반으로 로딩 시 또는 'gfxexport' 툴로 오프라인에서 이뤄질 수 있다. 어떻게 로딩되는지에 따라 내장 폰트는 자신의 폰트 글리프를 포함하는

동적 텍스처 집합을 갖거나 갖지 않게 된다. 만약 폰트가 정적 텍스처를 갖는다면 해당 폰트의 글리프를 렌더링하는 데 항상 사용된다. 그렇지 않으면 동적 캐시는 가용한 경우 사용된다.

사용중인 폰트 텍스처의 형태에 따르는 것에 더해, 렌더링 기법은 또한 글리프의 최종 픽셀 크기에 따른다. 다음과 같은 로직을 사용한다.

```
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;
}
```

위에 언급한 바와 같이, 텍스처 캐시를 사용할 수 없거나 글리프가 너무 커서 텍스처로부터 렌더링이 될 수 없을 때 벡터 렌더링을 사용한다. 캐시 접근법은 폰트 내 패킹된 글리프 텍스처의 가용성을 기반으로 한다. 두 가지 접근방식이 어떻게 설정될 수 있는가에 대한 세부사항을 다음에서 다룬다.

## 5.2 동적 폰트 캐시 사용

정적 캐시는 기본 라틴어, 그리스 어, 키릴 어 등과 같이 폰트와 문자셋이 제한되는 경우 효과적이다. 그러나 대부분의 아시아 언어에 있어 정적 캐시는 너무 많은 시스템과 비디오 메모리를 소비하므로 결과적으로 로딩이 느려진다. 이에 더해, 너무 많은 다양한 서체를 사용하는 경우에도 발생할 수 있다. 즉, 내장 글리프의 총 수가 큰 경우 (약 1 만개 이상)를 들 수 있다. 이러한 경우 동적 캐시 메커니즘을 사용하는 것이 타당하다. 이에 더해, 동적 캐시는 품질을 크게 향상시킬 가독성을 위한 최적화와 같은 더 많은 기능을 제공한다. 동적 캐시는 기본적으로 활성화되고 버퍼를 할당합니다. 이를 비활성화하려면 다음을 호출하십시오.

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

위의 호출에서 `renderer` 는 렌더 스레드에서 관리되는 `Render::Renderer2D` 객체입니다. 이를 호출하면 문자 캐시가 사용하는 동적 텍스처 수를 0 으로 설정하므로 문자 캐시를 비활성화하는 것과 마찬가지로입니다. 임시 버퍼가 기본으로 할당되지 않도록 하려면 HAL 이 초기화되기 전에 이 호출을 완료해야 합니다.

동적 캐시는 글리프를 래스터화 하고 텍스트를 그릴 때 요청에 따라 해당 텍스처를 갱신한다. 단순한 LRU (최소 최근 사용) 캐시 개념을 사용하나 "on the fly"에서 실행되는 텍스처로 스마트 글리프를 묶는다. 다음과 같이 텍스처 매개변수를 설정할 수 있다.

```
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);
```

위의 값들은 기본값으로 사용된다.

`TextureWidth`, `TextureHeight` - 캐싱 텍스처의 크기. 두 값 모두 가장 가까운 2 의 승수로 반올림된다.

`MaxNumTextures` - 캐싱에 사용하는 최대 텍스처 수

`MaxSlotHeight` - 글리프의 최대 높이. 실제 픽셀 글리프 높이는 이 값을 초과할 수 없다. 이보다 큰 글리프는 벡터로 렌더링된다.



`SlotPadding` - 글리프의 클리핑 및 오버래핑 방지를 위해 사용하는 마진 값. Value 2 면 대부분의 경우 충분하다.

`TexUpdWidth`, `TexUpdHeight` - 텍스처 갱신을 위해 사용하는 이미지의 크기. 일반적으로 256x512 (128K 의 시스템 메모리)면 모든 경우에 있어 충분하다. 256x256 또는 128x128 로 축소가 가능하지만 이 경우 텍스처 업데이트가 더 자주 일어난다.

총 캐시 용량 (캐시에 동시에 저장되는 서로 다른 글리프의 최대 개수)은 평균 글리프 크기에 의존한다. 일반적인 게임 UI 에 대해서 위의 매개변수를 통해 500 에서 2,000 개의 상이한 글리프의 캐싱을 예상할 수 있다. 일반적인 상황에서 70-75% 정도가 효과적으로 사용된다. 캐시가 이뤄진 글리프의 최대 개수는 단일 텍스트 필드에서 눈에 보이는 부분을 처리할 수 있을 정도로 충분해야 한다. 이 경우

단일 텍스트 필드의 가시적인 부분에 있는 서로 다른 글리프의 수가 캐시 용량을 벗어나는 경우 나머지 글리프는 벡터 도형으로 그려진다.

위에 언급한 바와 같이 동적 캐시는 추가적인 기능을 제공하며 “가독성을 위한 안티 앨리어싱” 및 “애니메이션용 안티 앨리어싱” 옵션 지원을 제공한다. 이러한 옵션은 텍스트 필드의 속성으로 플래시 디자이너는 텍스트 대화창 패널에서 선택이 가능하다. 가독성을 위해 최적화 된 텍스트는 더 날카롭고 읽기 쉬워 보인다. 애니메이션으로 만들 수 있지만 이러한 애니메이션은 더욱 빈번한 업데이트가 발생하기 때문에 메모리를 더 많이 사용한다. 이에 더해, 글리프는 자동으로 픽셀 그리드로 결합되어 픽셀 자동 맞춤 작동으로 뿌연 부분을 감소시킨다. 이는 텍스트 라인 또한 픽셀로 결합됨을 의미한다. 실질적으로 애니메이션하는 경우 특히 크기를 변화시키는 경우 지터 효과를 갖게 된다. 아래의 그림에서 이러한 옵션 사이의 차이점을 보여준다.

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

“가독성을 위한 안티 앨리어싱”을 사용할 때 글리프 래스터 장치는 자동 맞춤 절차 (또한 오토 힌트라고 알려진)을 실행한다. Scaleform 는 플래시 파일 또는 기타 폰트 소스로부터 어떤 종류의 글리프 힌트도 사용하지 않는다. 텍스트 렌더링에 필요한 것은 글리프 외곽선이다. 폰트 소스와는 별개로 텍스트와 동일한 모습을 제공한다. Scaleform 의 자동 힌터는 폰트로부터의 특정 정보, 즉 상단이 평평한 라틴 문자의 높이를 필요로 한다. 글리프를 overshoot (예, O, G, C, Q, o, g, e 등)로 올바르게 묶을 필요가 있다. 일반적으로 폰트는 이러한 정보를 제공하지 않으므로 어느 정도 추론할 필요가 있다. 이러한 목적으로 Scaleform 는 위가 평평한 라틴 문자를 사용한다.

대문자의 경우 H, E, F, T, U, V, W, X, Z

소문자의 경우 z, x, v, w, y

자동 피터링을 사용하기 위해 폰트는 최소한 위의 대문자 중 하나와 소문자 중 하나를 포함해야 한다. 폰트가 이를 포함하지 않으면 Scaleform 는 다음과 같은 로그 경고를 생성한다.

**“경고: 폰트 ‘Arial’; 힌트 문자가 없음 (‘HEFTUVWXZ’ 및 ‘rxvwy’). 자동 힌트 사용 불가”**

이러한 경고가 나타나면 플래시 디자이너는 이와 같은 문자를 내장해야 한다. 대화창에서 “문자 내장” 내에 “Zz”를 추가하거나 “Basic Latin”을 추가하면 된다.

### 5.3 폰트 압축기- *gfxexport* 사용하기

명령줄 *gfxexport* 도구의 목적은 SWF 파일을 사전 작업하여 게임에 배포하고 불러오는 GFX 파일을 생성하기 위함입니다. 사전 작업 동안, 도구는 SWF 파일에서 이미지를 빼내어 외부 파일로 추출할 수 있습니다. 외부 파일을 DDS 및 TGA 와 같은 여러 유용한 형식으로 저장할 수 있습니다. -fc 옵션을 선택하면, *gfxexport* 가 폰트 벡터 데이터도 압축하게 됩니다. 이것은 손실이 큰 압축이므로, 메모리 소모와 폰트 품질 사이에서 최적 절충안을 내는 파라미터를 이용한 테스트가 필요할 수 있습니다.

명령줄 옵션	동작
-fc	폰트 압축기를 활성화합니다.
-fcl <size>	명목 문자 크기를 설정합니다. 작은 명목 크기는 더 작은 데이터 크기를 갖지만 덜 정확한 문자를 생성하게 됩니다. 기본값은 256 입니다. 명목 크기를 256 으로 설정하면 일반적으로 가시적인 품질 저하 없이 약 25%의 메모리를 절약할 수 있습니다(Flash 에서 보통 사용하는 명목 크기 1024 에 대비했을 때). 그러나 아주 큰 문자에 대해서는 명목 크기를 늘릴 필요가 있습니다.
-fcm	압축 폰트에 대한 에지를 병합합니다. FontCompactor 가 동일한 윤곽 및 문자를 병합해야 함을 알려주는 Boolean 플래그입니다. 병합 시, 폰트에 따라 데이터를 더 압축하여 10 ~ 70%의 메모리를 절약할 수 있습니다. 그러나 폰트가 너무 많은 문자를 포함하는 경우에는, 해쉬 테이블이 각각의 고유 경로당 12(32 비트) 또는 16(64 비트) 바이트 + 각각의 고유 문자당 12(32 비트) 또는 16(64 비트) 바이트의 추가 메모리를 소모할 수 있습니다.

### 5.4 폰트 텍스처 전처리 - *gfxexport*

-fonts 옵션을 정의하면, *gfxexport* 는 패키징된 폰트 텍스처를 래스터 및 내보내며 사용자가 지정한 형태로 저장한다. Scaleform Player 에서 GFX 파일을 로딩할 때, 외부 텍스처가 자동으로 로딩되며 텍스트 렌더링 간 정적 캐시로 사용된다.

**일반적인 GfX 응용 프로그램에 대해 내보내기 한 텍스처의 사용은 권장하지 않습니다.** 그러나 저성능 모바일 플랫폼이나 일부 특별한 경우에는 이쪽이 유리할 수 있습니다.

폰트 텍스처 생성을 위해 `gfxexport` 를 사용하면 다음과 같은 장점이 있다.

- 외부 텍스처 저장을 통해 글리프 벡터 데이터를 분리하여 메모리를 절약할 수 있다. 그러나 글리프 데이터는 대신 로딩된 정적 텍스처를 통해 대체된다.
- CPU 에 제약이 있는 시스템에서 로딩 시간에 텍스처 파일을 생성하는 것보다 로딩하는 시간이 빠를 수 있다.
- 텍스처 파일을 컴팩트 한 게임별 포맷으로 변환할 수 있으며 `GfX::ImageCreator` 를 override 함으로써 로딩이 가능하다. 이러한 포맷을 직접 지원하는 고유의 `Render::Renderer` 를 구현할 때 좋다.

그러나 미리 생성된 텍스처를 사용하는 것은 동적 캐시를 사용하는 것에 비해 낮은 품질의 텍스트 렌더링과 잠재적으로 더 긴 로딩 시간이라는 단점을 갖는다. 정적 텍스트는 mip-maps 를 사용하므로 힌트가 될 수 없다. 이는 “가독성을 위한 안티 앨리어스” 설정을 사용하지 않음을 의미한다.

다음의 명령은 ‘test.swf’를 ‘test.gfx’로 사전 처리하며 내장된 모든 폰트에 대해 추가적인 텍스처 파일을 생성한다.

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

`-strip_font_shapes` 옵션은 내장된 폰트 벡터 데이터를 생성된 Scaleform 파일로부터 분리한다. 이러한 방법이 메모리를 절약하지만 큰 문자에 대한 벡터 렌더링이 불가능하며 이로 인해 폰트 텍스처의 일반적인 글리프 크기를 벗어나는 경우 품질 저하가 일어난다.

다음의 표는 `gfxexport` 에 대해 폰트와 관련된 목록을 나타낸다. 텍스처 크기, 일반적인 글리프 크기 및 목표 파일 포맷을 제어하기 위해 옵션을 제공한다. 이러한 옵션 중 대부분은 다음에서 언급할 글리프 패커 매개변수에 대응하는데 그 이유는 폰트 텍스처를 생성할 때 `gfxexport` 가 글리프 패커를 사용하기 때문이다.

명령줄 옵션	속성
-fonts	폰트 텍스처를 내보낸다. 지정하지 않은 경우 폰트 텍스처는 생성되지



명령줄 옵션	속성
	않는다 (대신 로딩 시간에 동적 캐시 또는 패킹을 허용)
-fns <size>	픽셀로 된 텍스트 글리프의 일반적인 크기. 지정하지 않은 경우 기본값인 48 이 사용된다. 텍스처 내 한 글리프의 크기가 최대인 경우 일반적인 크기. 작은 문자는 Tri-linear mip-map 필터링을 사용해서 런타임 시 렌더링된다.
-fpp <n>	개별 글리프 이미지 주변의 픽셀로 된 공간. 기본값은 3 이다.
-fts <WxH>	글리프가 패킹되는 텍스처 크기. 기본 크기는 256x256 이다. 정사각형 텍스처의 정의를 위해 단 하나의 크기만 정의할 수 있다. ('-fts 128'은 128x128. '-fts 512x128'는 직사각형 텍스처를 지정)
-fs	각 폰트에 관해 별개의 텍스처를 사용. 기본적으로 폰트는 텍스처를 공유한다.
-strip_font_shapes	폰트 글리프 모양 데이터를 결과로 나오는 GFX 파일에 기록하지 않는다.
-fi <format>	<format>이 TGA8 (그레이스케일), TGA24 (그레이스케일), TGA32 또는 DDS8 인 경우 폰트 텍스처에 대한 출력 포맷을 정의한다. 기본적으로 이미지 포맷 (-i 옵션)이 TGA 인 경우 TGA8 은 폰트 텍스처에 사용되며 기타의 경우 DDS A8 을 사용한다.

**경고:** 게임에서 압축된 정적 폰트나 문자 압축기만 사용하려면 섹션 5.2 에 설명한 대로 동적 문자 캐시 텍스처 할당을 비활성화해야 합니다. 이렇게 하지 않으면 사용하지 않더라도 기본 텍스처가 할당됩니다.

## 5.5 폰트 글리프 벡터 설정

Font Glyph Packer 는 파일을 로딩할 때 내장 글리프를 래스터하고 포장한다. 또한 GFxExport 를 통해 폰트를 미리 래스터하는 게 가능하다. 위에서 언급한 바와 같이 폰트 캐시 관리자는 동적 및 정적 메커니즘 모두를 동시에 사용할 수 있다. 예를 들어 문자셋이 큰 폰트가 동적 캐시를 사용하는 반면 Basic Latin 이 있는 폰트를 미리 래스터하고 정적으로 사용하는 방법으로 Font Glyph Packer 의 설정이 가능하다.

위에서 언급한 바와 같이 일반적으로 가용한 경우 미리 래스터한 정적 텍스처를 사용하고 그렇지 않은 경우 가능하면 동적 캐시를 사용한다.

폰트 글리프 패커는 기본적으로는 사용이 불가능하다. 즉, Scaleform 는 패킹 매개변수를 명백히 생성하지 않고 로더에 설정하지 않은 경우 모든 내장 폰트에 대해 동적 캐시를 사용한다. 이는 다음과 같은 방식으로 실행할 수 있다.

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

그러나 GfXExport (그리고 해당 Scaleform 파일)를 사용할 때 글리프를 미리 래스터할 수 있다. 위의 호출은 단지 "로딩 시 글리프를 묶지 말 것"을 의미한다. 글리프를 GfXExport 로 미리 묶는 경우 정적 캐시로 사용된다.

정적 캐시의 사용이 바람직한 경우 (내장 글리프의 전체 개수가 작은) 다음과 같이 구성할 수 있다.

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

SetUseSeparateTextures()는 패킹을 제어한다. 만약 참인 경우 패커는 모든 폰트에 대해 별도의 텍스처를 사용한다. 기타의 경우 글리프를 최대한 단순히 포장한다. 별도의 텍스처를 사용하는 것은 텍스처 사이의 스위치 수를 줄일 수 있으며 이를 통해 draw primitive 의 수도 감소한다. 그러나 일반적으로 시스템 및 비디오에서 사용하는 메모리의 크기가 증가한다. 기본값으로 참이다.

SetGlyphCountLimit()는 패킹된 글리프의 최대 수치를 제어한다. 기본적으로 0 이며 이는 "제한 없음"을 의미한다. 폰트 내 내장 글리프의 계가 한계를 넘어가는 경우 폰트가 패킹되지 않는다. 아시아 언어를 라틴어 또는 기타 언어와 사용할 때 매개변수는 타당하다. 이 한계를 500 으로 설정하는 경우 대부분의 아시아 폰트는 동적으로 캐시가 이뤄지며 (동적 캐시가 가능한 경우) 소수의 글리프가 있는 폰트에 대해선 정적 텍스처를 사용한다.

SetTextureConfig()는 다음과 같은 텍스처 매개변수를 제어한다.

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

위의 값을 기본값으로 사용한다.

`NominalSize` - 텍스처 내 저장된 안티 앨리어싱된 글리프의 일반적인 크기를 픽셀로 표시함.

매개변수는 가장 큰 글리프가 텍스처에 얼마나 들어가는 지를 제어하며 대부분의 글리프는 이보다 작은 것으로 간주한다. 텍스처 RAM 사용과 큰 텍스트의 날카로운 정도 사이의 상관관계를 제어한다. "NominalSize"라고 한다는 점을 확인한다. 글리프가 글리프 슬롯에 연결된 동적 캐시와는 달리 정적 캐시는 실제로 경계가 있는 상자를 사용해서 다른 크기의 글리프를 포장한다. `NominalSize` 는 텍스트의 높이가 픽셀로 설정된 것과 같이 정확히 동일한 값을 갖는다. 또한 동적 캐시가 정적 캐시에 비해 "해상도 능력"에 있어 약간 더 좋음을 의미한다.

`PadPixels` - 개별 글리프 이미지 주변에 어느 정도의 공간이 있는가를 나타냄. 최소 1 이상이어야 한다. 값이 클수록 최소화 된 텍스트의 경계가 더 부드러워지지만 더 많은 텍스처 공간이 낭비된다.

`TextureWidth`, `TextureHeight` - 글리프가 포장되는 텍스처의 크기. 이 값은 2 의 승수로 이뤄져야 한다.

몇몇 사용상 시나리오와 그 의미가 아래에 있다.

모든 값을 기본으로 설정한다. 동적 캐시가 가용하고, `Font Glyph Packer` 를 사요하지 않으며 동적 캐시를 미리 처리된 `GFX` 파일로부터 로딩된 사전 래스터 글리프 텍스처를 제외한 모든 경우에 사용한다.

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

폰트 글리프 패커를 항상 사용한다. 동적 캐시의 사용이 불가능하다.

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

`Font Glyph Packer` 와 동적 캐시 모두 사용한다. 여기에서 500 개 미만의 내장 글리프가 있는 폰트는 미리 래스터 된 정적 텍스처를 사용하며 그 외의 경우 동적 캐시를 사용한다.

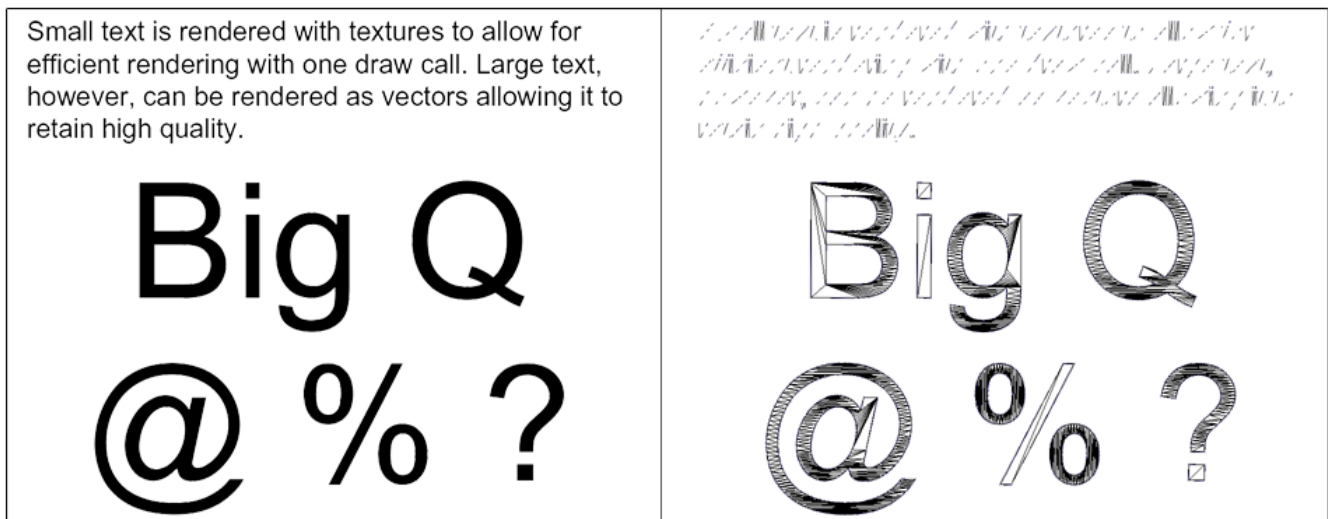
또한 이러한 기능 중 일부가 동적 캐시로만 가능하다는 점을 언급할 필요가 있다. 예를 들어 자동 픽셀 그리그 맞춤이 있는 가독성을 위해 최적화 된 텍스트 (자동 힌트라고 하는)는 동적 캐시로만 작동한다.

블러, 음영 및 글로우와 같은 효과 또한 동적 캐시로만 가능하다. 일반적으로 글리프 패커 및 정적 캐시는 성능이 낮은 절약형 시스템에 양호하다.

## 5.6 벡터화 제어

이 문서의 초반에서 다룬 것처럼, 채워진 벡터 도형은 텍스처 캐시에 맞지 않는 큰 글리프를 렌더링할 때 Scaleform 에서 사용된다. 플래시에서 개별 텍스트 문자의 크기에 제한이 없기 때문에 텍스처를 통해 렌더링에 필요한 메모리의 양은 특정 지점 이상을 넘지 못하게 된다. 이러한 문제를 해결하는 한 가지 방법은 글리프 비트맵 크기를 제한하고 해당 포인트로부터 2 진 필터링을 사용하는 것이다. 불행하게 이렇게 함으로써 렌더링 품질이 빠르게 저하된다.

크기가 큰 양질의 텍스트를 렌더링하기 위해 Scaleform 는 글리프 렌더링을 삼각형으로 변환할 수 있으며 그 후 에지 안티 앨리어싱 기술을 사용한다. 대부분의 경우 사용자가 변환을 인지하지 못하는데, 그 이유는 글리프의 모양이 텍스트 문자의 크기가 증가할 때 부드러운 모서리를 유지하고 있기 때문이다. 다음과 같은 그림은 텍스처와 삼각형 메시 렌더링 텍스트 사이의 차이를 나타낸다. Scaleform Player 에서 실행 가능한 CTRL + W 키는 와이어 프레임 모드를 토글하며 이를 통해 사용자가 렌더링이 어떻게 이뤄졌는지를 확인할 수 있다.



작은 텍스트는 텍스처로 렌더링되며 이는 한 번의 그리기 호출로 효과적인 렌더링을 위한 것이다. 그러나 큰 텍스트의 경우 벡터로 렌더링이 되면 높은 품질을 유지할 수 있다.

비록 삼각형으로 렌더링 된 글리프가 보기 좋지만 증가하는 삼각형 및 draw primitive 카운트의 증가로 인해 렌더링 간 더 많은 처리 시간을 필요로 한다. 동적 캐시를 사용할 때 최대 텍스처 글리프의

높이는 캐시의 MaxSlotHeight 값을 통해 결정되며 Render::GlyphCacheConfig::SetParams 를 통해 수정이 가능하다. 만약 표시될 글리프가 텍스처 슬롯에 맞는 경우 텍스처를 통해 렌더링이 이뤄지며 그렇지 않은 경우 벡터 렌더링을 사용한다. 사용하는 렌더링 기법이 글리프 당 결정되기 때문에 한 줄의 텍스트는 비트맵과 벡터 기호 모두를 포함할 수 있으며 이 때 픽셀 높이는 최대 캐시 슬롯의 높이에 근접한다. 하위 픽셀의 정확도로 인해 서로 다르게 렌더링 된 기호의 형태는 거의 구분되지 않는다.

정적 캐시는 다르게 작동한다. 정적 캐시에서 모든 글리프는 정의된 명목상, 그리고 tri-linear 필터를 사용해서 크기가 결정된 의 폰트 크기를 기준으로 미리 래스터된다. 명목상의 폰트 크기가 고정되기 때문에 텍스트 렌더링 기법의 선택은 명목상의 폰트 크기를 통해 이뤄지는 것이지 개별 글리프의 높이를 기준으로 이뤄지진 않는다.

GlyphCacheParams::MaxRasterScale 값을 변경하여 동적/정적 캐시에서 텍스처 렌더링에서 벡터 렌더링으로 전환할 위치를 마음대로 정할 수 있습니다.

SetMaxRasterScale 의 내용은 벡터로의 변환이 일어난 후 픽셀 단위로 최대 텍스처 슬롯 크기의 승수를 할당한다. MaxRasterScale 의 기본값은 1.25 이다. 1.25 가 의미하는 것은 텍스처에 저장된 명목상 글리프의 크기에 비해 화면상의 글리프 크기가 1.25 배 이상이 아닌 경우 플레이어는 텍스처를 사용하여 텍스트를 렌더링 한다는 것을 의미한다. 48 픽셀의 기본 명목상의 크기와 함께, 벡터 렌더링은 화면에서 60 픽셀 이상의 글리프에 사용된다.

만약 MaxRasterScale 값을 충분히 높게 잡으면 벡터 렌더링을 사용하지 않는다. 벡터 렌더링은 또한 -strip\_font\_shapes 옵션으로 gfxexport 툴의 실행을 통해 생성되는 GFX 파일을 불가능하게 한다.

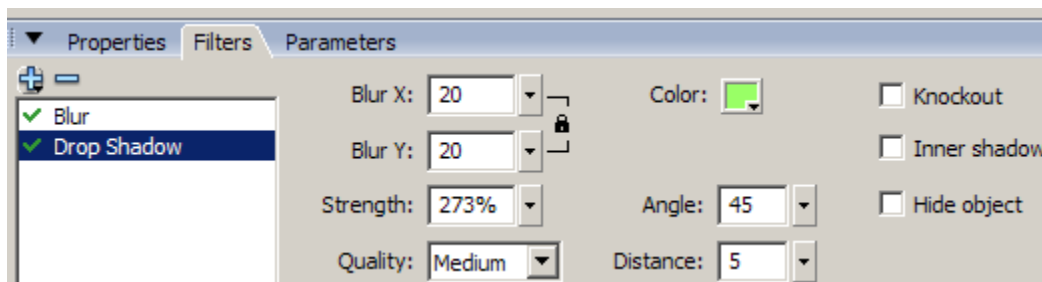
## 6 제 5 부: 텍스트 필터 효과 및 ActionScript 확장

동적 글리프 캐시가 가용할 때, Scaleform 2.x 는 Blur, Drop Shadow 및 Glow 필터 효과를 텍스트에 적용할 수 있다. 텍스트 필터 지원은 동적 캐시를 필요로 한다. 글리프 팩커 또는 정적 폰트 텍스처를 사용할 때 필터 효과를 지원하지 않는다. 현재 플레이어의 버전에서 필터는 텍스트 필터에 직접 적용될 때에만 작동하며 영화 클립에 적용될 때 효과는 없다.

Scaleform 텍스트 필터는 플래시와 유사하지만 다수의 차이점이 있다. 어도비 플래시에서 필터는 전체 텍스트 필드의 래스터 결과에 따라 순차적으로 적용된다. 이는 Blur, Drop Shadow 및 Glow 필터가 하나씩 적용될 수 있다는 점을 의미한다. 이 기법은 상당 수준의 유연성을 가지나 자원을 많이 소모한다. 플래시와 비교해서 Scaleform 에서의 필터 지원은 제한적이지만 매우 빠르다. 동적 수용 글리프 캐시와 필터는 일반 텍스트만큼이나 빨리 작동한다. 제한된 필터 지원에도 불구하고 Scaleform 는 소프트 셰도우 및 글로우 효과를 만드는 데 있어 상당히 좋은 기능을 제공한다.

### 6.1 필터 형태, 가용한 옵션 및 제약

플래시 스튜디오에서 필터는 한 번에 하나의 텍스트 필드에 추가될 수 있다.



플래시와 Scaleform 필터의 큰 차이점은 Scaleform 는 블러가 적용된 글리프의 캐시 비트맵 사본을 생성 및 저장하지만 플래시는 결과로 나오는 텍스트 필드에 필터를 적용한다.

Scaleform 는 Blur 와 Drop Shadow 필터만을 지원한다. Glow 필터는 사실 Drop Shadow 필터의 하위 집합이다. 플래시와 다르게 필터는 순차적으로 적용되지 않는다. 대신 두 개의 서로 다른 레이어로서 독립적으로 적용된다. Scaleform 에서 단 하나의 Drop Shadow 또는 Glow 필터 중 필터 목록에서

차후에 나타나는 필터를 고려한다. 이에 더해, 옵션 사항인 Blur 필터를 텍스트 자신에 적용할 수 있다. 일반적인 알고리즘은 다음과 같다.

#### 필터 목록을 통한 반복

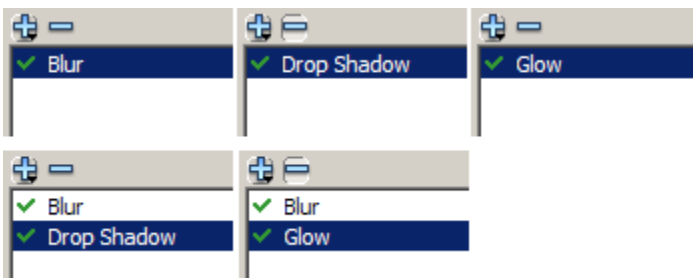
“Glow” 또는 “Drop Shadow”인 경우 세도우 필터 매개변수를 저장한다.

각 “Glow” 또는 “Drop Shadow”는 목록에서 이전에 나온 항목을 override 한다.

“Blur”인 경우 블러 필터 매개변수를 저장한다.

각 “Blur”는 이전에 목록에 나온 항목을 override 한다.

그 결과 필터의 결합은 다음과 같이 이뤄진다.

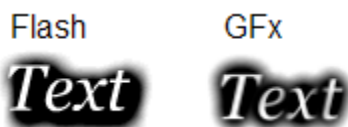


Drop Shadow 와 Glow 필터를 목록에 추가하면 나중에 추가한 항목만 고려된다.

Scaleform 에서의 렌더링은 또한 플래시에서와 다르다. 우선 “shadow” 또는 “glow” 레이어를 렌더링한다. 그 후 텍스트 자체는 원래 글리프에 적용된 “Blur” 필터로 shadow 위에 렌더링된다. 만약 “shadow” 또는 “glow” 필터가 “Knockout” 또는 “Hide Object” 플래그를 갖는 경우 텍스트 (두 번째 레이어)는 렌더링되지 않는다. “Inner Shadow”와 “Inner Glow”는 이 때 Scaleform 에 의해 지원되지 않는다.

알고리즘의 특성 때문에 Scaleform 내의 필터는 분명 제한사항을 갖는다. Blur X 와 Blur Y 의 최대값은 15.9 픽셀로 제한된다. Shadow 또는 Glow 값의 최대치는 1590%로 제한된다.

“Knockout” 옵션 또한 다르게 작동한다. 플래시에서 원래 이미지는 전체적으로 세도우로부터 추출되며 세도우 오프셋을 보호한다. Scaleform 에선 “글리프 별”로 이뤄지며 오프셋을 무시한다. 또한 세도우 또는 글로우 반지름이 큰 경우 글리프 이미지가 중첩되어 서로가 “어두워” 질 수 있다.









그 결과 "Knockout" 옵션은 기본적으로 필터의 지름이 작은 "Glow" 필터 (또는 0 의 오프셋을 갖는 "Drop Shadow"에 대해) 에서만 유용하다.

## 6.2 필터 품질

플래시는 이미지 블러를 위해 단순한 상자 필터링을 사용한다. "품질"은 필터가 실행한 패스의 횟수를 제어하는데 이는 결과 이미지에 큰 영향을 미친다. 예를 들어 낮은 품질로 된 3x3 블러 필터는 3x3 픽셀 영역의 평균값을 계산한다. 중간 품질은 동일한 필터를 두 번 적용한다. 그리고 고품질은 이 필터를 세 번 적용한다. 이는 품질이 필터의 눈에 가시적인 반지름에 큰 영향을 미친다는 것을 의미한다. 서로 다른 접근 방식으로 인해 비주얼의 결과가 다르나 비슷하다. 사실 단순한 하나의 패스 상자 필터는 너무 조악한 결과를 야기한다.

플래시와 다르게 Scaleform 는 가우시안 블러 필터와 스마트 순환 알고리즘을 사용하는데 이 속도는 필터의 지름에 의존하지 않는다. Scaleform 에선 낮음 또는 높음이라는 두 가지 품질만이 있으며 매우 유사한 비주얼 결과를 낳는다. 낮은 품질의 경우 "Quality:Low"를 필터 패널에 설정할 때만 사용된다. 그렇지 않으면 Scaleform 는 고품질 필터를 사용한다. 둘 사이의 차이점은 고품질 필터는 반지름을 분수값 (시그마)으로 사용할 수 있지만 저품질 필터의 경우 정수의 반지름을 사용한다. 더 많은 경우 Scaleform 는 글리프의 적절한 크기로 분수 반지름을 시뮬레이션하나 가독성을 위해 텍스트 필드를 안티 앨리어스하는 경우 저품질 세도우가 약간 불명확해 보일 수 있다. 일반적인 권장사항은 가독성을 위해 최적화 된 작은 텍스트에 대해 고품질을 사용하는 것이다.

	Low quality	Medium quality	High quality
Flash			
GfX			

보다시피 플래시에서 그 차이점이 분명하다. 그러나 Scaleform 에선 미묘하다. "저품질"만이 Scaleform 내 기타 항목과 다르다. "중간 품질" 및 "고품질"의 경우 동일한 결과를 만든다.



저품질 필터의 작동이 더 빠르나 글리프 캐시 시스템이 이러한 차이점을 상쇄한다. 그러나 성능이 낮은 시스템에선 저품질 필터의 사용이 훨씬 좋으며 특히 하드웨어에서 부동 소수점을 지원하지 않는 경우 더욱 그러하다.

고품질 필터는 부동 소수점 계산을 필요로 하며 여기에서 언급한 귀납적인 구현을 사용한다. 해당 내용은 <http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Smoothin.html> 에 있다.

### 6.3 필터 애니메이션

플래시 타임라인과 Scaleform ActionScript 확장을 사용해서 셰도우 효과를 보여줄 수 있다. 그러나 애니메이션 작업에 드는 비용을 이해해야 할 필요가 있다. 반지름, 강도 또는 품질을 변경할 때 Scaleform 는 글리프 이미지를 재생산해야 하며 이를 캐시에 저장해야 한다. 이로 인해 캐시 갱신을 더 자주하게 된다. 사실 위에 언급한 값을 바꾸는 것은 알파벳의 다양성을 증가시키는 것과 동일하다. 각 버전을 캐시에 저장해야 한다. 대신 색상 (알파 포함), 각도 및 거리의 변경은 성능에 영향을 주지 않는다. 예를 들어 셰도우 또는 글로우를 강하게 또는 약하게 하려는 경우 반지름과/또는 강도를 바꾸는 거 대신 색상의 반투명도 (알파)를 나타내는 것이 더 좋다.

## 6.4 *ActionScript* 에서 필터 사용하기

Scaleform 은 AS2 및 AS3 모두에 대해 동적/입력 텍스트 필드를 위한 표준 필터 클래스(예를 들어 DropShadowFilter, BlurFilter, ColorMatrixFilter, BevelFilter)를 지원합니다. 보다 상세한 내용은 Flash 설명서를 참조하십시오.