Autodesk®
Scaleform®

# Scaleform Integration Tutorial

This document introduces basic Scaleform usage and 3D engine integration through a DirectX 9 example.


Author: Ben Mowery


Version: 3. 09
Last Edited: October 7, 2013


Autodesk®
GAMEWARE

# Copyright Notice

**Autodesk® Scaleform® 4.3**

**© 2013 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, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk 123D, Autodesk Homestyler, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, Beast (design/logo), BIM 360, Built with ObjectARX (design/logo), Burn, Buzzsaw, CADmep, CAiCE, CAMduct, 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, Design Server, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWGX, DXF, Ecotect, ESTmep, Evolver, Exposure, Extending the Design Team, FABmep, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, ForceEffect, Freewheel, GDX Driver, Glue, 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, Map It, Build It, Use It, 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, Revit LT, RiverCAD, Robot, Scaleform, Scaleform GFx, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Socialcam, 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.

How to Contact Autodesk Scaleform:

| | |
|---|---|
| Document | Scaleform 4.3 Integration Tutorial |
| Address | Scaleform Corporation<br>6305 Ivy Lane, Suite 310<br>Greenbelt, MD 20770, USA |
| Website | www.scaleform.com |
| Email | info@scaleform.com |
| Direct | (301) 446-3200 |
| Fax | (301) 446-3199 |

# Table of Contents

# 1 Introduction

Autodesk Scaleform is a high-performance, proven visual user interface (UI) design middleware solution that allows developers to leverage Adobe Flash® Studio to quickly and inexpensively create modern GPU-accelerated animated UI and vector graphics, without learning new tools or processes. Scaleform creates a seamless visual development path from Flash Studio directly into the game UI.

In addition to UI, developers can use Scaleform to display Flash content inside the 3D environment as animating textures mapped onto 3D surfaces or as standalone 3D objects using the 3D capabilities of ActionScript 3 or AS2 3Di. Likewise, 3D objects and video can be displayed inside Flash UI. As a result, Scaleform works well as either a stand-alone UI solution or as a way to enhance an existing front-end game framework.

This tutorial will walk you through the process of installing and using Scaleform. We will enhance the DirectX ShadowVolume SDK sample with a Flash-based UI.

**Note:** Scaleform has already been integrated with most major game engines. Scaleform can be used directly with supported game engines with minimal coding. This guide is primarily targeted at engineers planning to integrate Scaleform with a custom game engine, or to those looking for a technical overview of Scaleform's capabilities.

**Note:** Please make sure to use the latest version of Scaleform when following this tutorial. The tutorial works with Scaleform 4.0 and above.

**Note:** The tutorial may be incompatible with certain older video cards. This is due to the DirectX SDK ShadowVolume code on which the tutorial is based and is not a compatibility issue with Scaleform. Should a "cannot create renderer" error message be encountered when running the tutorial, the source of the problem can be determined by checking if the Scaleform Player application runs successfully.

# 2 Documentation Overview

The latest multilingual Scaleform documentation can be found online in the Scaleform Developer center at: https://gameware.autodesk.com/scaleform/developer/?action=dl. Free registration is required to access the site.

Current documentation can be found here, https://gameware.autodesk.com/scaleform/developer/?action=doc, and includes the following:

- Web-based Scaleform SDK reference documentation
- PDF documentation
- Getting Started With Scaleform 4.3: How to get up and running quickly with Scaleform on various platforms
- Font Overview: Describes the font and text rendering system and provides details on how to configure both the art assets and Scaleform C++ APIs for internationalization.
- XML Overview: Describes the XML support available in Scaleform.
- Scale9 Grid: Explains how to use Scale9Grid functionality to create resizable windows, panels, and buttons.
- IME Configuration: Describes how to integrate Scaleform's IME support into end-user applications and how to create custom IME text input window styles in Flash.
- ActionScript Extensions: Covers Scaleform ActionScript extensions.

# 3 Installation and Build Dependencies

## *3.1 Installation*

Download the most recent Scaleform installers for Windows. Run it and keep the default installation paths and options. DirectX should be updated if necessary by the Scaleform installer. Scaleform will be installed to the C:\Program Files\Scaleform\GFx SDK 4.3 directory.

The layout is as follows:

**3rdParty**
External libraries required by Scaleform such as libjpeg and zlib.

**Projects**
Visual Studio projects to build the SDK, Scaleform Player and other applications.

**Apps\Samples**
Sample source code for the Scaleform player and other demos.

**Apps\Tutorial**
The source code and projects for this tutorial

**Bin**
Contains pre-built sample binaries and sample Flash content:

| | |
|---|---|
| FxPlayer: | Simple Flash text HUD. |
| Samples: | Various Flash FLA source samples, including UI elements such as buttons, edit boxes, keypads, menus, and spin counters. |
| Video Demo: | Sample Scaleform Video demo and files. |
| Win32: | Pre-compiled binaries for Scaleform Player and demo applications. |
| AMP: | Flash files for running AMP tool. |
| GFxExport: | GFxExport is a pre-processing tool that accelerates loading of Flash content and is covered in detail in section 7. |

**Note:** Re-building the 'Scaleform 4.3 SDK with demos.sln' Visual Studio projects will replace the binaries in the Win32 directory.

**Include**
Scaleform convenience headers.

**Lib**
Scaleform libraries.

**Resources**
CLIK components and tools.

**Doc**
PDF documentation described in section 2.

**Src**
Source code (if available) and related include files.

## *3.2 Compile the Demos*

In order to verify your system is configured properly to build Scaleform, first build the demo solutions found C:\Program Files (x86)\Scaleform\GFx SDK 4.3\Projects\Win32\Msvc90\Demos\GFx 4.3 Demos.sln. Once the .sln file is open in Visual Studio, choose the D3D9_Debug_Static configuration and build the Scaleform Player project.

Check the modification date of the GFx SDK 4.3\Bin\Win32\Msvc90\GFxPlayer\GFxPlayer_D3D9_Debug_Static.exe file to confirm it was successfully rebuilt and run the program.

This program and the other Scaleform Player D3Dx applications on the Start → All Programs → Scaleform → GFx SDK 4.3 → Demo Examples folder are hardware-accelerated SWF players. During development, Scaleform Flash playback can be tested and benchmarked with these tools.

## *3.3 Benchmark SWF Playback in Scaleform Player*

Run the Start → All Programs → Scaleform → GFx SDK 4.3 → GFx Players → GFx Player D3D9 program. Open the C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Data\AS2\Samples\SWFToTexture folder and drag 3DWindow.swf onto the application.

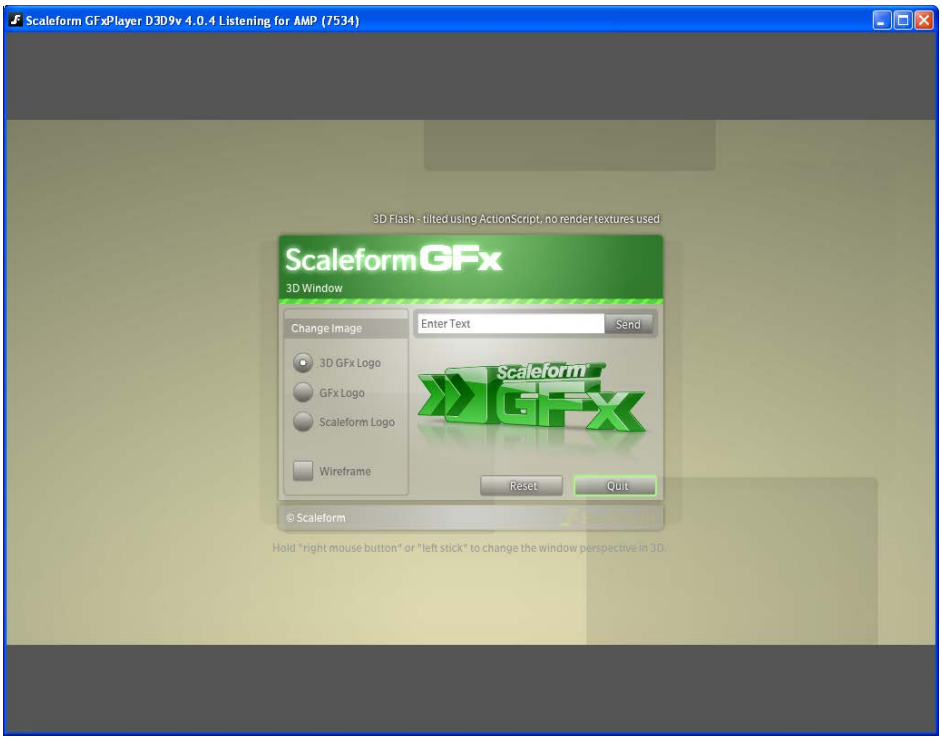


**Figure 1:** Hardware-accelerated playback of sample Flash content

Press the F1 key for the help screen. Try the following options:

1. Press CTRL+F to measure performance. The current FPS will appear in the title bar.
2. Press CTRL+W to toggle wireframe mode. Notice how Scaleform converted the Flash content into triangles for optimized hardware playback. Press CTRL+W again to leave wireframe mode.
3. To zoom in, hold down the CTRL key and the left mouse button, then move the mouse up and down.
4. To pan the image, hold down the CTRL key and the right mouse button, then move the mouse.
5. Press CTRL+Z to return to the normal view.
6. Press CTRL+U to toggle full screen playback.
7. Press F2 for statistics on the movie, including memory usage.

Notice how curved edges like the corners of buttons look sharp even when viewed closely. As the window is made larger or smaller, the Flash content scales. One advantage of vector graphics is that content scales to any resolution. Traditional bitmap graphics require one set of bitmaps for 800x600 screens and another for 1600x1200 screens.

Scaleform also supports traditional bitmap graphics, as can be seen in the “Scaleform” logo in the right center of the screen. Almost any content that an artist can create in Flash can be rendered by Scaleform.

Zoom in on the "3D Scaleform Logo" radio button and switch to wireframe mode by pressing CTRL+W. Notice that the circle has been tessellated into triangles. Press CTRL+A several times to toggle the anti-aliasing mode. In the Edge Anti-Aliasing mode (EdgeAA), additional sub-pixel triangles are added around the edges of the circle to create an anti-aliasing effect. This is typically more efficient than the video card's full-screen anti-aliasing (FSAA), which requires four times the framebuffer video memory and four times the pixel rendering in order to perform AA. Scaleform's proprietary EdgeAA technology leverages the object's vector representation to apply anti-aliasing to only those areas of the screen that can benefit most, typically curved edges and large text. Although the triangle count will increase, the performance impact is manageable as the number of draw primitives (DP) remains constant. It is typically more efficient than using the video card's anti-aliasing function, and EdgeAA as well as other quality settings can be disabled and adjusted.

The Scaleform Player tools are useful for debugging Flash content and performance benchmarking. Open task manager and look at the CPU usage. The CPU usage will likely be high, as Scaleform is rendering as many frames per second as it can for benchmarking purposes. Press CTRL+Y to lock the frame rate to the display refresh rate (typically 60 frames per second). Notice that the CPU usage declines significantly.

**Note:** When benchmarking your own application, make sure to run a release build, as debug Scaleform builds do not provide optimal performance.

## *3.4 Compile the Sample Base*

For Scaleform 4.1, open the Tutorial solution under Scaleform\GFx SDK 4.1\Apps\Tutorial. You can find the solutions for Visual Studio 2005 and Visual Studio 2008 in the windows start menu in Scaleform→ GFx SDK 4.1→ Tutorial. Make sure the project configuration is set to "Debug" and run the application.


**Figure 2:** The ShadowVolume application with the default UI

## *3.5 Scaleform Build Dependencies*

When creating a new Scaleform project, there are some Visual Studio settings that need to be configured prior to compiling. The tutorial already has relative paths in place that you can look at for reference when following the steps below. Keep in mind that $(GFXSDK) is an environment variable defined to the base SDK installation directory. The default location is C:\Program Files\Scaleform\GFx SDK 4.3\; if your libs are in a different location, you will need to replace the paths containing $(GFKSDK) to point to the location of the libs on your system. We use "Msvc90" for these examples; if you use Visual Studio 2008 or Visual Studio 2010, you will need to use Msvc90 or Msvc10 respectively.

Add Scaleform to the project's include paths for both the debug and release build configurations:

$(GFXSDK)\Src

$(GFXSDK)\Include

by pasting them into the Visual Studio "Additional Include Directories" field.

If you are using AS3, make sure to directly include the mandatory AS3 class registration file "GFx/AS3/AS3_Global.h" in your application. Developers can customize this file to exclude unneeded AS3 classes. See the document Scaleform LITE Customization for more details.

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)

**Note:** Change Msvc90 to the string corresponding to your version of Visual Studio.

Finally, add the Scaleform libraries and their dependencies:

libgfx.lib
libgfx_as2.lib
libgfx_as3.lib
libgfx_air.lib
libgfxexpat.lib
libjpeg.lib
zlib.lib
libcurl.lib
libpng.lib
libgfxrender_d3d9.lib
libgfxsound_fmod.lib

Make sure the sample application still compiles and links in both the debug and release configurations. For reference, a modified .vcproj file with the Scaleform include and linker settings is in the Tutorial\Section3.5 folder.

### 3.5.1 AS3_Global.h and Obj\AS3_Obj_Global.xxx files

Developers must note that AS3_Global.h and Obj\AS3_Obj_global.xxx are **completely unrelated** files. AS3_Obj_Global.xxx files contain implementation of so called "global" ActionScript 3 objects. Every swf file contains at least one object called "script", which is a global object. There is also a class GlobalObjectCPP, which is a global object for all classes implemented in C++. This is specific to the Scaleform VM implementation.

AS3_Global.h has a completely different purpose. This file contains the ClassRegistrationTable array. The purpose of this array is to reference C++ classes implementing corresponding AS3 classes. Without this reference, code will be excluded by a linker. So, ClassRegistrationTable has to be defined in your executable, otherwise you will get a linker error. Each of our demo players includes AS3_Global.h for this reason.

The whole purpose of putting the ClassRegistrationTable into an include file and requiring developers to include it, is to allow customization of the ClassRegistrationTable (for the purpose of potential code size reduction). The best way to do that is to make a copy of AS3_Global.h, comment out un-needed classes (after which they will not be linked in), and include the customized version into your app.

However, there is a catch related to this optimization. Because name resolution in the AS3 VM happens at run-time, it is possible that a needed class will not be found if commented out of the table. So, if you want to get rid of "unnecessary" classes, please make sure that your app is functional after that.

# 4 Game Engine Integration

Scaleform 4.3 integrates seamlessly in most major 3D game engines including: Unreal® Engine 3, Gamebryo™, Bigworld®, Hero Engine™, Touchdown Jupiter Engine™, CryENGINE™, and Trinigy Vision Engine™. Little or no coding is required to leverage Scaleform in games developed with these engines.

This section describes how to integrate Scaleform with a custom DirectX application. The DirectX ShadowVolume SDK sample is a standard DirectX application and has application and game loops similar to a typical game. As seen in the previous section, the application renders a 3D environment with a DXUT-based 2D UI overlay.

This tutorial will walk through the process of integrating Scaleform into the application to replace the default DXUT user interface with a Flash-based Scaleform interface.

The DXUT framework does not expose the underlying render loop to the application, instead exposing callbacks to abstract low level details. To understand how these steps relate to a standard Win32 DirectX render loop, compare with the GFxPlayerTiny.cpp sample that comes with the Scaleform SDK.

## *4.1 Rendering Flash*

The first step in the integration process is to render a Flash animation on top of the 3D background. This involves instantiating a GFx::Loader object to manage loading of all Flash content globally for the application, a GFx::MovieDef to contain the Flash content, and a GFx::Movie to represent a single playing instance of the movie. Additionally a Render::Renderer2D object and a Render::HAL object will be instantiated to act as the interface between Scaleform and the implementation-specific rendering API, in this case DirectX. We also discuss how to cleanly deallocate resources, respond to lost device events, and handle fullscreen/windowed transitions.

A version of ShadowVolume modified with this section's changes can be found in Tutorial\Section4.1. The code shown in this document is for illustration and is not complete.

### Step #1: Add Header Files

Add the required header files to ShadowVolume.cpp:

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

Several Scaleform objects are required to render video and will be kept together in a new class added to the application, GFxTutorial. In addition to making the code cleaner, keeping Scaleform state together in a class has the advantage that a single delete call will free all Scaleform objects. The interaction of these objects will be described in detail in the steps below.

```
// 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;
```

### Step #2: Initialize GFx::System

The first stage of Scaleform initialization is to instantiate a GFx::System object to manage Scaleform memory allocation. In WinMain we add the lines:

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

The GFx::System object must come into scope before the first Scaleform call and cannot leave scope until the application is finished using Scaleform which is why it is placed in WinMain. GFx::System as instantiated here uses Scaleform's default memory allocator but can be overridden with an application's custom memory allocator. For the purposes of this tutorial it is sufficient to simply instantiate GFx::System and take no further action.

GFx::System must leave scope before the application terminates, meaning that it should not be a global variable. In this case it will go out of scope when the GFxTutorial object is freed.

Depending on the structure of your particular application it may be easier to call the GFx::System::Init() and GFx::System::Destroy() static functions instead of creating the GFx::System object instance.

### Step #3: Loader and Renderer Creation

The remainder of Scaleform initialization will be performed right after the application's WinMain does its own initialization in InitApp(). Add the following code right after the call to InitApp():

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

GFxTutorial contains a GFx::Loader object. An application typically has only one GFx::Loader object, which is responsible for loading the SWF/GFx content and storing this content in a resource library, enabling resources to be reused in future references. Separate SWF/GFx files can share resources such as images and fonts saving memory. GFx::Loader also maintains a set of configuration states such as GFx::Log, used for debug logging.

The first step in GFxTutorial::InitGFx() is to set states on GFx::Loader. GFx::Loader passes debug tracing to the handler provided by SetLog. Debug output is very helpful when debugging, since many Scaleform functions will output the reason for failure to the log. In this case we use the default Scaleform PlayerLog handler, which prints messages to the console window, but integration with a game engine's debug logging system can be accomplished by subclassing GFx::Log.

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

GFx::Loader reads content through the GFx::FileOpener class. The default implementation reads from a file on disk, but custom loading from memory or a resource file can be accomplished by subclassing GFx::FileOpener.

```
// Give the loader the default file opener
Ptr<FileOpener> pfileOpener = *new FileOpener;
gfxLoader->SetFileOpener(pfileOpener);
```

Render::HAL is a generic interface that enables Scaleform to output graphics to a variety of hardware. We create an instance of the D3D9 renderer. The renderer object is responsible for managing the D3D device, textures, and vertex buffers used by Scaleform. Later on, in HAL::InitHAL, we will pass the Render HAL an IDirect3Device9 pointer initialized by the game, so that Scaleform can create DX9 resources and successfully render UI content.

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

The above code uses the default Render::D3D9::HAL object supplied with Scaleform. Subclassing Render::HAL enables better control over Scaleform's rendering behavior and can result in a tighter integration.

### Step #4: Load a Flash Movie

Now the GFx::Loader is ready to load a movie. Loaded movies are represented as GFx::MovieDef objects. The GFx::MovieDef encompasses all of the shared data for the movie, such as the geometry and textures. It does not include per-instance information, such as the state of individual buttons, ActionScript variables, or the current movie frame.

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

The LoadKeepBindData flag maintains a copy of texture images in system memory, which may be useful if the application will re-create the D3D device. This flag is not necessary on game console systems or under conditions where it is known that textures will not be lost.

LoadWaitFrame1 instructs CreateMovie not to return until the first frame of the movie has been loaded. This is significant if GFx::ThreadedTaskManager is used.

The last argument is optional and specifies the memory arenas to be used. Please refer to the Memory System Overview document for information on creating and using memory arenas.

### Step #5: Movie Instance Creation

Before rendering a movie, a GFx::Movie instance must be created from the GFx::MovieDef object. GFx::Movie maintains state associated with a single running instance of a movie such as the current frame, time in the movie, states of buttons, and ActionScript variables.

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

The first argument to CreateInstance determines whether the first frame is to be initialized. If the argument is false, we have the opportunity to change Flash and ActionScript state before the ActionScript first frame initialization code is executed. The second argument is optional and specifies the memory arenas to be used. Please refer to the Memory System Overview document for information on creating and using memory arenas.

Once the movie instance is created, the first frame is initialized by calling Advance(). This is only necessary if false was passed to CreateInstance.

```
// 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();
```

The first argument to Advance is the *difference* in time, in seconds, between the last frame of the movie and this frame. The current system time is recorded to enable calculation of the time difference between this frame and the next.

In order to alpha blend the movie on top of the 3D scene:

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

Without the above call, the movie will render but will cover the 3D environment with a background stage color specified by the Flash file.

### Step #6: Device Initialization

Scaleform must be given the handle to the DirectX device and presentation parameters in order to render. These values are wrapped in the Render::D3D9::HALInitParams structure and passed to the Render::D3D9::HAL::InitHAL function, called after the D3D device is created and before Scaleform is asked to render. InitHAL should be called again if the D3D device handle changes, which can occur on window resizes or fullscreen/windowed transitions.

ShadowVolume's OnResetDevice function is called by the DXUT framework after initial device creation and also after device reset. The following code is added to the corresponding OnResetDevice method in GFxTutorial:

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

The InitHAL call passes the D3D device information to Scaleform. The HAL_NoSceneCalls flag specifies that no DirectX BeginScene() or EndScene() calls will be made by Scaleform. This is necessary because the ShadowVolume sample already makes those calls for the application in the OnFrameRender callback.

### Step #7: Lost Devices

When the window is resized or the application is switched to fullscreen, the D3D device will be lost. All D3D surfaces including vertex buffers and textures must be reinitialized. ShadowVolume releases surfaces in the OnLostDevice callback. Render::HAL can be informed of the lost device and given a chance to free its D3D resources in the corresponding OnLostDevice method in ScaleformTutorial:

```
pRenderHAL->ShutdownHAL();
```

This and the previous step explained initialization and lost devices based on the DXUT framework's callback system. For an example of a basic Win32/DirectX render loop, see the GFxPlayerTiny.cpp example with the Scaleform SDK.

### Step #8: Resource Allocation and Cleanup

Because all Scaleform objects are contained in the GFxTutorial object, cleanup is as simple as deleting the GFxTutorial object at the end of WinMain:

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

The other consideration is the cleanup of DirectX 9 resources such as vertex buffers. This is taken care of by Scaleform, but for allocation and cleanup that occurs during the main game loop it is important to understand the role InitHAL() and ShutdownHAL() play.

In the DirectX 9 implementation, InitHAL allocates D3DPOOL_DEFAULT resources, including vertex buffer caches. When integrating with your own engine try to place the call to InitHAL in a location appropriate for allocating D3DPOOL_DEFAULT resources.

ShutdownHAL will free the D3DPOOL_DEFAULT resources. Applications that use the DXUT framework, including ShadowVolume, should allocate D3DPOOL_DEFAULT resources in the DXUT OnResetDevice callback and free resources in the OnLostDevice callback. The ShutdownHAL call in GFxTutorial::OnLostDevice matches the InitHAL call in GFxTutorial::OnResetDevice.

When integrating with your own engine, try to call InitHAL and ShutdownHAL together with any other calls to create and free engine D3DPOOL_DEFAULT resources.

### Step #9: Setting the Viewport

The movie must be given a certain viewport on the screen to render into. In this case, it occupies the entire window. Since the screen resolution can change, we reset the viewport every time the D3D device is reset by adding the following code to GFxTutorial::OnResetDevice:

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

The first two parameters to SetViewport specify the size of the framebuffer used, typically the size of the window for PC applications. The next four parameters specify the size of the viewport within the framebuffer that Scaleform is to render into.

The framebuffer size arguments are provided for compatibility with OpenGL and other platforms that may use different orientation of coordinate systems or not provide a way to query the framebuffer size.

Scaleform provides functions to control how Flash content is scaled and positioned within the viewport. We will examine these options in section 4.2 after the application is ready to run.

### Step #10: Rendering into the DirectX Scene

Rendering is performed in ShadowVolume's OnFrameRender() function. All D3D rendering calls are made between the BeginScene() and EndScene() calls. We'll call GFxTutorial::AdvanceAndRender() before the EndScene() call.

```
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 moves the movie forward by deltaTime seconds. The speed at which the movie is played is controlled by the application based on the current system time. It is important to provide real system time to GFx::Movie::Advance to ensure the movie plays back correctly on different hardware configurations.

### Step #11: Preserving Rendering States

Render::Renderer2D::Display makes DirectX calls to render a frame of the movie on the D3D device. For performance reasons, various D3D device states, such as blending modes and texture storage settings, are not preserved and the state of the D3D device will be different after the call to Render::Renderer2D::Display. Some applications may be adversely affected by this. The most straightforward solution is to save device state before the call to Display and restore it afterwards. Greater performance can be achieved by having the game engine re-initialize its required states after Scaleform rendering. For this tutorial we simply save and restore state using DX9's state block functions.

A DX9 state block is allocated for the life of the application and used before and after the calls to GFxTutorial::AdvanceAndRender():

```
// 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();
```

### Step #12: Disable Default UI

The final step is to disable the original DXUT-based UI. This is done by commenting out the relevant blocks of code and the final result is in Section4.1\Shadowvolume.cpp. Diff it with the previous section's code to see the changes. All the changes related to DXUT are marked with comments:

```
// Disable default UI
...
```

We now have a hardware-accelerated flash movie rendering in our DirectX application.


**Figure 3: The ShadowVolume application with a Scaleform Flash-based UI**

## *4.2 Scaling Modes*

The calls to GFx::Movie::SetViewport keep the viewport dimensions equal to the screen resolution. If the aspect ratio of the screen is different than the native aspect ratio of the Flash content the interface may become distorted. Scaleform provides functions to:

- Maintain the aspect ratio of content or allow it to stretch freely.
- Position content relative to the center, corners, or side of the viewport.

These functions are very useful for rendering the same content on both 4:3 and widescreen displays. One of the advantages of Scaleform is that scalable vector graphics enable content to resize freely to match any display resolution. Traditional bitmap graphics typically require artists to create large and small versions of bitmaps for different screen resolutions (e.g., one set for low resolution 800x600 displays and another set for high resolution 1600x1200 displays). Scaleform enables the same content to scale to any resolution. Additionally, traditional bitmap graphics are fully supported for those game elements where bitmaps are more appropriate.

GFx::Movie::SetViewScaleMode defines how scaling will be performed. To ensure the movie fits in the viewport without affecting the original aspect ratio the following call can be added to the end of GFxTutorial::InitGFx() along with the other calls to setup the GFx::Movie object:

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

The four possible arguments to SetViewScaleMode are covered in the online documentation and are:

| | |
|---|---|
| SM_NoScale | The size of the content is fixed to the native resolution of the Flash stage. |
| SM_ShowAll | Scales the content to fit the viewport while maintaining the original aspect ratio. |
| SM_ExactFit | Scales the content to fill the entire viewport without regard to the original aspect ratio. The viewport will be filled, but distortion may occur. |
| SM_NoBorder | Scales the content to fill the entire viewport while maintaining the original aspect ratio. The viewport will be filled, but some clipping may occur. |

The complementary SetViewAlignment controls the position of the content relative to the viewport. When the aspect ratio is maintained, some part of the viewport may be empty when SM_NoScale or SM_ShowAll are selected. SetViewAlignment decides where to position the content within the viewport. In this case, the interface buttons should be centered vertically on the far right of the screen:

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

Try changing the arguments to SetViewScaleMode and SetViewAlignment to see how the application behavior changes when the window is resized.

The SetViewAlignment function does not have any effect except when SetViewScaleMode is set to the default of SM_NoScale. For more complex alignment, scaling, and positioning requirements, GFx supports ActionScript extensions that enable the movie to choose its own size and position. The scale and alignment parameters can also be set through ActionScript instead of C++. SetViewScaleMode and SetViewAlignment modify the same properties represented by the ActionScript Stage class (Stage.scaleMode, Stage.align).

## *4.3 Processing Input Events*

Now that ShadowVolume's rendering pipeline has been modified to render Flash with Scaleform, we now want to interact with the playing Flash. For example, moving the mouse over a button should cause it to highlight and typing into a text box should cause new characters to appear.

The GFx::Movie::HandleEvent passes a GFx::Event object representing the type of event and other information such as the key pressed or mouse coordinates. The application simply constructs an event based on input and passes it to the appropriate GFx::Movie.

### 4.3.1 Mouse Events

ShadowVolume receives Win32 input events in the MsgProc callback. A call is added to GFxTutorial::ProcessEvent to run code to enable Scaleform to process the events. The code below processes WM_MOUSEMOVE, WM_LBUTTONDOWN, and WM_LBUTTONUP:

```
void ProcessEvent(HWND hWnd, UINT 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 expects mouse coordinates to be relative to the upper left corner of the specified viewport, not the native resolution of the movie. The below examples clarify this:

Example #1: Viewport matches screen dimensions

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

No transformation is necessary in this case: the mouse coordinates from Windows are already relative to the upper left corner of the movie since the movie is positioned at (0, 0). The coordinates are scaled internally by Scaleform from the viewport dimensions to the native movie resolution for internal processing.

Example #2: Viewport smaller than screen, but viewport is positioned in the upper left corner of the screen

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

Once again, no transformation is necessary in this case. The size and position of the buttons changes because the viewport has been scaled down. However, both coordinates used by HandleEvent and the Windows screen coordinates are still relative to the upper left corner of the window and no translation is necessary. Scaling of the coordinates from the viewport dimensions to the native movie resolution is handled internally by Scaleform.

Example #3: Viewport smaller than screen and centered

```
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);
```

Translation of the Windows screen coordinates is necessary in this case. The movie is no longer positioned at (0, 0) so its new position at (screen_width / 2 – movie_width / 2, screen_height / 2 – movie_height / 2) must be subtracted from the screen coordinates passed in by Windows.

Note that if the Flash content is centered or in some other way aligned by GFx::Movie::SetViewAlignment these transformations do not have to be performed. As long as the mouse coordinates are relative to the coordinates given to GFx::Movie::SetViewport, alignment and scaling performed by SetViewAlignment and SetViewScaleMode will be handled internally by Scaleform.

### 4.3.2 Keyboard Events

Keyboard events are also handled through GFx::Movie:HandleEvent. There are two kinds of key events: GFx::KeyEvent and GFx::CharEvent:

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

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

A GFx::KeyEvent is similar to a raw scan code; a GFx::CharEvent is similar to a processed ASCII character. In Windows, a GFx::CharEvent would be generated in response to the WM_CHAR message; GFx::KeyEvents are generated in response to WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN, and WM_KEYUP messages. Some examples:

- The 'c' key is pressed down while the SHIFT key is held down: A GFx::KeyEvent should be generated in response to the WM_KEYDOWN message to indicate that:
  - The 'c' key was pressed, and the scan code of that key;
  - The key was pressed down; and
  - The SHIFT key is active.

- At the same time a GFx::CharEvent should be fired in response to the WM_CHAR message to pass the "cooked" ASCII value 'C' to Scaleform.

- Once the 'c' key is released, a GFx::KeyEvent should be sent in response to the WM_KEYUP message. No GFx::CharEvent need be sent when a key is released.

- The F5 key is pressed: A GFx::KeyEvent is sent when the key goes down, and a second event when the key goes back up. It is not necessary to send a GFx::CharEvent because F5 does not correspond to a printable ASCII code.

Separate GFx::KeyEvent events are sent for key down and key up events. To enable platform independence, the key code is defined in GFx_Event.h to match the key codes used internally by Flash. The GFxPlayerTiny.cpp example and Scaleform Player program both contain code to convert Windows scan codes to the corresponding Flash codes. The final code for this section includes the ProcessKeyEvent function that can be reused when integrating with a custom 3D engine:

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

Simply call the function from the Windows WndProc function in response to WM_CHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN, and WM_KEYUP messages. The appropriate Scaleform events will be generated and sent to pMovie.

Sending both GFx::KeyEvent and GFx::CharEvent is important. For example, most text boxes respond only to GFx::CharEvent because they are interested in printable ASCII characters. Also, a text box should be able to accept Unicode characters (e.g., from a Chinese Input Method Editor [IME]). In the case of IME input, raw key codes are not useful and only the final character event (which typically results from several keystrokes processed by the IME) is of interest to the text box. In contrast, a list box control would need to intercept the Page Up and Page Down keys through GFx::KeyEvent because these keys do not correspond to printable characters.

The function is included with the final code for this section in Tutorial\Section4.3. Run the program and move the mouse over buttons. Buttons will highlight correctly, and those that do not require integration with the 3D engine will work properly. Pressing "Settings" will transition to the DX9 configuration screen without any C++ code because d3d9guideAS3.fla implements this simple logic using ActionScript in HUDMgr.as.

To see the keyboard processing code in action click "Change Mesh" and type into the text input box. There are some minor issues which will be fixed later on in the tutorial. Notice the animation that occurs when the "Change Mesh" button is pressed to open a text input box. This animation is easy to do in Flash with vector graphics, but impractical with a traditional bitmap-based interface. The animation would require custom code in addition to additional bitmaps, making the animation potentially slow to load, costly to render, and most importantly tedious to code.

### 4.3.3 Hit Testing

Run the application and move the mouse while holding down the left mouse button to change the direction the camera is pointing in the 3D world. Now move the mouse over one of the UI elements and do the same. Although the mouse click does generate the desired response in the interface, it still causes the camera to move.

Focus control between the UI and the 3D world is a problem that can be addressed through GFx::Movie::HitTest. This function determines whether viewport coordinates hit an element rendered in the Flash content. Modify GFxTutorial::ProcessEvent to call HitTest after processing a mouse event. If the event occurred over a UI element, it signals the DXUT framework not to pass the event to the camera for processing:

```
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 Keyboard Focus

Run the application and click the "Change Mesh" button to open a text input box. Type into the box and keyboard input will work because of the code added in section 4.3.2. However, entering the W, S, A, D, Q, and E keys will enter text but also move the camera in the 3D world. The keyboard event is processed by Scaleform, but also passed to the 3D camera.

Resolving this issue requires determining whether the text input box has focus. Section 6.1.2 will describe how Flash ActionScript can be used to send events to C++ enabling our event handler to track focus.

### 4.3.5 Touch Events

Touch events are like mouse events, but sent only when the user is touching the screen. To make use of touch events, we can route raw touch data from Windows API directly into Scaleform. The minimum required platform is Windows 7 running on a touch-capable system, without which you will be unable to follow this section of the tutorial and should skip ahead to the next section.

Before compiling, add **WINVER=0x601** to the project's Preprocessor Definitions. This tells the Windows API to compile functionality specific for Windows 7 (i.e. multitouch). In ShadowVolume.cpp, uncomment the macro `#define ENABLE_MULTITOUCH`, which guards unsupported code from compiling.

`#include <windows.h>` has been added in order to use the special multitouch libraries. During the window's initialization we register our intent to listen for touch events via RegisterTouchWindow, which enables reception of WM_TOUCH messages in the MsgProc callback for processing.

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

ProcessTouchEvent is our function responsible for parsing and relaying the WM_TOUCH message. Windows functions are used to convert the touch coordinates into pixel coordinates. Afterwards, a GFx::TouchEvent is created and sent to the movie via HandleEvent. The GFx::TouchEvent class is composed of a unique ID (managed by the OS), x/y coordinates, contact area, pressure (from 0..1), and whether or not it's a primary point.

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

The SWF movie is now receiving touch events, but GFx by default supports 0 max touch points. The last step is to define an inherited MultitouchInterface that describes our multitouch environment, specifically the max number of supported touch points and whether we'll also be relaying GFx::GestureEvent's (this tutorial doesn't cover native gesture events, but you can use the same principles outlined in this section in order to expose them to Scaleform).

```
class FxPlayerMultitouchInterface : public MultitouchInterface
{
public:
    // Return maximum number of touch points supported by hardware
    virtual unsigned GetMaxTouchPoints() const { return 2; }

    // Return a bit mask of supported gestures (none at the moment)
    virtual UInt32   GetSupportedGesturesMask() const { return 0; }

    // Is multitouch supported?
```

```
    virtual bool     SetMultitouchInputMode(MultitouchInputMode) { return
true; }
};
```

Once we send a reference of this interface class to our movie via SetMultitouchInterface, touch events are successfully propagated to the movie.

# 5 Introduction to Localization and Fonts

## *5.1 Font Overview and Capabilities*

Scaleform provides an efficient and flexible font and localization system. Multiple fonts, point sizes, and styles can be simultaneously rendered efficiently and with a low memory footprint. Font data can be obtained from embedded Flash fonts, shared font libraries, the operating system, and directly from TTF font libraries. Vector-based font compression reduces the memory footprint of large Asian fonts. Font support is fully cross platform and works equally well on console systems, Windows, and Linux. Full documentation on Scaleform's font and internationalization capabilities can be found on the Developer center documentation page: https://gameware.autodesk.com/scaleform/developer/?action=doc.

Typical font solutions involve rendering each character of an entire font to a texture and then characters would be texture mapped from the font texture to the screen as needed. Additional textures would be required for different font sizes and styles. For Latin characters the memory usage is acceptable, but rendering an Asian font with 5000 glyphs to a texture is impractical. A large amount of valuable texture memory is required as well as processing time to render each glyph. Simultaneously rendering different font sizes and styles is out of the question.

Scaleform solves this problem with a dynamic font cache. Characters are rendered on demand to the cache, and slots in the cache are replaced when necessary. Different font sizes and styles can share a single public cache using Scaleform's intelligent glyph-packing algorithm. Scaleform works with vector fonts, meaning only a single TTF font needs to be stored in memory to render crisp and clear characters of any size. Finally, Scaleform supports "fake italic" and "fake bold" functionality, enabling a single regular font's vector representation to be transformed to italic or bold on demand, saving additional memory.

Very large text can be rendered directly as tessellated triangles. This is useful for small quantities of large text, as might be found in a game's title screen. Find the font configuration example in Bin\Data\AS2\Samples\FontConfig and drag the sample.swf file onto an open Scaleform Player D3D9 window.

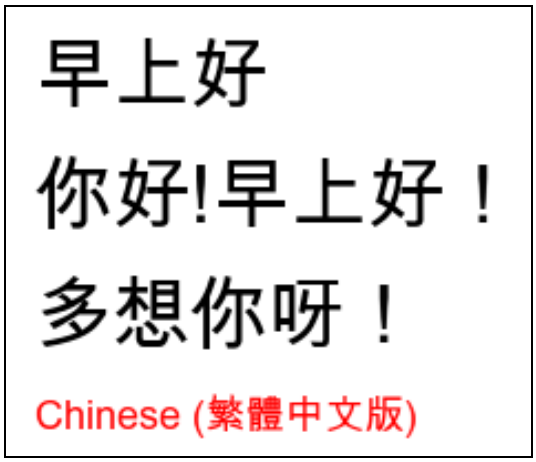


**Figure 4a:** Small Chinese characters

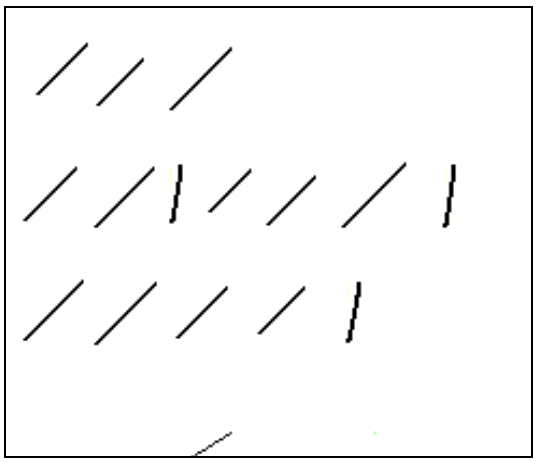

**Figure 4b:** Wireframe representation

Pressing CTRL+W to view the wireframe representation of these characters shows that each character is represented as two texture mapped triangles. Since these are smaller characters it is more efficient to render them as bitmaps through the dynamic font cache.

Increase the size of the window while staying in wireframe mode. The characters will switch from rendering with texture maps to rendering as solid color triangles:


**Figure 4c:** Large Chinese character

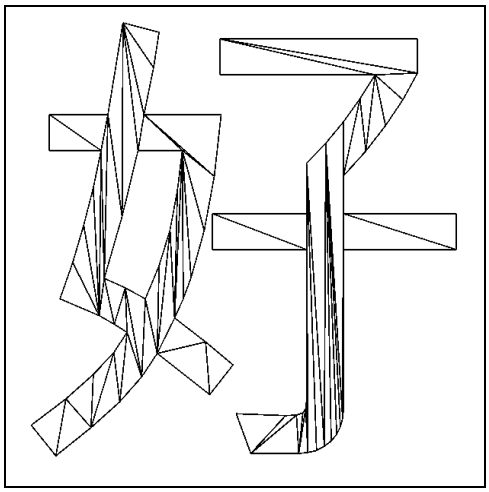

**Figure 4d:** Wireframe representation

For large characters, it is more efficient to render as solid color triangles. Operating on large bitmaps is costly due to excessive memory bandwidth usage. Only pixels that require color are set, avoiding wasting processing power on the empty areas of the character. In figures 4c and 4d, Scaleform detected the size of the character passed a certain threshold and tessellated it into triangles, rendering the character as geometry instead of as a bitmap.

Font soft shadow, blur, and other effects are supported. Simply set the appropriate filters on the text field in Flash to generate the desired effect. Additional details are in the *Font and Text Configuration Overview* linked to above. Examples can be downloaded from the developer center as gfx_2.1_texteffects_sample.zip:


**Figure 5:** Text effects

# 6 Interfacing C++, Flash, and ActionScript

Flash's ActionScript scripting language enables creation of interactive movie content. Events such as clicking a button, reaching a certain frame, or loading a movie can execute code to dynamically change movie content, control the flow of the movie, and even launch additional movies. ActionScript is powerful enough to create full mini-games entirely in Flash. Like most programming languages, ActionScript supports variables and subroutines. Scaleform provides a C++ interface to directly manipulate ActionScript variables and arrays, as well as directly invoke ActionScript subroutines. Scaleform also provides a callback mechanism that enables ActionScript to pass events and data back to the C++ program.

## *6.1 ActionScript to C++*

Scaleform provides two mechanisms for the C++ application to receive events from ActionScript: FSCommand and ExternalInterface. Both FSCommand and ExternalInterface register a C++ event handler with GFx::Loader to receive event notification. FSCommand events are triggered by the ActionScript fscommand function, receive two string arguments, and cannot return a value. ExternalInterface events are triggered when ActionScript calls the flash.external.ExternalInterface.call function, receive a list of any number of typed GFx::Value arguments (covered in section 6.1.2), and can return a value to the caller.

Due to limited flexibility, FSCommands are made obsolete by ExternalInterface and are no longer recommended for use. They are described here for completeness and because fscommands may be encountered in legacy code. Additionally, gfxexport can generate a report of all fscommands used in a SWF file with the –fstree, –fslist and –fsparams options. This is not possible with ExternalInterface and in some cases may be sufficient reason to use fscommands.

### 6.1.1 FSCommand Callbacks

The ActionScript fscommand function passes a command and a data argument to the host application. Typical usage in ActionScript would be something like:

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

Any non-string arguments to fscommand, such as Booleans or integers, will be converted to strings. ExternalInterface can directly receive integer arguments.

This passes two strings to the GFx FSCommand handler. An application registers a fscommand handler by subclassing GFx::FSCommandHandler and registering the class with either the GFx::Loader or with individual GFx::Movie objects. If a command handler is set on GFx::Movie, it will receive callbacks for only the

fscommand calls performed in that movie instance. The GFxPlayerTiny example demonstrates this process (search for "FxPlayerFSCommandHandler") and we will add similar code to ShadowVolume. The final code for this section is in Tutorial\Section6.1.

First, subclass GFx::FSCommandHandler:

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

The Callback method receives the two string arguments passed to fscommand in ActionScript as well as a pointer to the specific movie instance that called fscommand.

Next, register the handler after creating the GFx::Loader object in GFxTutorial::InitGFx():

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

Registering the handler with the GFx::Loader causes every GFx::Movie to inherit this handler. SetFSCommandHandler can be called on individual movie instances to override this default setting.

Our custom handler simply prints each fscommand event to the debug console. Run ShadowVolume and click the "Toggle Fullscreen" button. Notice that events are printed whenever a UI event happens:

```
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");
}
```

As an exercise, change fscommand("ToggleFullscreen") to fscommand("ToggleFullscreen", String(hud.visible)) to print the value of the hud.visible value to C++. Export the flash movie (CTRL+ALT+Shift+S) and replace d3d9guideAS3.swf.

The buttons on the UI can be integrated with the ShadowVolume sample by triggering the appropriate code when FSCommand events happen. As an example, add the following lines to the fscommand handler to toggle fullscreen mode:

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

The DXUT function OnFrameMove is called right before a frame is rendered. Add the corresponding code in at the end of the the OnFrameMove callback:

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

In general, event handlers should be non-blocking and return to the caller as soon as possible. Event handles are typically only called during Advance or Invoke calls.

### 6.1.2 ExternalInterface

The Flash ExternalInterface.call method is similar to fscommand but is preferred because it provides more flexible argument handling and can return values.

Registering an ExternalInterface handler is similar to registering an fscommand handler:

```
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:
                          Printf("%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");
        }
};
```

And register the handler with GFx::Loader:

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

An external interface call will be made from ActionScript when the text input box gains or lose focus. 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);
}
```

**Note::** The MeshPathFocus callback is called whenever the text input box gains or lose focus

The ExternalInterface calls will trigger our ExternalInterface handler and pass it the focus state of the text input box (true or false) and the arbitrary command string "MeshPathFocus." Open the text input, click on the text area, and then click on an empty area of the screen to move focus away from the text input. The console output should be similar to:

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

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

The event handler can be modified to detect when focus is gained or lost and pass that information to the GFxTutorial object:

```
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 will only pass keyboard events to the movie if the textbox has focus. If a keyboard event is passed to the textbox, a flag is set to prevent it from being passed to the 3D engine:

```
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;
      }
}
```

Space (ASCII code 32) and tab (ASCII code 9) are always passed through as they correspond to the "Toggle UI" and "Settings" buttons.

In order to enable the user to change the mesh being rendered, they click the "Change Mesh" button to open the text box, enter a new mesh name, and then press enter. When enter is pressed, the text box will invoke an ActionScript event handler, which calls ExternalInterface with the name of the new mesh. The additional code in OurExternalInterfaceHandler::Callback is:

```
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));
}
```

As with the fullscreen toggle, the actual work is done in the DXUT OnFrameMove callback. The code is based on the event handler for the default DXUT interface.

## *6.2 C++ to ActionScript*

Section 6.1 explained how ActionScript can call into C++. This section describes how to communicate in the other direction using Scaleform functions that enable the C++ program to initiate communication with the playing movie. Scaleform supports C++ functions to get and set ActionScript variables as well as invoke ActionScript subroutines.

### 6.2.1 Manipulating ActionScript Variables

Scaleform supports GetVariable and SetVariable, which enable direct manipulation of ActionScript variables. The code in Tutorial\Section6.2 has been modified to load the yellow HUD display (fxplayer.swf) and increment a counter whenever F5 is pressed:

```
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 returns the value of the _root.counter variable. Initially the variable does not exist and GetVariableDouble returns zero. The counter is incremented and the new value saved to _root.counter by way of SetVariable. The online documentation for GFx::Movie lists the different variations of GetVariable and SetVariable.

The fxplayer Flash file has two dynamic text fields that can be set to arbitrary text by the application. The MessageText text field is centered in the middle of the screen and the HUDText variable is positioned in the upper left corner of the screen. A string is set based on the value of the _root.counter variable and SetVariable is used to update the message text.

**Performance note:** The preferred method to change the value of a Flash dynamic text field is to set the TextField.text variable or to call GFx::Movie::Invoke to run an ActionScript routine to change the text (more on this in section 6.2.2). *Do not* bind a dynamic text field to an arbitrary variable and then change that variable to change the text. Although this works, it incurs a performance penalty as Scaleform must check the value of that variable on every frame.


**Figure 8:** SetVariable changing the value of HUD text

The above usage deals with variables directly as strings or numbers. The online documentation describes an alternate syntax using GFx::Value objects to efficiently process variables directly as integers and eliminate the need for string to integer conversion.

SetVariable has an optional third argument of type GFx::Movie::SetVarType that declares the assignment "sticky." This is useful when the variable being assigned has not yet been created on the Flash timeline. For example, suppose that the text field root.mytextfield is not created until frame 3 of the movie. If SetVariable("root.mytextfield.text", "testing", SV_Normal) is called on frame 1, right after the movie is created, then the assignment would have no effect. If the call is made with SV_Sticky (the default value) then the request is queued up and applied once the root.mytextfield.text value becomes valid on frame 3. This makes it easier to initialize movies from C++. Generally SV_Normal is more efficient than SV_Sticky so SV_Normal should be used where possible.

SetVariableArray passes an entire array of variables to Flash in one operation. This function can be used for operations such as dynamically populating a dropdown list control. The following code is placed in GFxTutorial::InitGFx() and sets the value of the _root.SceneData dropdown:

```
// 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);
```

The code in Tutorial\Section6.1 contains additional ExternalInterface handlers to respond to the luminance control, number of lights, type of scene, and other controls present in the original ShadowVolume interface.

**Note:** This example populated the dropdown menus through C++ for illustration purposes. Generally dropdown menus with a static list of choices should be initialized through ActionScript instead of C++.

### 6.2.2 Executing ActionScript Subroutines

In addition to modifying ActionScript variables, ActionScript code can be triggered through the GFx::Movie::Invoke method. This is useful for performing more complicated processing, triggering animation, changing the current frame, programmatically changing the state of UI controls, and dynamically creating UI content such as new buttons or text.

This section uses GFx::Movie::Invoke to programmatically open the "Change Mesh" text input box when F6 is pressed and close it when F7 is pressed. This could not be done by using SetVariable to change state since

animation occurs when the radio buttons are clicked. SetVariable cannot trigger animation, but ActionScript routines called with Invoke can.

GFx::Movie::Invoke can be called to execute the openMeshPath routine in the WM_CHAR keyboard handler:

```
...

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);
}

...
```

One common error in using Invoke is calling an ActionScript routine that is not yet available, in which case an error will be printed to the Scaleform log. An ActionScript routine will not become available until the frame it is associated with has been played or the nested object it is associated with has been loaded. All ActionScript code in frame 1 will be available as soon as the first call to GFx::Movie::Advance is made, or if GFx::MovieDef::CreateInstance is called with initFirstFrame set to true.

This example used the printf style of Invoke. Other versions of the function use GFx::Value to efficiently process non-string arguments. InvokeArgs is identical to Invoke except that it takes a va_list argument to enable the application to supply a pointer to a variable argument list. The relationship between Invoke and InvokeArgs is similar to the relationship between printf and vprintf.

## 6.3 Communication between Multiple Flash Files

The communication methods discussed so far have all involved C++. In a large application where the UI is spilt into multiple SWF files, a great deal of C++ code would need to be written to enable the different components of the interface to communicate with each other.

For example, a massively multiplayer online game (MMOG) might have an inventory HUD, an avatar HUD, and a trading room. Items such as swords and money could move from the player's inventory to the trading room give it to another player. When the player wears an item of clothing it would move from the inventory HUD to the avatar HUD.

These three interfaces must be broken into separate SWF files and loaded separately in order to conserve memory. However, the writing C++ code to maintain communication between the three GFx::Movie objects for these three interfaces will quickly become cumbersome.

There is a better way. ActionScript supports the loadMovie and unloadMovie methods, which enable multiple SWF files to be swapped into and out a single GFx::Movie. As long as the SWF movies are in the same GFx::Movie, they share the same variable space, eliminating the need for C++ code to initialize each movie every time it is loaded.

As an example, we will create a container.swf (container does not exist in any of the tutorial versions) file with ActionScript functions to load and unload movies into a shared variable space. The container file contains no art assets and is nothing more than several ActionScripts to manage movie loading and unloading. The first step is to create a MovieClipLoader object:

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

For more information on this and other ActionScript functions used here see the Flash documentation.

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

Using `loadMovie` to load a movie into a specific named clip enables more structured organization of a complex interface. Movies can be arranged in a tree structure and variables can be addressed accordingly (e.g., _root.inventoryWindow.widget1.counter).

Many Flash movie clips reference variables based on _root. If a movie is loaded into a specific level, _root refers to the base of that level. For example, for a movie loaded into level 6, _root.counter and _level6.counter refer to the same variable

The same movie loaded with `loadMovie` into _root.myMovie will not work normally, since _root.counter refers to the counter variable at the base of the tree. Movies organized into a tree structure should set this:_lockroot = true. Lockroot is an ActionScript property that causes all references to _root to point to the root of the submovie, not the root of the level. For more on ActionScript variables, levels, and movie clips see the Adobe Flash documentation.

Regardless of how submovies are organized, movie clips running inside the same GFx::Movie can access each other's variables and operate off of shared state, greatly simplifying creation of complex interfaces.

Container.fla also contains corresponding functions to unload unneeded movie clips to reduce memory consumption (e.g., once a user closes a window, the content can be freed).

When `loadMovie` returns, the movie has not necessarily finished loading. The loadClip function only initiates the loading of the movie, which then continues in the background.If your application must know when the movie has completed loading (e.g., to programmatically initialize state), Loader listener functions can be used. Container.fla has placeholder implementations of these functions that can be extended to either process the events in ActionScript or make an ExternalInterface call to enable the C++ application to take action:

```
function loadMovie(url:String):void {
        var loader = new Loader();
        loader.load( new URLRequest( url ) );
        loader.contentLoaderInfo.addEventListener( Event.OPEN, handleLoadOpen,
                                                   false, 0, true );
        loader.contentLoaderInfo.addEventListener( Event.INIT, handleLoadInit,
                                                   false, 0, true );
        loader.contentLoaderInfo.addEventListener( Event.PROGRESS,
                                        handleLoadProgress, false, 0, true );
        loader.contentLoaderInfo.addEventListener( Event.COMPLETE,
                                        handleLoadComplete, false, 0, true );
        addChild( loader )
    }

    function handleLoadOpen( e:Event ):void {
        trace("Event.OPEN - Load Started!");
    }
    function handleLoadInit( e:Event ):void {
        trace("Event.INIT - Loaded Content Ready!");
    }
    function handleLoadProgress( e:Event ):void {
        trace("Event.PROGRESS - Load Progress!");
    }
    function handleLoadComplete( e:Event ):void {
        trace("Event.COMPLETE - Load Complete!");
    }
```

# 7 Rendering To Texture

Rendering to a texture is a commonly used advanced technique in 3D rendering. This technique allows the user to render to a texture surface instead of the back buffer. The resulting texture can then be used a regular texture and applied to scene geometry as desired. For example, this technique can be used to create an "in game billboard". First compose your billboard as a SWF file in Flash Studio, render the SWF file to a texture and then apply the texture to the appropriate scene geometry.
In this tutorial, we simply render the UI from the last tutorial to the back wall. If you depress the middle mouse button to move the light source around, you will see that the appearance of the UI changes appropriately.


We will now walk you through the inner workings of this sample.

1. In all of the previous tutorials, we have been loading the cell.x scene file. This file contains the vertex coordinates, triangle indices and texture information needed to display the floor, walls and ceiling of the room. If you look at this file, you will notice that the same wall texture (cellwall.jpg) is used for all the four walls. We want our UI to be rendered only on the back wall. Therefore, we create a separate texture to be used for the back wall called cellwallRT.jpg. Next, modify the MeshMaterialList array to refer to this texture for the back wall.

```
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. As noted from previous tutorials, the parsing of the cell.x file is done internally by the DXUT framework. During this parsing, DXUT initializes the vertex and index buffers and creates the textures referred to in the materials list. The textures are created using the CreateTexture call; however the usage parameters passed to this function are not appropriate for Render Textures. Please refer to DXSDK documentation for more information on this subject. In order to create a texture that can be used as a render target, recreate the render texture using the right usage parameters and attach it to the background mesh. Also release the texture originally created by DXUT to avoid memory leaks.

```
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. Next modify the AdvanceAndRender function to render GFx to our texture instead of on the backbuffer. The key steps in this process are as follows:
   a. First save the original back buffer surfaces, so that we can revert back once we are done rendering.
      ```
      pd3dDevice->GetRenderTarget(0, &poldSurface);
      pd3dDevice->GetDepthStencilSurface(&poldDepthSurface);
      ```
   b. Next, obtain the render surface from our texture and set it as the render target.
      ```
      ptex->GetSurfaceLevel(0, &psurface);
      HRESULT hr = pd3dDevice->SetRenderTarget(0, psurface );
      ```
   c. Adjust the movie view port to the dimensions of our texture, call Display and then revert back to the original render surface saved in step a.

# 8 OpenGL Sample

We have also included a small OpenGL/GLUT integration sample (Tutorial\OpenGL) based on Nvidia tessellation example (http://developer.download.nvidia.com/SDK/10/opengl/samples.html#tessellation). This is a very minimal sample and it uses a slightly different approach than the D3D9 tutorial. As you can see by comparing tessellation.cpp with tessellation_original.cpp there are only a few lines of code added to the original NVIDIA sample. The main program communicates to GFx through high level call calls such as Init(), ShutDown, AdvanceAndDisplay(), etc. and all the GFx specific implementation code is encapsulated into GFxPlayerImpl class (GFxPlayerGL.cpp).

# 9 Pre-processing with GFxExport

Until now we've been loading Flash SWF files directly. This streamlines development because artists can swap in new SWF content as they develop, and see the results in game without requiring a revised game executable. However, including SWF content in a release is not recommended because loading a SWF file requires processing overhead and impacts load time.

GFxExport is a utility that processes SWF files into a format that is optimized for streamlined loading. During preprocessing, images are extracted into separate files for management by the game's resource engine. Images can be converted to DDS files with DXT texture compression for optimized loading and run-time memory savings. Embedded fonts are recompressed, or font textures can optionally be pre-computed for cases in which bitmap fonts are to be used. GFxExport output can optionally be compressed.

Deploying with GFX files is simple. The GFxExport utility supports a wide variety of options that are documented in the help screen. To convert the d3d9guide.swf and fxplayer.swf files to GFx format:

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

gfxexport.exe is located in the C:\Program Files\Scaleform\ $(GFXSDK) directory. These commands are also in the convert.bat file in Tutorial\Section7.

The -i option specifies the image format, in this case DDS. DDS makes the most sense for DirectX platforms because it enables DXT texture compression, which will typically result in 4x run-time texture memory savings.

The –c option enables compression. Only the vector and ActionScript content in the GFx file is compressed. Image compression depends on the image output format chosen and DXT compression options.

The –share_images option reduces memory usage by identifying identical images in different SWF files and loading only a single shared copy.

The version of ShadowVolume in Tutorial\Section8 has been modified to load the GFx files simply by changing the filename argument to GFx::Loader::CreateMovie.

# 10 Next Steps

This tutorial has provided a basic overview of Scaleform's capabilities. Here are some additional topics we suggest you explore:

- In-game Flash render-to-texture. See the Scaleform Player SWF to Texture SDK sample, the Gamebryo™ integration demo, and the Unreal® Engine 3 integration demo.
- Performance and other issues covered in the developer center FAQs.
- Scale9 window support documented in the Scale9Grid Overview on the documentation page.
- Custom loading: in addition to loading from GFx or SWF files, Flash content can be loaded directly from memory or the game's resource manager by subclassing GFx::FileOpener.
- Custom images and textures through GFx::ImageCreator.
- Localization with GFx::Translator.