

Autodesk® Scaleform®

Scaleform 통합시스템 학습서

이 문서는 DirectX 9 를 통한 기본적인 Scaleform 사용법 및 3D 엔진 통합을 설명하고 있다.

집필: Ben Mowery
버전: 3.09
최종 편집: 2013년 10월 7일

Copyright Notice

Autodesk® Scaleform® 4.4

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

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

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

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

Disclaimer

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

연락처:

문서	Scaleform 4.3 통합시스템 학습서
주소	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
홈페이지	www.scaleform.com
이메일	info@scaleform.com
직통전화	(301) 446-3200
팩스	(301) 446-3199

목차

1	소개	1
2	문서 개요	2
3	설치와 빌드 의존성	3
3.1	설치	3
3.2	데모 컴파일	4
3.3	Scaleform Player에서 SWF 재생 벤치마크	5
3.4	샘플 베이스 컴파일	7
3.5	Scaleform 빌드 의존성	7
3.5.1	AS3_Global.h 파일과 Obj\AS3_Obj_Global.xxx 파일	9
4	게임 엔진 통합	10
4.1	플래시 렌더링	10
4.2	크기조절 모드	19
4.3	입력 이벤트 처리	21
4.3.1	마우스 이벤트	21
4.3.2	키보드 이벤트	23
4.3.3	히트 테스트	24
4.3.4	키보드 포커스	25
4.3.5	터치 이벤트	26
5	지역화와 폰트 소개	28
5.1	폰트개요 및 기능	28
6	C++, 플래시, 액션스크립트 연결하기	31
6.1	액션스크립트에서 C++로	31
6.1.1	FS Command 콜백	31
6.1.2	ExternalInterface	33
6.2	C++에서 액션스크립트로	36
6.2.1	액션스크립트 변수 조작	36

6.2.2 액션스크립트 서브루틴 실행	39
6.3 플래시 파일간 통신	40
7 텍스처에 렌더링하기	42
8 OpenGL 샘플	44
9 GFxExport로 전처리	44
10 다음 단계	46

1 소개

Scaleform 는 이미 검증된 고성능 비주얼 사용자 인터페이스 (UI) 디자인 미들웨어 솔루션으로서 개발자들이 플래시를 사용해서 빠르고 낮은 비용으로 최신 GPU 가속 애니메이션 UI 와 벡터 그래픽을 새로운 도구나 프로세스를 배우지 않고도 구현할 수 있도록 한다. Scaleform 는 플래시에서 게임 UI 까지 끊기지 않고 연계된 개발을 지원한다.

UI 뿐만 아니라, 개발자들은 Scaleform 를 사용하여 3D 표면에 애니메이션 텍스처를 맵핑함으로써 3D 환경 내의 플래시를 표현할 수 있게 되었다. 둘 3 게임 화면 또는 3D 오브젝트 상의 플래시 UI 를 생각해보라. 이와 유사하게, 3D 오브젝트와 비디오를 플래시 UI 내에서 표시할 수 있다. 그 결과, Scaleform 는 단일 UI 솔루션뿐만 아니라 기존의 첨단 게임 프레임워크를 향상시키는 방법으로 그 역할을 제대로 수행하고 있다.

이 튜토리얼은 Scaleform 의 설치 및 사용에 관한 절차를 다루게 될 것이다. 또한 플래시 기반의 UI 를 사용하는 DirectX ShadowVolume SDK 샘플도 다룰 것이다.

주의: Scaleform 은 이미 주요 게임 엔진과 통합되었다. Scaleform 는 최소한의 코딩으로 Scaleform 를 지원하는 게임과 직접 사용이 가능하다. 이 문서는 우선적으로 Scaleform 를 일반적인 게임 엔진과 통합하거나 Scaleform 의 기능에 대한 심도 깊은 기술상의 논의를 하려는 기술자들을 위해 작성되었다.

주: 이 튜토리얼에 있는 내용을 실행할 때 Scaleform 의 최신 버전을 사용한다. 이 튜토리얼은 Scaleform 버전 4 이상에 적용된다.

주: 이 학습서는 구형 비디오 카드와 호환되지 않는다. 이것은 본 학습서가 기반하고 있는 DirectX SDK ShadowVolume 코드 때문이다. 샘플코드가 작동중에 충돌되면 “Cannot create renderer” 에러 메시지가 뜬다. 문제에 대한 소스는 Scaleform Player 어플리케이션 성공적으로 작동하는지 체크해 보도록 하자.

2 문서 개요

가장 최근에 여러 언어로 작성된 Scaleform 문서는 Scaleform 개발자 센터 (<http://gameware.autodesk.com/scaleform/developer/>)에서 찾을 수 있다. 해당 사이트 접속을 위해 무료 회원 가입이 필요하다.

이 문서에 포함되는 내용은 다음과 같다.

- 웹 기반 Scaleform SDK 참고 문서.
- PDF 문서.
- 폰트 개요: 폰트와 텍스트 렌더링을 설명하고 국제화를 위한 아트 애셋과 Scaleform C++ API 설정에 관한 세부 사항 제공
- XML 개요: SCALEFORM에서 가능한 XML 지원 설명
- Scale9Grid: 크기조절이 가능한 윈도우, 패널 및 버튼 생성을 위한 Scale9Grid의 사용법 설명
- IME 설정: Scaleform의 IME 지원을 최종 사용자 어플리케이션에 어떻게 통합하고 플래시에서 일반적인 IME 텍스트 입력 윈도우 스타일을 어떻게 생성하는지 설명
- 액션스크립트 확장: Scaleform 액션스크립트 확장을 다룸

3 설치와 빌드 의존성

3.1 설치

가장 최신의 Windows 용 Scaleform 인스톨러를 다운 받는다. 그것을 실행하고 설치 경로와 옵션을 유지한다. 필요하다면 Scaleform 인스톨러가 DirectX 를 업데이트 할 것이다. Scaleform 는 C:\Program Files\Scaleform\GFx SDK 4.4 디렉토리에 설치된다.

레이아웃은 다음과 같다.

3rdParty

libjpeg이나 zlib 처럼 Scaleform 가 필요로하는 외부 라이브러리

Projects

위의 어플리케이션을 빌드하는 비주얼 스튜디오 프로젝트

Apps\Samples

Scaleform Player 및 다른 데모들에 대한 샘플 소스 코드

Apps\Tutorial

이 튜토리얼에 대한 소스 코드 및 프로젝트들

Bin

이미 만들어진 샘플 바이너리 및 샘플 플래시 콘텐츠들이 포함되어 있음.

FxPlayer: 단순한 플래시 텍스트 HUD

Samples: 다양한 플래시 FLA 소스 샘플로 버튼, 편집 상자, 키패드, 메뉴 및 스픈 카운터와 같은 내용을 포함

Win32: Scaleform Player 및 데모 어플리케이션을 위해 미리 컴파일 된 2 진 파일

AMP : AMP 툴의 실행을 위한 플래시 파일들

GFxExport: GFxExport 는 선행 프로세싱 툴로서 플래시 컨텐츠 로드 시간을 가속시켜준다. 7 장에서 자세히 다룬다

주: demo.sln 비주얼 스튜디오 프로젝트를 리빌드하면 Win32 디렉토리 내의 실행 파일이 대체된다.

Include

Scaleform 헤더

Lib

Scaleform 라이브러리

Resources

CLIK 컴포넌트와 툴

Doc

2장에서 설명된 PDF 문서

3.2 데모 컴파일

Scaleform 빌드를 위한 시스템이 제대로 구성되어 있는지를 확인하기 위해 우선 [시작 → 모든 프로그램 → Scaleform → GFx SDK 4.4 → Demo MSVC 9.0 Solutions → GFx 4.4 Demos.sln]에 있는 데모 솔루션을 빌드한다. .sln 파일이 비주얼 스튜디오에서 열리면 D3D9_Debug_Static 설정을 선택하고 Scaleform Player 프로젝트를 빌드한다

C:\Program Files\Scaleform\GFx SDK

4.4\Bin\Win32\Msvc90\GFxPlayer\GFxPlayer_D3D9_Debug_Static.exe 파일의 수정 일자를 확인하여 제대로 빌드 되었는지 확인한 후 프로그램을 실행한다.

해당 프로그램과 [시작 → 모든 프로그램 → Scaleform → GFx SDK 4.4 → Demo Examples 폴더]에 있는 기타 Scaleform Player D3Dx 어플리케이션은 하드웨어에서 가속화되는 SWF 플레이어다. 이러한 툴을 이용하여 Scaleform 플래시 재생을 테스트하고 벤치마킹 할 수 있다

3.3 Scaleform Player에서 SWF 재생 벤치마크

시작 → 모든 프로그램 → Scaleform → GFx SDK 4.4 → GFx Players → GFx Player D3D9. 프로그램을 실행한다 C:\Program Files\Scaleform\GFx SDK 4.4\Bin\Data\AS2\Samples\SWFToTexture 폴더를 연 후 3D Windows.swf 를 어플리케이션으로 드래그한다.



그림 1: 샘플 플래시 컨텐츠의 하드웨어 가속 재생

F1 을 눌러 도움말을 본 다음에 다음과 같은 옵션을 테스트 해보자.

1. 성능 측정을 위해 CTRL+F 을 누른다. 현재 FPS 가 제목 표시줄에 나타난다.
2. 와이어프레임 모드를 토글하기 위해 CTRL+W 를 누른다. 최적화 된 하드웨어 재생을 위해 플래시 컨텐츠가 삼각형으로 어떻게 변환되는지를 확인한다. 와이어프레임 모드 종료는 다시 CTRL+W 를 누르면 된다.
3. 화면 확대를 하려면 CTRL 키를 누른 채로 마우스 왼쪽을 클릭한 후 마우스를 위아래로 움직인다.
4. 이미지를 좌우로 움직이려면 CTRL 키를 누른 채로 마우스 오른쪽을 클릭한 후 마우스를 이동시킨다.
5. 정상보기로 돌아오려면 CTRL+Z 를 누른다.

6. 전체화면 모드 전환은 CTRL+U 를 누른다.
7. 메모리 사용 등을 포함한 무비의 통계값을 보려면 F2 를 누른다.

버튼 모서리처럼 곡선으로 된 면을 크게 확대해서 봄도 깔끔하다는 사실에 주목하자. 창의 크기를 크게 또는 작게 하면 플래시 컨텐츠의 크기도 변한다. 벡터 그래픽의 장점 중 하나는 어떠한 해상도로도 컨텐츠 크기를 조절할 수 있다는 점이다. 전통적인 비트맵 그래픽은 800x600 비트맵 셋과 1600x1200 화면용 한 세트가 필요하다.

스크린의 오른쪽 센터에 Scaleform 로고를 볼 수 있는 것처럼 Scaleform 는 전통적인 비트맵 그래픽도 지원한다. 플래시에서 만들 수 있는 어떤 컨텐츠도 Scaleform 로 렌더링 가능하다

"3D Scaleform Logo" 라디오 버튼을 확대하고 CTRL+W 키를 눌러 와이어프레임 모드로 전환한다. 원이 삼각형으로 분할되어 있다는 것에 주목하라. CTRL+A 키를 몇 번 누르면 안티 앤티 엘리어싱 모드로 전환한다. Edge Anti-Aliasing 모드 (EdgeAA)에서, 추가로 서브픽셀 삼각형을 원 모서리에 추가하여 안티 엘리어싱 효과를 생성한다. 이는 비디오 카드가 AA 의 실행을 위해 네 배의 프레임 버퍼 비디오 메모리와 네 배의 픽셀 렌더링을 더 필요로 하는 전체 화면 안티 엘리어싱 (FSAA)보다 더 효과적이다. 스케일폼의 고유 기술인 EdgeAA 기술은 안티 엘리어싱을 일반적으로 굽은 모서리 및 대형 텍스트와 같이 이점을 가장 많이 받을 수 있는 영역에만 안티 엘리어싱을 적용하기 위해 객체의 벡터 특성을 이용한다. 삼각형의 수가 늘어나더라도 draw primitive (DP)의 수는 일정하게 유지되기 때문에 성능관리가 가능하다. 이것은 비디오 카드의 안티 엘리어싱 기능을 사용하는 것보다 일반적으로 볼 때 더 효율적이며 EdgeAA 를 비롯한 기타 품질설정을 사용하거나 불가하는 등의 조작이 가능하다.

Scaleform Player 툴은 플래시 컨텐츠의 디버깅과 성능 벤치마킹에 유용하다. 작업 관리자를 연 후 CPU 사용을 확인한다. CPU 사용은 Scaleform 가 벤치마킹 목적으로 초당 가능한 많은 수의 프레임을 렌더링하기 때문에 높게 나올 수 있다. CTRL+Y를 누르면 프레임 레이트를 화면 갱신주기(일반적으로 초당 60 프레임)로 고정한다. CPU 사용량이 급격하게 감소함을 확인할 수 있다.

주: 자신의 어플리케이션을 벤치마킹할 때는 Scaleform 디버그 빌드가 최적 성능을 제공하지 않기 때문에 릴리즈 빌드를 실행할 것.

3.4 샘플 베이스 컴파일

Scaleform 4.4 사용자는 Scaleform\GFX SDK 4.4\Apps\Tutorial 경로에 있는 Tutorial Solution 을 연다. 윈도우 시작 메뉴 내 Scaleform → GFx SDK 4.4 → Tutorial 에서 비주얼 스튜디오 2005 및 2008 용 솔루션을 확인할 수 있다. 프로젝트 설정이 "Debug"로 되어 있음을 확인한 후 어플리케이션을 실행한다



그림 2: 기본 UI 가 있는 ShadowVolume 어플리케이션

3.5 Scaleform 빌드 의존성

새로운 Scaleform 프로젝트를 생성할 때, 컴파일 실시 전 비주얼 스튜디오에서 설정할 몇 가지 사항이 있다. 튜토리얼에는 아래의 단계를 따를 때 참고로 볼 수 있는 상대 경로를 이미 갖고 있다. \$(GFXSDK)는 기본 SDK 설치 디렉토리에서 정의된 환경 변수임을 주지한다. 기본 위치는 C:\Program Files\Scaleform\GFX SDK 4.4\이다. 만약 라이브러리가 다른 위치에 있다면 시스템 상에 있는 라이브러리의 위치 지정을 위해 \$(GFXSDK)를 포함하는 경로로 바꿀 필요가 있다. 이 예에선 "Msvc90"을 사용한다. 비주얼 스튜디오 2008 또는 2010 사용자인 경우 각각 Msvc90 또는 Msvc10 을 사용할 필요가 있다.

Scaleform 를 프로젝트의 디버그 및 릴리즈 빌드 설정에서 include 경로에 추가한다.

\$(GFXSDK)\Src

\$(GFXSDK)\Include

다음의 내용을 비주얼 스튜디오 “Additional Include Directories” 항목에 붙여 넣는다.

컨비니언스 헤더(GFx.h)가 포함되어 있지 않을 경우, AS3 를 사용하고 있다면 필수 AS3 클래스 등록 파일 “GFx/AS3/AS3_Global.h”를 응용 프로그램에 직접 설치하십시오. 개발자가 이 파일에서 직접 불필요한 AS3 클래스를 제외할 수 있습니다. 자세한 내용은 [Scaleform LITE 사용자 정의 문서](#)를 참조하십시오.

The following library directories should be added to the “Additional Library Directories” field for all build configurations:

```
"$(DXSDK_DIR)\Lib\x86";  
"$(GFXSDK)\3rdParty\expat-2.1.0\lib\$(PlatformName)\Msvc90\Release";  
"$(GFXSDK)\3rdParty\pcre\Lib\$(PlatformName)\Msvc90\Release";  
"$(GFXSDK)\3rdParty\zlib-1.2.7\Lib\$(PlatformName)\Msvc90\Release";  
"$(GFXSDK)\3rdParty\libpng-1.5.13\Lib\$(PlatformName)\Msvc90\Release";  
"$(GFXSDK)\3rdParty\curl-7.29.0\lib\$(PlatformName)\Msvc90\Release"  
"$(GFXSDK)\3rdParty\jpeg-8d\Lib\$(PlatformName)\Msvc90\Debug;"  
"$(GFXSDK)\Lib\$(PlatformName)\Msvc90\$(ConfigurationName)
```

주: Msvc90 은 설치된 비주얼 스튜디오 버전에 적합한 문자열로 변형시킨다.

마지막으로 Scaleform 라이브러리와 의존성을 추가한다.

libgfx.lib

libgfx_as2.lib

libgfx_as3.lib

libgfx_air.lib

libgfonexpat.lib

libjpeg.lib

zlib.lib

libcurl.lib

libpng.lib

libgfrxrender_d3d9.lib

libgfsound_fmod.lib

디버그 및 릴리즈 설정 모두에서 샘플 어플리케이션을 컴파일하고 링크한다는 점을 기억한다. 참고를 위해, Scaleform include로 수정된 vcproj 파일과 링커 설정이 Tutorial/Section3.5 폴더에 나와 있다.

3.5.1 AS3_Global.h 파일과 Obj\AS3_Obj_global.xxx 파일

AS3_Global.h 와 Obj\AS3_Obj_global.xxx 가 서로 연관성이 전혀 없는 파일이라는 점에 주목하십시오.

AS3_Obj_Global.xxx 파일은 "글로벌" ActionScript 3 객체라고도 하는 객체의 구현을 포함합니다. 모든 swf 파일은 최소한 하나 이상의 "스크립트" 객체(글로벌 객체)를 포함합니다. C++로 구현된 모든 클래스를 위한 글로벌 객체인 GlobalObjectCPP 클래스도 있습니다. 이는 Scaleform VM 구현에만 해당합니다.

AS3_Global.h 는 전혀 다른 목적으로 사용합니다. 이 파일은 ClassRegistrationTable 배열을 포함합니다. 이 배열은 해당 AS3 클래스를 구현하는 C++ 클래스를 참조할 목적으로 사용합니다. 이 참조가 없으면 링커가 코드를 제외시킬 것 입니다. 그러므로 실행 파일 내에서 반드시 ClassRegistrationTable 을 정의해야 합니다. 그렇지 않으면 링커 에러가 발생할 것입니다. 이런 이유로 각 데모 플레이어는 AS3_Global.h 를 포함합니다.

사용자가 원하는 대로 ClassRegistrationTable 을 변경하려면(코드 크기를 줄이려면) ClassRegistrationTable 을 포함한 include 파일을 포함해야 합니다. 가장 좋은 방법은 AS3_Global.h 의 복사본을 만들고 필요 없는 클래스를 주석으로 처리하여 제외한 다음 응용프로그램에 포함시키는 것입니다. 이렇게 제외된 클래스는 사용되지 않습니다.

그렇지만 이 최적화 방법과 관련하여 한가지 알아야 할 것이 있습니다. AS3 VM에서 이름 분석은 런타임에 실행되므로 주석으로 처리되어 테이블에서 제외된 클래스가 필요한 경우 이를 찾지 못할 수 있습니다. 따라서 "불필요한" 클래스를 삭제해도 응용프로그램이 제대로 작동하는지 미리 확인하십시오.

4 게임 엔진 통합

Scaleform은 Unreal® Engine 3, Gamebryo™, Bigworld®, Hero Engine™, Touchdown Jupiter Engine™, CryENGINE™ 및 Trinigy Vision Engine™와 같은 대부분의 주요 3D 게임 엔진을 위한 통합 레이어를 제공한다. 이러한 엔진으로 개발된 게임에서 Scaleform 사용을 위해 필요한 코딩은 완전히 없거나 있더라도 해도 거의 없다.

여기에는 일반적인 DirectX 어플리케이션을 Scaleform과 어떻게 통합하는지를 다룬다. DirectX ShadowVolume SDK 샘플은 표준 DirectX 어플리케이션이며 일반적인 게임과 유사한 어플리케이션 및 게임 루프를 갖고 있다. 앞서 살펴본 바와 같이 이 어플리케이션은 DXUT 기반의 2D UI 오버레이로 된 환경에서 3D 환경을 렌더링한다.

이 튜토리얼은 기본 DXUT 사용자 인터페이스를 플래시 기반의 Scaleform 인터페이스로 대체하기 위해 Scaleform를 어플리케이션으로 통합하는 과정에 대해서 다룰 것이다.

DXUT는 어플리케이션에 기존의 렌더 루프를 노출시키지 않는 대신 낮은 수준의 세부사항에 콜백을 제공한다. 이러한 단계가 표준 Win32 DirectX 렌더 루프와 어떤 관계가 있는지를 이해하기 위해 Scaleform SDK와 함께 제공되는 GFxPlayerTiny.cpp 샘플과 비교해보자.

4.1 플래시 렌더링

통합 과정 중 첫번째 단계는 3D 백그라운드 상단에 플래시 애니메이션을 렌더링하는 것이다. 이를 위해 어플리케이션에 대해 전역적으로 사용되는 모든 플래시 컨텐츠의 로딩을 관리하는 [GFx::Loader](#) 객체 인스턴스와 무비의 단일재생 인스턴스를 나타내기 위한 [GFx::MovieDef](#)가 필요하다. 추가적으로 [Render::Renderer2D](#) 객체와 [Render::HAL](#) 객체를 인스턴스화 하여 Scaleform과 구현별 렌더링 API(여기에는 DirectX) 사이에서 인터페이스의 역할을 맡는다. 또한 어떻게 자원을 해제하고, 어떻게 로스트 디바이스(lost device) 이벤트에 응답할지, 그리고 어떻게 전체화면/창모드 전환을 처리할 것인지에 대하여 다룰 것이다.

변경 사항을 반영하여 수정된 ShadowVolume 버전은 Tutorial\Section4.1에서 찾아볼 수 있다. 이 문서에서 보여주는 코드는 예시용이지 완성된 코드가 아니다.

1 단계: 헤더 파일 추가

필요한 헤더 파일을 ShadowVolume.cpp 에 추가한다.

```
#include "GFx_Kernel.h"
#include "GFx.h"
#include "GFx_Renderer_D3D9.h"
```

비디오 렌더링을 위해 몇 가지 Scaleform 객체가 필요하며 GFxTutorial 어플리케이션에 추가되는 새로운 클래스 내에 함께 위치하게 된다. 하나의 클래스에 Scaleform 상태를 함께 넣어두면 코드를 더 깔끔하게 만들 뿐만 아니라 삭제호출 한방으로 모든 Scaleform 객체를 날려 버릴 수 있다는 장점이 있다. 이러한 객체 사이의 상호 작용은 밑에서 상세히 다룰 것이다.

```
// One GFx::Loader per application
Loader gfxLoader;

// One GFx::MovieDef per SWF/GFx file
Ptr<MovieDef> pUIMovieDef;

// One GFx::Movie per playing instance of movie
Ptr<Movie> pUIMovie;

// Renderer data
Ptr<Render::D3D9::HAL> pRenderHAL;
Ptr<Render::Renderer2D> pRenderer;
MovieDisplayHandle hMovieDisplay;
```

2 단계: GFx::System 초기화

Scaleform 초기화의 첫번째 단계는 Scaleform 메모리 할당을 관리할 [GFx::System](#) 오브젝트를 할당하는 것이다. WinMain 에 다음 코드를 추가한다.

```
// One GFx::System per application
GFx::System gfxInit;
```

GFx::System 오브젝트는 반드시 첫번째 Scaleform 호출 전에 스코프에 들어와있어야 하며 Scaleform 를 사용한 어플리케이션 종료 전까지 스코프를 떠나서는 안 된다. 이것은 WInMain 안에 있기 때문이다. 여기서 초기화된 GFx::System 은 Scaleform 기본 메모리 할당자를 사용한다. 그러나 어플리케이션의 커스텀 메모리 할당자로 오버라이드 가능하다. 이 학습서의 목적상 간단하게 GFx::System 을 초기화하면 되기 때문에 여기서는 이 정도로 그치도록 하자.

GFx::System 는 반드시 어플리케이션이 종결되기 전에 유효범위에서 벗어나야 한다. 따라서, 반드시 전역변수일 필요는 없다. 이 경우 GFxTutorial 이 해제되기 전에 스코프를 벗어날 것이다.

각 어플리케이션의 구조에 따라 달라지겠지만, GFx::System::Init()과 GFx::System::Destroy() 정적함수를 호출하는 것이 GFx::System 객체 인스턴스를 만드는 것보다는 쉬울 것이다.

3 단계: 로더 및 렌더러 생성

나머지 Scaleform 초기화는 InitApp() 내에서 어플리케이션의 WinMain이 자체적으로 초기화 된 직후 일어난다. 호출 후 아래와 같은 코드를 InitApp()에 추가한다

```
gfx = new GFxTutorial();
assert(gfx != NULL);
if(!gfx->InitGFx())
    assert(0);
```

GFxTutorial 은 [GFx::Loader](#) 객체를 포함하고 있다. 일반적으로 어플리케이션은 하나의 GFx::Loader 객체를 갖고 있으며 이 객체는 SWF/GFx 컨텐츠의 로딩과 해당 컨텐츠를 자원 라이브러리에 저장하는 역할을 하며 이를 통해 향후 참조시에 자원을 재사용할 수 있게 된다. 별도의 SWF/GFx 파일은 이미지와 폰트 같은 자원을 공유함으로써 메모리를 절약할 수 있다. GFx::Loader 는 또한 디버그 로깅에 사용되는 [GFx::Log](#) 와 같은 일련의 설정 상태를 유지한다

GFxTutorial::InitGFx()에서 첫 번째 단계는 GFx::Loader 에 대한 상태를 설정하는 것이다. GFx::Loader 는 [SetLog](#) 가 제공하는 핸들러로 디버그 추적정보를 전달한다. 다수의 Scaleform 함수가 오류가 발생한 이유를 로그에 출력하기 때문에 디버깅에 매우 유용하다. 이번에는 기본 Scaleform PlayerLog 핸들러를 사용하는데, 이 핸들러는 메시지를 콘솔창에 출력한다. 하지만 게임 엔진의 디버그 로깅 시스템과의 통합은 GFx::Log 를 상속받은 클래스로 구현해서 만들 수 있을 것이다.

```
// Initialize logging -- Scaleform will print errors to the log
// stream.
gfxLoader->SetLog(Ptr<Log>(*new Log()));
```

GFx::Loader는 [GFx::FileOpener](#) 클래스를 통해 컨텐츠를 읽는다. 기본 구현에선 디스크로부터 파일을 읽지만 메모리 또는 자원 파일로부터의 사용자 로딩은 GFx::FileOpener를 상속받으면 가능하다.

```
// Give the loader the default file opener
```

```
Ptr<FileOpener> pfileOpener = *new FileOpener;
gfxLoader->SetFileOpener(pfileOpener);
```

Render::HAL는 일반 인터페이스인데 Scaleform가 다양한 하드웨어로 그래픽을 출력할 수 있도록 해준다. D3D9 렌더러의 인스턴스를 생성하여 로더와 결합한다. 렌더러 객체는 D3D 장치, 텍스처 및 Scaleform가 사용하는 정점 버퍼의 관리를 담당한다. 향후에는 HAL::InitHAL에서 Render HAL를 게임이 초기화시킨 IDirect3DDevice9 포인터로 전달하면 Scaleform가 DX9 자원을 생성하여 UI 컨텐츠를 성공적으로 렌더링 할 수 있게 된다.

```
pRenderHAL = *new Render::D3D9::HAL();
if (!(pRenderer = *new Render::Renderer2D(pRenderHAL.GetPtr())))
    return false;
```

위의 코드는 Scaleform가 제공하는 기본 Render::D3D9::HAL 객체를 사용한다. Render::HAL를 상속받으면 Scaleform 렌더링 속성에 대해 향상된 제어가 가능하며 더 강력한 통합을 할 수 있다.

4 단계: 플래시 무비 로딩

이제 GFx::Loader는 무비를 로딩할 준비가 되었다. 로딩된 무비는 [GFx::MovieDef](#) 객체로 표시된다. GFx::MovieDef는 기하정보 및 텍스처처럼 무비에 대한 모든 공유 데이터를 포함한다. 그러나 개별 버튼의 상태, 액션스크립트 변수 또는 현재 무비 프레임과 같은 인스턴스별 정보는 포함하지 않는다

```
// Load the movie
pUIMovieDef = *gfxLoader.CreateMovie(UIMOVIE_FILENAME,
                                      Loader::LoadKeepBindData |
                                      Loader::LoadWaitFrame1, 0);
```

LoadKeepBindData 플래그는 시스템 메모리 내에 텍스터 이미지의 사본을 유지하기 때문에 어플리케이션이 D3D 장치를 재생성 할 경우 유용할 수 있다. 이 플래그는 게임 콘솔 시스템이나 텍스처라 로스트 되지 않는다면 필요 없다.

LoadWaitFrame1 명령은 [CreateMovie](#) 가 무비의 최초 프레임을 로딩할 때까지 반환되지 않도록 한다. 이는 GFx::ThreadedTaskManager를 사용하는 경우 상당히 중요하다.

메모리 아레나(Arena)를 사용하는데 마지막 인자를 선택적으로 정의한다. 메모리 아레나의 생성 및 사용에 대한 정보는 [Memory System Overview](#) 문서를 참조하기 바란다.

5 단계: 무비 인스턴스 생성

무비를 렌더링하기 전 GFx::MovieDef 객체를 통해 [GFx::Movie](#) 인스턴스를 생성해야 한다. GFx::Movie는 현재 프레임, 무비 내 시간, 버튼 상태 및 액션스크립트 변수와 같은 무비의 단일재생 인스턴스와 관련된 상태를 유지한다.

```
pUIMovie = *pUIMovieDef->CreateInstance(true, 0, NULL);  
assert(pUIMovie.GetPtr() != NULL);
```

[CreateInstance](#)의 전달인자는 첫 번째 프레임을 초기화하는지 여부를 결정한다. 만약 false인 경우 액션스크립트의 최초 프레임 초기화 코드를 실행하기 전 플래시 및 액션스크립트를 변경할 수 있는 기회를 갖게 된다. 메모리 아레나(Arena)를 사용하는데 마지막 인자를 선택적으로 정의한다. 메모리 아레나의 생성 및 사용에 대한 정보는 [Memory System Overview](#) 문서를 참조하기 바란다.

무비 인스턴스 생성 후, 첫 번째 프레임은 Advance()를 호출함으로써 초기화된다. 이는 false이 CreateInstance로 전달된 경우에만 필요하다.

```
// Advance the movie to the first frame  
pUIMovie->Advance(0.0f, 0, true);  
  
// Note the time to determine the amount of time elapsed between  
// this frame and the next  
MovieLastTime = timeGetTime();
```

[Advance](#)의 첫 번째 인자는 무비의 마지막 프레임과 해당 프레임 사이의 시간 차를 초로 표시하는 것이다. 현재 시스템 시간을 기록하여 해당 프레임과 다음 프레임 사이의 시간 차이를 계산할 수 있게 된다.

3D 씬 위에 무비를 알파 블랜드 하려면

```
pUIMovie->SetBackgroundAlpha(0.0f);
```

위의 호출이 없으면, 무비는 렌더링 되겠지만 플래시 파일에서 정의한 배경색으로 3D 환경을 덮어 쓰게 된다.

6 단계: 디바이스 초기화

렌더를 위한 DirectX 디바이스와 프리젠테이션 매개변수들이 Scaleform에 전달 되어야 한다. 이 값들은 Render::D3D::HALInitParams 구조체 안에서 래핑되고,

Render::D3D9::HAL::InitHAL 함수에 전달되며, D3D 디바이스가 생성되기 이전, Scaleform 가 렌더를 요청하기 이전에 호출된다. D3D 디바이스 핸들이 변경되면 InitHAL 를 호출해야하며 이는 윈도우 크기조절 또는 전체 화면/창모드 간 변경 시에 발생할 수 있다.

ShadowVolume 의 OnResetDevice 함수는 DXUT 에 의해서 초기 디바이스 생성시, 그리고 디바이스 리셋 후에 호출된다. 다음 코드가 GFxTutorial 내의 OnResetDevice 메소드에 추가되었다.

```
pRenderHAL->InitHAL(  
    Render::D3D9::HALInitParams(pd3dDevice, presentParams,  
        Render::D3D9::HALConfig_NoSceneCalls));
```

InitHAL 호출은 Scaleform 에 D3D 장치 정보를 전달한다.
HAL_NoSceneCalls 플래그는 DirectX BeginScene()와 EndScene() 호출이 Scaleform 를 통해 이뤄지지 않는다는 것이다. 이는 ShadowVolume 샘플이 OnFrameRenderer 콜백에서 이들 호출을 이미 하고 있기 때문에 필요한 것이다.

7 단계: 로스트 디바이스

윈도우 크기를 조절하거나 어플리케이션을 전체 화면으로 전환한 경우, D3D 디바이스는 로스트 될 것이다. 정점버퍼 및 텍스처를 포함한 모든 D3D 서비스를 다시 초기화해야 한다. ShadowVolume 은 OnLostDevice 콜백에서 서비스를 해제한다. Render::HAL 에 로스트 디바이스에 관한 내용을 알려주면 GFxTutorial 내 해당 OnLostDevice 에서 D3D 자원을 해제할 기회를 주게 된다.

```
pRendererHAL->ShutdownHAL();
```

이 단계와 이 전 단계는 DXUT 프레임워크의 콜백 시스템을 기반으로 하는 초기화와 로스트 디바이스를 설명한 것이다. 기본 Win32/DirectX 렌더루프에 대한 예제는 Scaleform SDK 에서 제공하는 GFxPlayerTiny.cpp 파일을 참고하자.

8 단계: 자원 할당 및 청소(cleanup)

모든 Scaleform 객체가 GFxTutorial 객체 내에 포함되어 있기 때문에 청소는 WinMain 의 마지막에서 GFxTutorial 객체를 삭제하는 것만큼 간단하다.

```
delete gfx;  
gfx = NULL;
```

기타 고려할 사항으로는 정점 버퍼와 같은 DirectX 9 자원의 청소다. Scaleform 를 통해 이뤄지지만 주 메인 게임 루프 중에 발생하는 할당 및 청소에 대해서 InitHAL()와 ShutdownHAL()가 어떤 역할을 하는 지 이해하는 것이 중요하다.

DirectX9 구현에서 InitHAL 는 정점 버퍼를 포함한 D3DPOOL_DEFAULT 자원을 할당한다. 엔진과 통합할 때는 해당 호출을 D3DPOOL_DEFAULT 관련 자원 할당하는 곳에서 InitHAL 를 호출하도록 한다.

ShutdownHAL 는 D3DPOOL_DEFAULT 자원을 해제한다. ShadowVolume 을 포함해서 DXUT 프레임워크를 사용하는 어플리케이션은 DXUT OnResetDevice 콜백에서 D3DPOOL_DEFAULT 자원을 할당해야 하며 OnLostDevice 콜백에서 해제 해야한다. GFxTutorial::OnLostDevice 내의 ShutdownHAL 호출은 GFxTutorial::OnResetDevice 내의 InitHAL 호출과 짹을 이룬다.

엔진과 통합하는 경우 엔진 D3DPOOL_DEFAULT 자원을 생성하거나 해제하려면 기타 호출과 함께 InitHAL 와 ShutdownHAL 를 호출한다.

9 단계: 뷰포트 설정

무비는 스크린 상에 렌더링 되기 위한 특정 뷰포트가 반드시 필요하다. 이번 경우에는 전체 윈도우를 점유하고 있다. 화면 해상도가 변경가능 하기 때문에 D3D 디바이스가 리셋될때마다 뷰포트도 함께 리셋되도록 GFxTutorial::OnResetDevice 에 다음 코드를 추가한다.

```
// Use the window client rect size as the viewport.  
RECT windowRect = DXUTGetWindowClientRect();  
DWORD windowHeight = windowRect.right - windowRect.left;  
DWORD windowWidth = windowRect.bottom - windowRect.top;  
pUIMovie->SetViewport(windowWidth, windowHeight, 0, 0,  
                        windowWidth, windowHeight);
```

[SetViewport](#)의 처음 두 개의 매개변수는 사용된 프레임 버퍼의 크기 (일반적으로 PC 어플리케이션용 윈도우 크기)를 정의한다. 다음에 나오는 네 개의 매개변수는 Scaleform가 렌더링되는 프레임 버퍼 내의 뷰포트 크기를 정의한다.

프레임버퍼 크기 전달인자는 OpenGL과의 호환성, 다른 좌표계 시스템을 사용하는 시스템, 프레임버퍼 크기를 알 수 없을 때 등의 이유로 필요하다.

Scaleform는 플래시 컨텐츠의 크기를 결정하고 뷔포트 내에 어떻게 위치시킬 것인지에 관한 함수를 제공한다. 어플리케이션의 실행 준비를 마친 후 4.2에서 이러한 옵션에 관해 다룰 것이다.

10 단계: DirectX 씬으로 렌더링

ShadowVolume 의 OnFrameRender() 함수에서 렌더링이 이뤄진다. 모든 D3D 렌더링 호출은 BeginScene()과 EndScene() 호출 사이에서 이뤄진다. 여기에선 EndScene() 이전에 GFxTutorial::AdvanceAndRender()를 호출한다.

```
void AdvanceAndRender(void)
{
    DWORD mtime = timeGetTime();
    float deltaTime = ((float)(mtime - MovieLastTime)) / 1000.0f;
    MovieLastTime = mtime;

    pUIMovie->Advance(deltaTime, 0);
    pRenderer->BeginFrame();

    if (hMovieDisplay.NextCapture(pRenderer->GetContextNotify()))
    {
        pRenderer->Display(hMovieDisplay);
    }

    pRenderer->EndFrame();
}
```

Advance 는 deltaTime 초만큼 무비를 재생한다. 무비의 재생 속도는 현재 시스템 시간을 기준으로 어플리케이션을 통해 제어된다. 서로 다른 하드웨어 설정에서 무비를 제대로 재생하는지를 확인하기 위해 실제 시스템 시간을 GFx::Movie::Advance 에 제공하는 것이 중요하다.

11 단계: 렌더링 상태 보호

[Render::Renderer2D::Display](#)는 D3D 디바이스에서 무비 프레임을 렌더링하기 위한 DirectX 호출을 실행한다. 성능상의 이유로 블렌딩 모드와 텍스처 저장 설정은 저장되지 않는다. 따라서, D3D 디바이스 상태는 Render::Renderer2D::Display 호출 후에 달라질 수 있다. 이로 인해 몇몇 어플리케이션은 악영향을 받을 수 있다. 가장 직접적인 해결책은 Display 호출 이전에 디바이스 상태를 저장하고 나중에 이를 복구하는 것이다. Scaleform 렌더링 후 필요한 상태로 게임 엔진을 재초기화 함으로써 더 나은 성능을 얻을 수 있다. 이 학습서에서는 DX9의 상태 블록 함수를 사용해서 간단하게 상태를 저장하고 복구한다.

DX9 상태 블록은 어플리케이션이 작동중인 동안만 할당되며 GFxTutorial::AdvanceAndRender() 호출 앞 뒤에 사용된다.

```
// Save DirectX state before calling Scaleform  
g_pStateBlock->Capture();  
// Render the frame and advance the time counter  
gfx->AdvanceAndRender();  
// Restore DirectX state to avoid disturbing game render state  
g_pStateBlock->Apply();
```

12 단계: 기본 UI 사용하기

마지막 단계는 DXUT 기반의 UI 사용을 막는 것이다. 이는 코드의 관련 블록을 커멘트 처리함으로써 이뤄지며 최종 결과는 Section4.1\ShadowVolume.cpp에 나와 있다. 차이점을 보기 위해 이전 코드와 비교한다. DXUT 와 관련된 모든 변경을 확인하자.

```
// Disable default UI
```

```
..
```

이제 DirectX 어플리케이션에서 하드웨어 가속되는 플래시 무비를 렌더링하게 된 것이다.



그림 3: Scaleform 플래시 기반의 UI가 있는 ShadowVolume 어플리케이션

4.2 크기조절 모드

GFx::Movie::SetViewport호출은 뷰포트의 크기를 화면 해상도와 동일하게 유지시킨다. 만약 화면 비율이 플래시 컨텐츠의 원래 화면 비율과 다를 경우 인터페이스가 왜곡될 수 있다. 따라서, Scaleform는 다음 함수를 제공한다.

- 컨텐츠의 화면 비율을 유지하거나 자유롭게 크기를 변화시킬 수 있도록 함
- 뷰포트의 중앙, 모서리 또는 측면에 기준한 위치 지정

이러한 함수는 4:3과 와이드스크린 화면 모두에서 컨텐츠를 렌더링하는 데 있어 상당히 유용한다. Scaleform의 장점 중 하나는 확대 가능한 벡터 그래픽을 이용해서 어떠한 화면 해상도라도 자유롭게 변환이 가능하다는 점이다. 기존의 비트맵 그래픽에선 일반적으로 작업자들이 서로 다른 화면 해상도에 따라 다른 크기의 비트맵을 작성해야 한다 (즉, 800x600의 저해상도 디스플레이와 1600x1200의 고해상도 디스플레이). Scaleform는 동일한 컨텐츠가 모든 해상도를 가질 수 있도록 한다. 또한, 기존의 비트맵 그래픽이 더 적절한 경우를 위해서 비트맵 그래픽도 완벽히 지원한다.

[GFx::Movie::SetViewScaleMode](#)는 크기 조절이 어떻게 이뤄질지를 정의한다. 원래의 화면 비율에 영향을 주지 않고 뷰포트에 무비를 맞추기 위해 다음과 같은 호출을 GFxTutorial::InitGFx()의 마지막에 추가하고 GFx::Movie 객체 설정을 위해 기타 설정을 추가할 수 있다.

```
pUIMovie->SetViewScaleMode(Movie::SM_ShowAll);
```

온라인 문서에 보면 SetViewScaleMode에 관한 네 가지 설정이 가능하며 그 내용은 다음과 같다

SM_NoScale	컨텐츠의 크기를 플래시 스테이지의 원래 해상도로 고정한다.
SM_ShowAll	원래의 화면 비율을 유지하면서 뷰포트에 맞추기 위해 컨텐츠 크기를 조절한다.
SM_ExactFit	원래 화면 비율에 상관 없이 전체 뷰포트를 채우기 위해 컨텐츠 크기를 조절한다. 뷰포트는 채워지겠지만 왜곡이 발생할 수 있다.
SM_NoBorder	원래 화면 비율을 유지하면서 전체 뷰포트를 채우기 위해 컨텐츠 크기를 조절한다. 뷰포트는 채워지겠지만 클리핑이 발생할 수 있다.

보충 요소인 [SetViewAlignment](#)는 뷰포트를 기준으로 컨텐츠의 위치를 제어한다. 화면 비율이 유지될 때 SM_NoScale 또는 SM_ShowAll을 선택하는 경우 뷰포트 중 일부분이 빌 수 있다.

SetViewAlignment는 뷔포트 내에서 컨텐츠를 어디에 놓을 지 결정한다. 이 경우 인터페이스 버튼을 화면 최우측 수직 중심에 놓아야 한다.

```
pUIMovie->SetViewAlignment(Movie::Align_CenterRight);
```

윈도우 크기를 조절할 때 어플리케이션 특성이 어떻게 변하는지를 보기 위해 SetViewScaleMode와 SetViewAlignment의 내용을 변경시켜보자.

SetViewAlignment 함수는 SetViewScaleMode가 SM_NoScale 기본값으로 설정될 경우를 제외하고는 어떠한 영향도 주지 않는다. 더 복잡한 정렬, 크기조절, 위치조정 등의 요구사항에 대하여 Scaleform는 무비가 자신의 크기 및 위치를 선택할 수 있도록 하는 액션스크립트확장을 지원한다. 샘플 액션스크립트가 d3d9guide.fla에 있다.

크기 및 정렬 매개변수는 C++이 아닌 액션스크립트를 통해 설정이 가능하다. SetViewScaleMode와 SetViewAlignment는 액션스크립트 Stage클래스(Stage.scaleMode, Stage.align)가 제공하는 동일한 속성값을 수정한다.

4.3 입력 이벤트 처리

ShadowVolume 의 렌더링 파이프라인이 Scaleform 로 플래시를 렌더링하도록 수정되었기 때문에 이제는 재생중인 플래시와 상호작용 해보도록 하자. 예를 들어서 버튼 위로 마우스가 움직이는 경우 하이라이트가 발생하고 텍스트 상자에 글자를 입력하면 새로운 글자가 나타나도록 해는 것이다.

The [GFx::Movie::HandleEvent](#) 는 눌린 키 또는 마우스 좌표와 같은 이벤트 타입 및 기타 정보를 나타내는 GFx::Event 객체를 전달한다. 어플리케이션은 입력에 기반한 이벤트를 구성해서 이를 적절한 GFx::Movie 로 전달한다.

4.3.1 마우스 이벤트

ShadowVolume 는 MsgProc 콜백에서 Win32 입력 이벤트를 수신한다. Scaleform 가 이벤트를 처리하도록 하는 코드를 실행시키도록 GFxTutorial::ProcessEvent 에 추가한다. 다음 코드는 WM_MOUSEMOVE, WM_LBUTTONDOWN 및 WM_LBUTTONUP 을 처리한다:

```
void ProcessEvent(HWND hWnd, unsigned uMsg, WPARAM wParam, LPARAM lParam,
                  bool *pbNoFurtherProcessing)
{
    int mx = LOWORD(lParam), my = HIWORD(lParam);
    if (pUIMovie)
    {
        if (uMsg == WM_MOUSEMOVE)
        {
            MouseEvent mevent(GFx::Event::MouseMove, 0, mx, my);
            pUIMovie->HandleEvent(mevent);
        }
        else if (pMovieButton && uMsg == WM_LBUTTONDOWN)
        {
            ::SetCapture(hWnd);
            MouseEvent mevent(GFx::Event::MouseDown, 0, mx, my);
            pUIMovie->HandleEvent(mevent);
        }
        else if (pMovieButton && uMsg == WM_LBUTTONUP)
        {
            ::ReleaseCapture();
            MouseEvent mevent(GFx::Event::MouseUp, 0, mx, my);
            pUIMovie->HandleEvent(mevent);
        }
    }
}
```

Scaleform 는 무비의 원래 해상도를 따르지 않고, 지정된 뷰포트의 좌측상단에 기준한 상대 좌표를 필요로 한다. 다음 예는 이 같은 내용을 확실하게 해준다.

예 #1: 화면 크기와 동일한 뷰포트

```
pMovie->SetViewport(screen_width, screen_height, 0, 0,  
                     screen_width, screen_height, 0);
```

이 경우 변환이 필요 없다. 윈도우로부터의 마우스 좌표는 무비가 (0, 0)에 위치하고 있기 때문에 무비의 좌측상단에 대한 상대좌표가 된다. 내부처리를 통해서 원래 무비 해상도와 뷰포트 크기로부터 좌표가 크기 조절된다.

예 #2: 뷰포트가 화면보다 작지만, 화면의 좌측상단에 위치한다.

```
pMovie->SetViewport(screen_width, screen_height, 0, 0,  
                     screen_width / 4, screen_height / 4, 0);
```

이 경우에도 변환이 필요 없다. 뷰포트의 크기가 작아졌기 때문에 버튼의 크기와 위치가 변한다. 그러나 HandleEvent 와 윈도우 화면좌표 모두 윈도우의 좌측상단에 대한 상대 좌표이므로 변환이 필요 없다. 좌표계 크기변환은 뷰포트 크기와 기본 무비해상도에 의해서 내부적으로 처리된다.

예 #3: 뷰포트가 화면보다 작고 가운데에 위치한다

```
movie_width = screen_width / 6;  
movie_height = screen_height / 6;  
pMovie->SetViewport( screen_width, screen_height,  
                      screen_width / 2 - movie_width / 2,  
                      screen_height / 2 - movie_height / 2,  
                      movie_width, movie_height);
```

이 경우 윈도우 화면 좌표의 이동이 필요하다. 무비는 더 이상 (0, 0)에 없으므로 (screen_width / 2 - movie_width / 2, screen_height / 2 - movie_height / 2)에 있는 새로운 위치는 윈도우에게 전달된 스크린 좌표값에서 빼야 한다.

플래시 컨텐츠가 중앙에 있거나 [GFx::Movie::SetViewAlignment](#) 에 의해서 다른 형태로 정렬되어 있다면 이러한 변환이 필요 없다는 점에 유의한다. 주어진 [GFx::Movie::SetViewport](#) 좌표 계에 마우스 좌표가 상대적인 경우라면 SetViewAlignment 와 [SetViewScaleMode](#) 를 통해 정렬과 크기조절이 Scaleform 내부적으로 처리된다

4.3.2 키보드 이벤트

키보드 이벤트도 [GFx::Movie::HandleEvent](#) 를 통해 처리된다. [GFx::KeyEvent](#) 와 GFx::CharEvent 라는 두가지 중요한 이벤트가 있다.

```
KeyEvent(EventType eventType = None,
         Key::Code code = Key::None,
         UByte asciiCode = 0,
         UInt32 wcharCode = 0,
         UInt8 keyboardIndex = 0)

CharEvent(UInt32 wcharCode, UInt8 keyboardIndex = 0)
```

GFx::KeyEvent 는 원시 스캔 코드와 유사하고, GFx::CharEvent 는 처리된 ASCII 문자와 유사하다. 윈도우에서 GFx::CharEvent 는 WM_CHAR 메시지에 대한 응답으로 생성된다. GFx::KeyEvent 는 WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN 및 WM_KEYUP 메시지에 대한 응답으로 생성된다. 다음은 몇 가지 예를 든 것이다.

- SHIFT 키를 누른 상태에서 'c' 키를 누른다: WM_KEYDOWN 메시지에 대한 응답으로 GFx::KeyEvent 가 생성되어야 하며 다음과 같은 내용을 알려줄 것이다.
 - 'c' 키가 눌렸고, 해당 키의 스캔 코드
 - 키를 누르고 있다. 그리고
 - SHIFT 키가 작동 중이다.
- 이와 동시에 “요리가 끝난” ASCII 값 'C'를 Scaleform 로 전달하기 위해 WM_CHAR 에 대한 응답으로 GFx::CharEvent 가 실행되어야 한다.
- 'c' 키를 놓은 경우 WM_KEYUP 메시지에 대한 응답으로 GFx::KeyEvent 를 보내야 한다. 키를 놓는 경우 GFx::CharEvent 를 보낼 필요는 없다.
- F5 키를 누름: 키를 누를 때 GFx::KeyEvent 를 보내며 키가 올라갈 때 두 번째 이벤트를 보낸다. F5 가 출력 가능한 ASCII 코드에 대응하지 않기 때문에 GFx::CharEvent 를 보낼 필요는 없다.

별도의 GFx::KeyEvent 이벤트가 키를 누르고 떼는 이벤트를 위해 전송된다. 플랫폼에 독립적으로 작동하기 위해, 키 코드를 GFx_Event.h 파일에서 정의하여 플래시가 내부적으로 사용하는 키 코드를 매칭시킨다. GFxPlayerTiny.cpp 와 Scaleform Player 프로그램 둘 다 윈도우 스캔 코드를

해당 플래시 코드로 변환하기 위한 코드를 포함하고 있다. 이 장에서의 마지막 코드는 커스텀 3D 엔진과 통합할 때 재사용이 가능한 ProcessKeyEvent 함수를 포함하고 있다.

```
void ProcessKeyEvent(Movie *pMovie, unsigned uMsg, WPARAM wParam, LPARAM lParam)
```

윈도우의 WndProc 함수에서 단순히 WM_CHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN, WM_KEYUP 메시지에 대응하는 함수를 호출한다. 적절한 Scaleform 이벤트가 생성되어 pMovie로 전송될 것이다.

GFx::KeyEvents와 GFx::CharEvents 모두를 전송하는 게 중요하다. 예를 들어, 대부분의 텍스트 상자는 GFx::CharEvents에만 반응하는데 그 이유는 출력 가능한 ASCII 문자와 관련되기 때문이다. 또한 텍스트 상자는 유니코드 문자(예, 중국어 IME)를 수용할 수 있어야 한다. IME 입력의 경우, 원시 키 코드는 유용하지 않으며 최종 문자 이벤트(일반적으로 IME가 처리하는 여러 번의 키 스트로크로 인한 결과)가 텍스트 상자에 필요한 것이다. 반면 리스트 상자 제어는 GFx::KeyEvent를 통해 Page Up과 Page Down 키를 가로챌 필요가 있는데 그 이유는 해당 키가 출력 가능한 문자와 대응되지 않기 때문이다.

Tutorial\Section4.3에 이번 장에서 설명한 최종 코드와 함수가 포함되어 있다. 프로그램을 실행하고 버튼 위로 마우스를 움직인다. 버튼이 올바르게 깜박이며 3D 엔진과의 통합을 필요로 하지 않는 부분이 제대로 작동할 것이다. "Settings"를 누르면 C++ 코드 없이 DX9 설정 창으로 전환되는데 그 이유는 d3d9guide.fla가 액션스크립트를 사용하여 단순하게 구현했기 때문이다.

액션에서 키보드 처리 코드를 보기 위해 "Change Mesh"를 클릭하고 텍스트 입력 상자에 입력을 실시한다. 학습 시에서 앞으로 몇 가지 결정해야 할 사소한 사항이 있다. "Change Mesh" 버튼을 눌러 텍스트 입력 상자를 열 때 나타나는 애니메이션에 주목하자. 애니메이션은 플래시에서 벡터 그래픽을 사용해서 손쉽게 처리할 수 있지만, 기존의 비트맵 기반 인터페이스의 측면에서 볼 때는 비실용적이다. 애니메이션은 추가적인 비트맵뿐만 아니라 커스텀 코드도 필요로 하기 때문에 애니메이션 로딩이 느려지고, 렌더링 시 비용이 발생하며 무엇보다도 코드 작업이 길어진다는 단점이 있다.

4.3.3 히트 테스트

어플리케이션을 실행한 후에 카메라 방향을 변경하기 위해 마우스 왼쪽 버튼을 누른 상태에서 마우스를 움직인다. 이제 UI 요소 중 하나를 골라서 그 위에 마우스를 이동한 후 동일한 행동을

반복한다. 마우스를 클릭하면 인터페이스가 반응하겠지만 여전히 카메라도 움직이게 된다. 이 문제를 어떻게 처리해야 할까?

UI 와 3D 세계 사이의 포커스 제어는 GFx::Movie::HitTest 를 통해 해결이 가능하다. 이 함수는 뷰포트 좌표가 렌더링 된 플래시 컨텐츠 요소를 히트했는지를 판정한다. 마우스 이벤트 처리 후에 GFxTutorial::ProcessEvent 를 수정하여 [HitTest](#) 를 호출한다. UI 요소 상에서 이벤트가 발생하는 경우, 해당 이벤트는 DXUT 프레임워크에 신호를 보냄으로써 카메라 처리 이벤트를 발생시키지 않도록 한다.

```
bool processedMouseEvent = false;
if (uMsg == WM_MOUSEMOVE)
{
    MouseEvent mevent(GFx::Event::MouseMove, 0, (float)mx, (float)my);
    pUIMovie->HandleEvent(mevent);
    processedMouseEvent = true;
}
else if (uMsg == WM_LBUTTONDOWN)
{
    ::SetCapture(hWnd);
    MouseEvent mevent(GFx::Event::MouseDown, 0, (float)mx, (float)my);
    pUIMovie->HandleEvent(mevent);
    processedMouseEvent = true;
}
else if (uMsg == WM_LBUTTONUP)
{
    ::ReleaseCapture();
    MouseEvent mevent(GFx::Event::MouseUp, 0, (float)mx, (float)my);
    pUIMovie->HandleEvent(mevent);
    processedMouseEvent = true;
}

if (processedMouseEvent && pUIMovie->HitTest((float)mx, (float)my,
                                                Movie::HitTest_Shapes))
    *pbNoFurtherProcessing = true;
```

4.3.4 키보드 포커스

어플리케이션을 실행하고 “Change Mesh” 버튼을 클릭하여 텍스트 입력 상자를 연다. 글을 입력하면 키보드 입력이 작동하는데 이는 4.3.2 에서 추가한 코드 때문이다. 그러나 W, S, A, D, Q, E 키는 텍스트 입력에도 사용되나 3D 월드에서 카메라도 이동시킨다. 키보드 이벤트가 Scaleform 를 통해 처리될 뿐 아니라 3D 카메라로도 전달되고 있는 것이다.

이 사항을 해결하기 위해선 텍스트 입력 상자가 포커스를 갖는 지에 관해서 결정을 할 필요가 있다. 6.1.2 장에선 플래시 액션스크립트를 사용하여 우리의 이벤트 핸들러가 포커스를 추적할 수 있도록 C++에 이벤트를 어떻게 보내는지를 다룰 것이다.

4.3.5 터치 이벤트

터치 이벤트는 마우스 이벤트와 비슷하지만, 사용자가 화면을 터치할 때만 전송됩니다. 터치 이벤트를 사용하기 위해 Windows API에서 Scaleform으로 직접 원시 터치 데이터를 보낼 수 있습니다. 필요한 최소 플랫폼은 터치가 가능한 시스템에서 실행되는 Windows 7입니다. 최소 플랫폼이 없으면 튜토리얼의 이 섹션을 따를 수 없고 다음 섹션으로 건너뛰어야 합니다.

컴파일하기 전에 프로젝트의 전처리기 정의에 **WINVER=0x601**을 추가합니다. 이 정의는 Windows 7의 기능(예: 다중 터치)을 컴파일하는 Windows API임을 나타냅니다. ShadowVolume.cpp에서 지원되지 않는 코드가 컴파일되지 않도록 보호하는 매크로 `#define ENABLE_MULTITOUCH`의 주석 처리를 제거합니다.

특수 다중 터치 라이브러리를 사용하기 위해 `#include <windows.h>` 가 추가되었습니다. 창 초기화 중에 RegisterTouchWindow를 통해 터치 이벤트를 받도록 등록하여 처리를 위한 MsgProc 콜백에서 WM_TOUCH 메시지를 수신할 수 있습니다.

```
if(uMsg == WM_TOUCH)
{
    ProcessTouchEvent(pUIMovie, uMsg, wParam, lParam);
}
```

ProcessTouchEvent 는 WM_TOUCH 메시지의 분석과 전달을 담당하는 함수입니다. Windows 함수는 터치 좌표를 픽셀 좌표로 변환하는 데 사용됩니다. 나중에 GFx::TouchEvent 가 생성되고 HandleEvent 를 통해 동영상으로 전송됩니다. GFx::TouchEvent 클래스는 고유 ID(OS 에서 관리), x/y 좌표, 접촉 영역, 압력(0~1) 및 주요 지점인지 아닌지로 구성됩니다.

```
TouchEvent( EventType evtType,
            unsigned id,
            float _x,
            float _y,
            float wcontact = 0,
            float hcontact = 0,
            bool primary = true,
            float pressure = 1.0f)
```

SWF 동영상이 이제 터치 이벤트를 수신할 수 있으며, 기본적으로 GFx 는 0 개의 최대 터치 지점을 지원합니다. 마지막 단계는 지원하는 최대 터치 지점 수 및 GFx::GestureEvent 도 전달할지 등, 다중

터치 환경을 설명하는 상속된 MultitouchInterface 를 정의하는 것입니다. 이 튜토리얼에서는 자체 제스처 이벤트를 다루지 않지만, 이 섹션에서 설명하는 동일한 원칙을 사용하여 Scaleform에 노출할 수 있습니다.

```
class FxPlayerMultitouchInterface : public MultitouchInterface
{
public:
    // 하드웨어에서 지원하는 터치 지점의 최대 개수 반환
    virtual unsigned GetMaxTouchPoints() const { return 2; }

    // 지원하는 제스처의 비트마스크 반환(현재 없음)
    virtual UInt32 GetSupportedGesturesMask() const { return 0; }

    // 다중 터치를 지원합니까?
    virtual bool SetMultitouchInputMode(MultitouchInputModule) { return
true; }
};
```

SetMultitouchInterface 를 통해 동영상에 이 인터페이스 클래스의 참조를 전송하면 터치 이벤트가 성공적으로 동영상에 전파됩니다.

5 지역화와 폰트 소개

5.1 폰트개요 및 기능

Scaleform은 효율적이고 유연한 폰트 및 지역화 시스템을 제공한다. 다양한 폰트, 포인트 크기 및 스타일을 효율적이고 낮은 메모리 사용량으로 한꺼번에 렌더링시킬 수 있다. 폰트 데이터는 내장 플래시 폰트, 공유 폰트 라이브러리, 운영 체계 및 TTF 폰트 라이브러리 디렉토리를 통해 얻을 수 있다. 벡터 기반의 폰트 압축은 크기가 큰 아시아 폰트의 메모리 사용량을 감소시킨다. 폰트 지원은 완벽한 크로스 플랫폼으로 이뤄지며 콘솔 시스템, 윈도우 및 리눅스에서 동등하게 작동한다. Scaleform의 폰트와 국제화 능력에 대한 전체 문서가 개발자 센터 문서 페이지에 나와 있으며 주소는 <http://gameware.autodesk.com/scaleform/developer/?action=doc>이다.

전형적인 폰트 솔루션은 전체 폰트의 각 글자를 텍스처로 렌더링하고 그 후 필요한 경우 글자는 폰트 텍스처로부터 화면으로 텍스처 매핑되는 방법을 사용한다. 이 방법은 서로 다른 폰트 크기와 스타일이 필요하게 될 경우에 추가 텍스처가 필요하다. 라틴 문자는 메모리 사용량이 수용 가능한 수준이지만 5000 개의 글리프로 이루어진 아시아 폰트의 렌더링은 비실용적이다. 많은 양의 소중한 텍스처 메모리가 필요하며 또한 각 글리프를 렌더링 할 처리 시간도 필요하다. 서로 다른 폰트 크기와 스타일을 동시에 렌더링 해야 함은 물론이다.

Scaleform은 동적 폰트 캐시로 이 문제를 해결한다. 요청이 있을 때 캐시에 문자를 렌더링하며 캐시 내 슬롯은 필요 시 교체된다. 서로 다른 폰트 크기와 스타일은 Scaleform의 지능형 글리피팩킹(glyph-packing) 알고리즘으로 단일 공용 캐시에서 공유된다. Scaleform은 벡터 폰트로 작업을 하는데, 이는 어떤 크기의 문자 레더링에서도 하나의 TTF 폰트를 메모리에 저장하면 된다는 뜻이다. 마지막으로 Scaleform은 "fake italic"과 "fake bold" 기능을 제공하는데, 이를 통해 추가적인 메모리를 절약하면서 한 폰트의 벡터 표현을 이탤릭이나 볼드체로 변환할 수 있다

매우 큰 텍스트는 분할 삼각형으로 직접 렌더링이 가능하다. 이는 크기가 큰 텍스트가 많지 않은 경우 유효하며 게임의 타이틀 화면 등과 같은 부분에서 유용하다.

Bin\Data\AS2\Samples\FontConfig에서 폰트 설정 예제를 찾아 sample.swf 파일을 Scaleform Player D3D9 창에 드래그해서 재생해보자.

早上好

你好! 早上好 !

多想你呀 !

Chinese (繁體中文版)

그림 4a: 작은 중국어

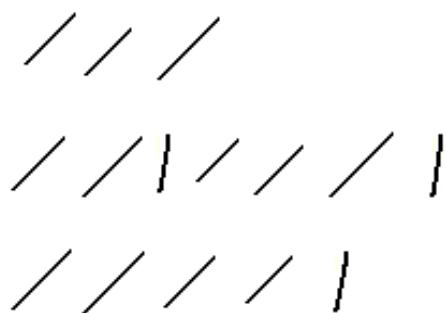


그림 4b: 와이어 프레임 표현

이러한 문자의 와이어 프레임 출력을 보려면 CTRL+W를 누르면 각각의 글자가 텍스처 맵핑된 2개의 텍스처로 출력된다. 글자의 크기가 작기 때문에 동적 폰트 캐시를 통해 비트맵으로 렌더링하는 게 더 효율적이다.

와이어 프레임 모드를 유지한 상태에서 창의 크기를 증가시킨다. 글자가 텍스처 맵에서 내부가 단일 컬러로 채워진 삼각형 렌더링으로 변환된다:



그림 4c: 큰 중국어

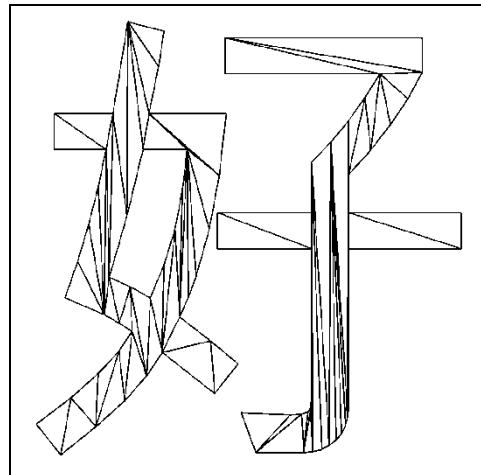


그림 4d: 와이어 프레임 표현

큰 글자의 경우 단일 컬러 삼각형으로 렌더링하는게 더 효율적이다. 크기가 큰 비트맵을 여는 것은 과도한 메모리 대역폭을 사용하기 때문에 메모리 사용량이 크게 증가하게된다. 컬러가 필요한 픽셀만 설정하여, 글자의 공란을 처리하는 데 사용되는 불필요한 처리 과정을 피하게 된다. 그림

4c 와 4d 에서, Scaleform 는 특성한계를 넘어서는 글자의 크기를 파악하고, 이를 분할 삼각형으로 변환한 후 비트맵 대신 기하형태로 글자를 렌더링한다.

폰트 소프트 쉐도우, 블러 및 기타 효과를 지원한다. 필요한 효과를 생성하려면 단순히 플래시 내의 텍스트 필드에 적절한 필터를 설정한다. 추가적인 세부사항은 폰트 및 텍스트 설정 개요에 나와 있다. 예제 파일의 이름은 gfx_2.1_texteffects_sample.zip 이며 개발자 센터에서 다운로드 받을 수 있다.



그림 5: 텍스트 효과

6 C++, 플래시, 액션스크립트 연결하기

플래시의 액션스크립트스크립트 언어를 통해 인터랙티브 무비 컨텐츠를 만들수 있다. 버튼 클릭, 특정 프레임의 도달 또는 무비의 로딩과 같은 이벤트는 무비 컨텐츠를 동적으로 변경하고, 무비의 흐름을 제어하며 추가 무비를 로딩하기 위한 코드를 실행시킬 수 있다. 액션스크립트는 플래시만으로 제작된 미니 게임을 만들기에 충분할 정도로 강력하다. 대부분의 프로그래밍 언어와 마찬가지로 Scaleform 는 액션스크립트 변수 및 서브 루틴을 제공한다. 액션스크립트 변수 및 배열을 직접 조작하고 액션스크립트 서브루틴을 직접 호출하기 위해 Scaleform 는 C++ 인터페이스를 제공한다. 또한 액션스크립트가 C++ 프로그램에게 이벤트와 데이터를 전달할 수 있도록 하는 콜백 메커니즘을 제공한다.

6.1 액션스크립트에서 C++로

Scaleform 는 C++ 어플리케이션이 ActionScript::FSCommand 및 외부 인터페이스로부터 이벤트를 받을 수 있도록 두 가지 메커니즘을 제공한다. FScommand 및 외부 인터페이스 모두 이벤트 통지를 받기 위해 C++ 이벤트를 GFx::Loader 에 등록한다. FSCommand 이벤트는 ActionScript 의 FSCommand 함수를 통해 발생하며 두 개의 스트링 문장을 받으나 값을 반환할 수 없다. 외부 인터페이스 이벤트는 액션스크립트가 flash.external.ExternalInterface. 함수를 호출하고, [GFx::Value](#) 자료 형 숫자 목록을 수신할 때 발생된다(6.1.2 에서 다룸). 그리고 호출자에게 값을 반환할 수 있다.

제한적인 유연성 때문에, FSCommand 는 ExternalInterface 를 통해 무효가 되므로 더 이상 사용하지 않기 바란다. 여기에서 설명하고 있는 이유는 fscommand 를 옛날 코드에서 볼 수도 있기 때문이다. 이에 더해, gfxexport 는 -fstree, -fslist 및 -fsparams 옵션을 사용하면 SWF 파일 내에서 사용된 모든 fscommand 의 보고서를 생성할 수 있다. 이는 ExternalInterface 에서 불가능하기 때문에, 몇몇 경우 fscommand 를 사용하기 위한 충분한 이유가 될 수 있다.

6.1.1 FS Command 콜백

ActionScript 의 fscommand 함수는 명령과 데이터 전달인자를 호스트 어플리케이션에 전달한다. 액션스크립트내에서의 일반적인 사용법은 다음과 같다.

```
fscommand( "setMode" , "2" );
```

부울 또는 정수같은 값은 fscommand에서 문자열로 변환 된다. ExternalInterface 정수를 직접 수신할 수 있다.

이 명령은 두 개의 문자열을 GFx FSCommand 핸들러로 전달한다. 어플리케이션은 GFx::FSCommandHandler를 상속받아서 fscommand 핸들러를 등록하고, GFx::Loader 나 개별 GFx::Movie 객체로 등록한다. 만약 명령 핸들러가 GFx::Movie에 설정되면 해당 무비 인스턴스에서 실행된 fscommand만을 호출하기 위한 콜백을 받는다. GFxPlayerTiny의 예는 이러한 프로세스를 예로 나타낸 것이내며 ("FxPlayerFSCommandHandler"를 볼 것) ShadowVolume에 유사한 코드를 추가한다. 이 장에서의 최종 코드는 Tutorial\Section6.1에 있다.

우선, GFx::FSComandHandler를 서브 클래스화 한다.

```
class OurFSCommandHandler : public FSCommandHandler
{
public:
    virtual void Callback(Movie* pmovie,
                          const char* pcommand, const char* parg)
    {
        GFxPrintf("FSCommand: %s, Args: %s", pcommand, parg);
    }
};
```

콜백 메소드는 액션스크립트내 fscommand로 전달된 두 개의 문자열뿐만 아니라 fscommand를 호출한 무비 인스턴스의 포인터도 받는다.

다음으로, GFxTutorial::InitGFx() 내에 GFx::Loader 객체를 생성한 후 핸들러를 등록한다.

```
// Register our FSCommand handler
Ptr<GFxFSCommandHandler> pcommandHandler = *new OurFSCommandHandler;
gfxLoader->SetFSCommandHandler(pcommandHandler);
```

GFx::Loader로 핸들러를 등록하면 모든 GFx::Movie가 이 핸들러의 속성을 물려받게 된다. SetFSCommandHandler는 이러한 기본 설정을 오버라이드 하기한 개별 무비 인스턴스에서 호출가능하다.

우리의 커스텀 핸들러는 단순히 fscommand 이벤트를 디버그 콘솔에 출력한다. ShadowVolume을 실행하고 "Toggle Fullscreen" 버튼을 클릭한다. UI 이벤트가 발생할 때마다 이벤트가 출력됨을 확인한다.

```
FSCommand: ToggleFullscreen, Args:
```

Open Flash/HUDMgr.as in Flash Studio. This class is referenced from d3d9guideAS3.fla. The association between this external class and the Flash content is made by setting the ActionScript class for the "hudMgr" Symbol in d3d9guideAS3's Symbol Library.

Compare the events printed to the screen with the fscommand() calls made from the HUDMgr class in the following function:

```
function toggleFullscreen(ev:MouseEvent) {
    fscommand( "ToggleFullscreen" );
}
```

연습삼아 fscommand("ToggleFullscreen")을 fscommand("ToggleFullscreen", hud.visible)로 바꿔서 hud.visible.value 값을 C++에 출력한다. 플래시 무비 파일을 엑스포트해서(CTRL+ALT+Shift+S) d3d9guideAS3.swf 를 교체한다.

UI 상의 버튼은 FSCommand 이벤트가 발생할 때 적절한 코드를 작동시킴으로써 ShadowVolume 샘플에 통합될 수 있다. 예로서, 다음의 코드를 fscommand 핸들러에 추가하여 전체 화면 모드 토글 기능을 추가한다.

```
if (strcmp(pcommand, "ToggleFullscreen") == 0)
    doToggleFullscreen = true;
```

DXUT 함수인 OnFrameMove 는 프레임 렌더링 직전에 호출된다. OnFrameMove 콜백 끝부분에 해당 코드를 추가한다.

```
if (doToggleFullscreen)
{
    doToggleFullscreen = false;
    DXUTToggleFullScreen();
}
```

일반적으로, 이벤트 핸들러는 넌블로킹이어야 하며 가능한 빨리 호출자로 복귀해야 한다. 이벤트 핸들러는 일반적으로 Advance 또는 Invoke 호출간에만 호출된다.

6.1.2 ExternalInterface

플래시의 ExternalInterface.call 은 fscommand 와 유사하나 유연한 전달인자 처리와 값을 반환할 수 있다는 측면 때문에 더 선호된다.

ExternalInterface 핸들러를 등록하는 것은 fscommand 핸들러를 등록하는 것과 유사하다.

```
class OurExternalInterfaceHandler : public ExternalInterface
{
public:
    virtual void Callback(Movie* pmovieView,
                          const char* methodName,
                          const Value* args,
                          unsigned argCount)
    {
        GFxPrintf("ExternalInterface: %s, %d args: ",
                  methodName, argCount);
        for(unsigned i = 0; i < argCount; i++)
        {
            switch(args[i].GetType())
            {
                case Value::VT_Null:
                    GFxPrintf("NULL");
                    break;
                case Value::VT_Boolean:
                    GFxPrintf("%s", args[i].GetBool() ? "true" : "false");
                    break;
                case Value::VT_Int:
                    GFxPrintf("%s", args[i].GetInt());
                    break;
                case Value::VT_Number:
                    GFxPrintf("%3.3f", args[i].GetNumber());
                    break;
                case Value::VT_String:
                    GFxPrintf("%s", args[i].GetString());
                    break;
                default:
                    GFxPrintf("unknown");
                    break;
            }
            GFxPrintf("%s", (i == argCount - 1) ? "" : ", ");
        }
        GFxPrintf("\n");
    };
};
```

그리고 핸들러를 GFx::Loader에 등록한다.

```
Ptr<ExternalInterface> pEIHandler = *new OurExternalInterfaceHandler;
gfxLoader.SetExternalInterface(pEIHandler);
```

외부 인터페이스 호출은 텍스트 입력 상자가 포커스를 얻거나 잃는 경우 액션스크립트를 통해 생성된다. Open d3d9HUD.as used by d3d9guideAS3.fla and look at the ActionScript for the following functions:

```
function onSubmit(path:String) {
    ExternalInterface.call("MeshPath", path);
    CloseMeshPath();
```

```

}

function onFocusIn(ev: FocusHandlerEvent) {
    ExternalInterface.call("MeshPathFocus", true);
}

function onFocusOut(ev: FocusHandlerEvent) {
    ExternalInterface.call("MeshPathFocus", false);
}

```

MeshPathFocus 콜백은 텍스트 입력 상자가 포커스를 얻거나 잃는 경우 호출된다.

ExternalInterface 호출은 ExternalInterface 핸들러를 작동시키고, 텍스트 입력 상자 (참 또는 거짓)의 포커스 상태를 전달하며, 임의의 명령어인 "MeshPathFocus"를 전달한다. 텍스트 입력을 열어서 텍스트 영역을 클릭한 후 화면의 빈 공간을 클릭하여 텍스트 입력이 갖는 포커스를 제거한다. 콘솔 출력은 다음과 메시지와 유사할 것이다.

```

Callback! MeshPathFocus, nargs = 1
arg(0) = true

Callback! MeshPathFocus, nargs = 1
arg(0) = false

```

이벤트 핸들러를 수정하여 포커스를 얻거나 잃을 경우를 감지하고 해당 정보를 GFxTutorial 객체로 전달할 수 있다.

```

if(strcmp(methodName, "MeshPathFocus") == 0 && argCount == 1 &&
args[0].GetType() == Value::VT_Boolean) {
    if (args[0].GetType() == Value::VT_Boolean)
        gfx->SetTextboxFocus(args[0].GetBool());
}

```

GFxTutorial::ProcessEvent 는 텍스트 상자가 포커스를 받는 경우 키보드 이벤트만을 무비에 전달한다. 만약 키보드 이벤트가 텍스트 상자에 전달되면 플래그를 설정하여 3D 엔진으로의 전달을 막게 된다.

```

if (uMsg == WM_SYSKEYDOWN || uMsg == WM_SYSKEYUP ||
    uMsg == WM_KEYDOWN || uMsg == WM_KEYUP ||
    uMsg == WM_CHAR)
{
    if (textboxHasFocus || wParam == 32 || wParam == 9)
    {
        ProcessKeyEvent(pUIMovie, uMsg, wParam, lParam);
        *pbNoFurtherProcessing = true;
    }
}

```

스페이스 (ASCII 코드 32)와 탭 (ASCII 코드 9)은 항상 전달되며, 각각 "Toggle UI"와 "Settings"에 대응한다.

사용자가 렌더링 중인 메시를 교체할 수 있도록 "Change Mesh" 버튼을 클릭하여 텍스트 상자를 열고, 새로운 메시 이름을 입력한 후 엔터키를 누른다. 엔터키를 눌렀을 때, 텍스트 상자는 액션스크립트이벤트 핸들러를 호출하는데 이는 새로운 메시의 이름으로 ExternalInterface 를 호출한다. ExternalInterfaceHandler::Callback 에서 추가되는 코드는 다음과 같다.

```
static bool doChangeMesh = false;
static wchar_t changeMeshFilename[MAX_PATH] = L"";

...
if(strcmp(methodName, "MeshPath") == 0 && argCount == 1)
{
    doChangeMesh = true;
    const char *filename = args[0].GetString();
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, filename, -1,
                         changeMeshFilename, _countof(changeMeshFilename));
}
```

전체화면 토글과 함께, 실제 작업은 DXUT OnFrameMovie 콜백에서 이뤄진다. 코드는 기본 DXUT 인터페이스용 이벤트 핸들러를 기반으로 한다.

6.2 C++에서 액션스크립트로

6.1은 액션스크립트가 어떻게 C++로 호출이 가능한지를 다루고 있다. 여기에선 C++ 프로그램이 무비 재생과의 통신을 시작하는지에 대해 Scaleform 함수를 사용한 기타 통신방법을 다룬다. Scaleform은 C++ 함수를 지원하여 액션스크립트 변수를 얻고(get) 설정(set)하며 동시에 액션스크립트 서브 루틴을 호출한다.

6.2.1 액션스크립트 변수 조작

Scaleform은 [GetVariable](#) 과 [SetVariable](#) 을 지원하는데 이를 통해 액션스크립트 변수의 직접적인 조작이 가능하다. Tutorial\Section6.2에 나와 있는 코드를 수정하여 Scaleform Player 프로그램이 사용하는 노란색 HUD 디스플레이를 로딩하고 (fxplayer.swf) F5를 누를 때마다 카운트를 증가시킨다

```
void GFxTutorial::ProcessEvent(HWND hWnd, unsigned uMsg, WPARAM wParam,
```

```

        LPARAM lParam, bool *pbNoFurtherProcessing)
{
    int mx = LOWORD(lParam), my = HIWORD(lParam);

    if (pHUDMovie && uMsg == WM_KEYDOWN)
    {
        if (wParam == VK_F5)
        {
            int counter = (int)pHUDMovie
                ->GetVariableDouble("_root.counter");
            counter++;
            pHUDMovie->SetVariable("_root.counter",
                Value((double)counter));
            char str[256];
            sprintf_s(str, "testing! counter = %d", counter);
            pHUDMovie->SetVariable("_root.MessageText.text", str);
        }
    }
    ...
}

```

[GetVariableDouble](#)은 _root.counter 변수 값을 반환한다. 초기에는 변수가 존재하지 않으며 GetVariableDouble 은 0 을 반환한다. 카운트가 증가하면 새로운 값은 SetVariable 을 통해 _root.counter 에 저장된다. [GFx:Movie](#)에 관한 온라인 문서는 GetVariable 과 SetVariable 의 다양한 변종을 다루고 있다.

fxplayer 플래시 파일은 어플리케이션을 통해 임의의 텍스트로 설정될 수 있는 두 개의 동적 텍스트 필드를 갖고 있다. MessageText 텍스트 필드가 화면 중앙에 위치하며 HUDText 변수는 화면 좌측 상단에 위치한다. _root.counter 변수값을 기반으로 문자열이 생성되며 SetVariable 은 메시지 텍스트의 업데이트에 사용된다.

성능관련 주의사항: 플래시의 동적 텍스트 필드를 변경할때 선호되는 방법은 TextField.text 변수를 설정하거나 [GFx::Movie::Invoke](#) 를 호출하여 텍스트를 변경하는 액션스크립트 루틴을 실행시키는 것이다(보다 자세한 내용은 6.2.2 참고). 동적 텍스트 필드를 임의의 변수와 결합하지 말라. 그리고, 텍스트를 변경학; | 위해 변수를 바꾸지 말라. 작동은 되더라도 Scaleform 가 매 프레임마다 변수값을 계산해야 하기 때문에 성능상 장애를 유발할 수 있다.



그림그림 8: HUD 텍스트 값을 변경하는 SetVariable

앞에서 소개한 사용예는 문자열이나 숫자를 변수로 직접 처리한다.

SetVariable은 “sticky” 한 대입을 선언하는 GFx::Movie::SetVarType 형태에 대해 세 가지 옵션이 있다. 이는 할당된 변수가 플래시 타임라인에 아직 생성되지 않은 경우 유용하다. 예를 들어, 텍스트 필드의 _root.mytextfield 가 무비에서 3 번째 프레임까지 생성되지 않은 경우를 살펴보자. 만약 SetVariable(“_root.mytextfield.text”, “testing”, SV_Normal)를 무비 생성 직후인 프레임 1에서 호출했다면, 대입이 아무런 효과가 없을 것이다. 호출이(기본값인) SV_Sticky로 호출되었다면 요청에 큐에 대기되고 _root.mytextfield.text 값에 한번만 적용되고, 프레임 3에서 유효하게 될 것이다. 이를 통해 C++을 이용해 무비를 초기화하는 것이 더 쉬워진다. 일반적으로 SV_Normal 이 SV_Sticky에 비해 더욱 효율적이므로 가능하다면 SV_Normal 을 사용하는게 좋다.

SetVariableArray 는 한 번의 동작으로 변수 배열을 플래시에 전달한다. 이 함수는 동적으로 작동하는 드롭 다운 목록 제어와 같은 작동에 사용할 수 있다. 다음의 코드를 GFxTutorial::InitGFx()에 추가하고 _root.SceneData 드롭 다운 값을 설정한다.

```
// Initialize the scene dropdown
Value sceneData[3];
sceneData[0].SetString("Scene with shadow");
sceneData[1].SetString("Show shadow volume");
sceneData[2].SetString("Shadow volume complexity");
pUIMovie->SetVariableArraySize("_root.SceneData", 3);
pUIMovie->SetVariableArray(Movie::SA_Value,
```

```
"_root.SceneData", 0, sceneData, 3);
```

Tutorial\Section6.1에 있는 코드는 추가적인 ExternalInterface 핸들러를 포함하고 있는데 이는 조명 제어, 조명 수, 장면 형태 및 원래 ShadowVolume 인터페이스에 있는 기타 제어에 응답하기 위한 것이다.

주: 이 예는 보여주기 위한 목적으로 C++을 이용해 드롭 다운 메뉴를 작성한 것이다. 일반적으로 정적인 선택 사항이 있는 드롭 다운 메뉴는 C++ 대신 액션스크립트로 초기화되어야 한다. 6.2부터, d3d9guide.swf/fla 파일은 드롭 다운 목록 초기화를 위해 액션스크립트를 사용한다.

6.2.2 액션스크립트 서브루틴 실행

액션스크립트 변수 수정뿐만이 아니라, 액션스크립트 코드는 [GFx::Movie::Invoke](#)를 통해 구동될 수 있다. 이는 더욱 복잡한 처리, 애니메이션 구동, 현재 프레임 변경, 프로그램을 통한 UI 제어 상태 변경 및 새로운 버튼이나 텍스트와 같은 UI 컨텐츠를 동적으로 생성할 때 유용하다.

이 장에선 GFx::Movie::Invoke를 사용하여 프로그램적으로 F6를 누르면 “Change Mesh” 텍스트 입력 상자를 열고 F7을 누르면 닫는다. 라디오 버튼을 클릭하면 애니메이션이 발생하기 때문에, SetVariable로 상태를 변경시키는 방법으로는 이러한 처리를 할 수 없다. SetVariable은 애니메이션을 구동시킬 수 없지만 Invoke로 호출된 액션스크립트 루틴은 할 수 있다.

GFx::Movie::Invoke는 WM_CHAR 키보드 핸들러에서 openMeshPath 루틴을 실행하기 위해 호출될 수 있다.

```
...
else if (wParam == VK_F6)
{
    bool retval = pHUDMovie->Invoke("root.ui.hud.OpenMeshPath", "");
    GFxPrintf("root.ui.hud.OpenMeshPath returns '%d'\n", (int) retval);
}
else if (wParam == VK_F7)
{
    const char *retval = pHUDMovie->Invoke(
        "root.ui.hud.CloseMeshPath", "");
    GFxPrintf("root.ui.hud.CloseMeshPath returns '%d'\n", (int) retval);
}
...
```

Invoke 를 사용하는 데 있어 일반적인 에러는 아직 가용하지 않은 액션스크립트 루틴을 호출하는 것인데, 이 경우 Scaleform 로그에 에러가 기록된다. 액션스크립트 루틴은 관련된 프레임이 재생되거나 관련된 인접 객체의 로딩이 이뤄질 때까지 가용하지 않다. 프레임 1에 있는 모든 액션스크립트는 첫 번째 [GFx::Movie::Advance](#) 호출이 이뤄지거나, true로 설정된 initFirstFrame 으로 [GFx::MovieDef::CreateInstance](#) 를 호출할 때 이뤄진다.

이 예에서는 Invoke 의 printf 스타일을 사용했다. 다른 버전의 함수는 GFx::Value 를 사용해서 효율적으로 비 스트링 문자열을 처리한다. [InvokeArgs](#) 는 Invoke 와 동일하나 어플리케이션이 변수 목록에 포인터를 제공하기 위해 va_list 를 사용한다는 점은 다르다. Invoke 와 InvokeArgs 와의 관계는 printf 와 vprintf 와의 관계와 유사하다.

6.3 플래시 파일간 통신

이제까지 논했던 통신 기법은 모두 C++ 을 사용했다. UI 가 다중 SWF 파일로 나뉘는 대형 어플리케이션의 경우, 상당량의 C++ 을 사용해서 인터페이스 내의 컴포넌트들이 서로 통신할 수 있도록 해야 한다.

예를 들어, 다중 사용자 온라인 게임 (MMOG) 는 인벤토리 HUD, 아바타 HUD, 물물교환 가게가 있을 수 있다. 검이나 돈 같은 아이템을 플레이어의 인벤토리에서 상점으로 이동시켜 다른 플레이어에게 줄 수도 있다. 플레이어가 옷 아이템을 갖고 있는 경우 해당 아이템은 인벤토리 HUD에서 아바타 HUD로 이동하게 된다.

이러한 세 가지 인터페이스는 별도의 SWF 파일로 나뉘어야 하며 메모리 보호를 위해 별도로 로딩되어야 한다. 그러나 이러한 세 가지 GFx::Movie 객체 사이의 통신 유지를 위해 C++ 코드를 작성하는 것은 순식간에 귀찮은 작업으로 변모한다.

더 좋은 방법이 있다. 액션스크립트는 loadMovie 와 unloadMovie 기법을 지원하는데 이를 통해 다중 SWF 파일을 스왑하여 하나의 GFx::Movie 를 만들 수 있다. SWF 무비가 동일한 GFx::Movie 에 있는 한, 동일한 변수 공간을 공유하므로 로딩할 때마다 초기화를 위해 C++ 코드를 사용해야 할 필요성이 없어진다.

일례로, 공유 변수 공간에 무비를 로딩하고 해제하기 위한 액션스크립트 함수로 구성된 container.swf 파일을 만든다. 컨테이너 파일은 어떠한 아트 애셋도 포함하지 않으며 단지

몇 개의 액션스크립트로만 구성되어 무비의 로딩과 언로딩을 관리하게 된다. 첫 번째 단계는 MovieClipLoader 객체를 생성하는 것이다.

```
var mclLoader:MovieClipLoader = new MovieClipLoader();
```

이 내용 및 여기에서 사용된 기타 액션스크립트 함수에 대한 자세한 정보는 플래시 문서를 참고한다.

```
function loadMovie(url:String):void {  
    var loader = new Loader();  
    loader.load( new URLRequest( url ) );  
    addChild( loader )  
}
```

7 텍스처에 렌더링하기

텍스처 렌더링은 3D 렌더링에서 자주 사용하는 고급 기술입니다. 이 기술을 사용하면 백 버퍼 대신 텍스처 표면에 렌더링할 수 있습니다. 결과 텍스처는 일반 텍스처와 마찬가지로 원하는 장면 지오메트리에 적용할 수 있습니다. 예를 들어 이 기술로 “게임 내 빌보드”를 만들 수 있습니다. 먼저 Flash Studio에서 빌보드를 SWF 파일로 작성하고 이 SWF 파일을 텍스처에 렌더링합니다. 그런 다음 이 텍스처를 적당한 장면 지오메트리에 적용합니다. 이 튜토리얼에서는 간단하게 이전 튜토리얼의 UI를 뒷면 벽에다 렌더링하겠습니다. 마우스 가운데 버튼을 누른 채로 광원을 움직이면 UI 외관이 이에 따라 바뀝니다.



이제 이 예제의 작동원리에 대해 알아보도록 하겠습니다.

- 예전의 모든 튜토리얼에서는 항상 cell.x 장면 파일을 불러왔습니다. 이 파일은 방바닥과 벽, 천장을 표시하는 데 필요한 꼭지점(vertex) 좌표와 삼각형 인덱스 그리고 텍스처 정보를 포함합니다. 이 파일을 살펴보면 네 벽면 모두에 같은 벽 텍스처(cellwall.jpg)가 사용되었다는 것을 알 수 있습니다. 이 튜토리얼에서는 UI를 뒷벽에 렌더링하려고 합니다. 먼저 뒷벽으로

사용할 텍스처를 별도로 만들고 파일 이름을 cellwallRT.jpg 라고 불입니다. 그런 다음 MeshMaterialList 배열이 뒷벽을 렌더링할 때 이 텍스처를 참조하도록 수정합니다.

```
Material {
    1.000000;1.000000;1.000000;1.000000;;
    40.000000;
    1.000000;1.000000;1.000000;;
    0.000000;0.000000;0.000000;;
    TextureFilename {
        "cellwallRT.jpg";
    }
}
```

2. 이전 튜토리얼에서 말했듯이 cell.x 파일은 DXUT 프레임워크 내부에서 구문분석(parsing)됩니다. DXUT는 cell.x 파일을 구문분석하는 동안 꼭지점 버퍼와 인덱스 버퍼를 초기화하고 재질(material) 목록이 참조하는 텍스처를 만듭니다. 텍스처는 CreateTexture 호출을 통해 만들어지지만, 이 함수에 전달된 이용 매개변수는 Render Texture에서 사용하기에 적합하지 않습니다. 이에 대한 자세한 내용을 보려면 DXSDK 문서를 참조하십시오. 렌더 타겟으로 사용할 수 있는 텍스처를 만들려면 올바른 이용 매개변수를 사용하여 렌더 텍스처를 다시 만들고 이 텍스처를 배경 메시에 붙여야 합니다. 또한 DXUT가 만든 원본 텍스처를 삭제하여 메모리 누수를 방지하십시오.

```
pd3dDevice->CreateTexture(rtw,rth,0,
    D3DUSAGE_RENDERTARGET|D3DUSAGE_AUTOGENMIPMAP, D3DFMT_A8R8G8B8,
    D3DPOOL_DEFAULT, &g_pRenderTex, 0);

g_Background[0].m_pTextures[BACK_WALL] = g_pRenderTex;
ptex->Release();
```

3. 그런 다음 백버퍼대신 GFx를 텍스처에 렌더링하도록 AdvanceAndRender 함수를 수정합니다. 이 과정에서 중요한 단계는 다음과 같습니다.
 - a. 첫 번째, 렌더링이 끝난 다음 원본을 복구할 수 있도록 원본 백버퍼의 표면을 저장합니다.

```
pd3dDevice->GetRenderTarget(0, &poldSurface);
pd3dDevice->GetDepthStencilSurface(&poldDepthSurface);
```
 - b. 그런 다음 텍스처에서 렌더 표면(surface)을 얻어 렌더링 대상으로 설정합니다.

```
ptex->GetSurfaceLevel(0, &psurface);
HRESULT hr = pd3dDevice->SetRenderTarget(0, psurface );
```
 - c. 무비 뷰 포트를 텍스처 크기에 맞추고 Display를 호출한 다음 a 단계에서 저장한 원본 렌더 표면을 복구합니다.

8 OpenGL 샘플

Nvidia 공간 분할의 예를 기반으로 하는 소규모 OpenGL/GLUT 통합 샘플(Tutorial\OpenGL)이 첨부되어 있으니 참조하시기 바랍니다

(<http://developer.download.nvidia.com/SDK/10/opengl/samples.html#tessellation>). 해당 샘플은 극히 최소화된 것으로 D3D9 자습서와는 약간 다른 접근 방식을 사용합니다. tessellation.cpp 를 tessellation_original.cpp 와 비교해보면 알 수 있듯이 원본 NVIDIA 샘플에 추가된 코드는 몇 줄에 불과합니다. 주 프로그램은 Init(), ShutDown, AdvanceAndDisplay() 등과 같은 상위 수준 호출을 통해 GFx 와 통신하며 모든 GFx 특정 구현 코드는 GFxPlayerImpl 클래스(GFxPlayerGL.cpp)에 캡슐화됩니다.

9 GFxExport 로 전처리

지금까지는 플래시 SWF 파일을 직접 로딩했다. 아티스트가 개발중인 SWF 내부에 접근할 수 있고, 게임실행을 하지 않고도 결과를 볼수 있기 때문에 개발중에는 이처럼 SWF를 사용하는 것이 맞다. 그러나 배포판에서 SWF 컨텐츠를 포함하는 것은 SWF 파일을 로딩하는데 프로세스에 과부하가 걸리고 로딩 시간에 영향을 주기 때문에 적절하지 않지 않다.

GFxExport 는 SWF 파일을 간략한 로딩을 위해 최적화 된 포맷으로 처리하는 유ти리티다. 전처리를 하는 동안, 게임의 자원 엔진이 이미지를 관리하도록 이미지를 별도의 파일로 추출한다. 이미지는 최적화 된 로딩과 런타임 메모리 절약을 위해 DXT 텍스처 압축이 있는 DDS 파일로 변환될 수 있다. 내장 폰트를 권장하기도 하고 비트맵 폰트를 사용해야 하는 경우 폰트 텍스처를 최적화하여 미리 계산할 수 있다. GFxExport 출력은 옵션을 통해서 압축도 가능하다.

Scaleform 파일을 배포하는 것은 간단하다. GFxExport 유ти리티는 도움말에서 문서화 된 다양한 옵션을 지원한다. d3d9guide.swf 와 fxplayer.swf 파일을 Scaleform 포맷으로 변환해보자.

```
gfixexport -i DDS -c d3d9guideAS3.swf  
gfixexport -i DDS -c fxplayer.swf
```

gfixexport.exe 는 C:\Program Files\Scaleform\\$(GFXSDK) 디렉토리에 있다. 이러한 명령어는 또한 Tutorial\Section7 의 convert.bat 파일에 있다.

-i 옵션은 이미지 포맷을 정의하며 이 경우 DDS다. DDS는 DirectX 플랫폼에선 가장 일반적인 형태인데 그 이유는 DXT 텍스처 압축이 가능하기 때문이며, 일반적으로 4 배 정도 런타임 텍스처 메모리를 절약해준다.

-c 옵션은 압축을 정의한다. Scaleform 파일 내 벡터 및 액션스크립트 컨텐츠만이 압축된다. 이미지 압축은 선택한 이미지 출력 포맷과 DXT 압축 옵션에 따른다.

-share_images 옵션은 서로 다른 SWF 파일 내 동일한 이미지를 찾아내서 하나의 공유 사본만을 로딩함으로써 메모리 사용을 절약한다.

Tutorial\Section8 내의 ShadowVolume은 GFx::Loader::CreateMovie에서 로딩되는 파일명을 변경하도록 수정된 버전이다.

10 다음 단계

이 학습서는 Scaleform 기능의 기본 개요에 관해 다루고 있다. 보다 자세한 내용을 위해 다음 사항을 참고하면 된다.

- 게임 내 플래시에서 렌더투텍스처. Scaleform Player SWF to Texture SDK 예 참고, Gamebryo™ 통합 데모 및 Unreal® Engine 3 통합 데모 참고
- 개발자 센터 [FAQs](#)에서 다루는 성능 및 기타 사항
- 문서 페이지에서 [Scale9Grid Overview](#) 개요의 Scale9 창 문서 지원
- 커스텀 로딩: Scaleform 또는 SWF 파일로부터의 로딩에 뿐만아니라 [GFx::FileOpener](#)를 상송받으면 메모리나 게임 자원 관리자에서 직접 로딩이 가능하다.
- [GFx::ImageCreator](#)를 통한 커스텀 이미지 및 텍스처
- [GFx::Translator](#)를 통한 지역화