

GfX Integration Tutorial

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

Author: Ben Mowery
Version: 2.04
Last Edited: November 11, 2009

Copyright Notice

Autodesk® Scaleform® 3

© 2011 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/or other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, Algor, Alias, Alias (swirl design/logo), AliasStudio, Alias|Wavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backburner, Backdraft, Beast, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DXF, Ecotect, Exposure, Extending the Design Team, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, Freewheel, GDX Driver, Green Building Studio, Heads-up Design, Heidi, HumanIK, IDEA Server, i-drop, Illuminate Labs AB (design/logo), ImageModeler, iMOUT, Incinerator, Inferno, Inventor, Inventor LT, Kynapse, Kynogon, LandXplorer, LiquidLight, LiquidLight (design/logo), Lustre, MatchMover, Maya, Mechanical Desktop, Moldflow, Moldflow Plastics Advisers, MPI, Moldflow Plastics Insight, Moldflow Plastics Xpert, Moondust, MotionBuilder, Movimento, MPA, MPA (design/logo), MPX, MPX (design/logo), Mudbox, Multi-Master Editing, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProMaterials, RasterDWG, RealDWG, Real-time Roto, Recognize, Render Queue, Retimer, Reveal, Revit, RiverCAD, Robot, Scaleform, Scaleform AMP, Scaleform CLIK, Scaleform GFx, Scaleform IME, Scaleform Video, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Softimage|XSI (design/logo), Sparks, SteeringWheels, Stitcher, Stone, StormNET, StudioTools, ToolClip, Topobase, Toxik, TrustedDWG, U-Vis, ViewCube, Visual, Visual LISP, Volo, 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	GFx 3.3 Integration Tutorial
Address	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1. Introduction	1
2. Documentation Overview	2
3. Installation and Build Dependencies	3
3.1 Installation	3
3.2 Compile the Demos	4
3.3 Benchmark SWF Playback in GfxPlayer	5
3.4 Compile the Sample Base	6
3.5 Gfx Build Dependencies	7
4. Game Engine Integration	10
4.1 Rendering Flash	10
4.2 Scaling Modes	19
4.3 Processing Input Events	21
4.3.1 Mouse Events	21
4.3.2 Keyboard Events	23
4.3.3 Hit Testing	24
4.3.4 Keyboard Focus	25
5. Introduction to Localization and Fonts	26
5.1 Font Overview and Capabilities	26
5.2 Font Example: Embedding Chinese Characters	28
6. IME Overview	32
7. Interfacing C++, Flash, and ActionScript	34
7.1 ActionScript to C++	34
7.1.1 FSCommand Callbacks	34
7.1.2 ExternalInterface	37
7.2 C++ to ActionScript	40
7.2.1 Manipulating ActionScript Variables	40
7.2.2 Executing ActionScript Subroutines	42
7.3 Communication Between Multiple Flash Files	44
8. Pre-processing with GfxExport	50
9. Next Steps	51

1. Introduction

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

In addition to UI, developers can use GfX to display Flash content inside the 3D environment as animating textures mapped onto 3D surfaces—think *Doom 3* screens or Flash UI on 3D objects. Likewise, 3D objects and video can be displayed inside Flash UI. As a result, Scaleform GfX 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 through the process of installing and using Scaleform GfX. 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 GfX can be used directly with supported game engines with minimal coding. This guide is primarily targeted at engineers planning to integrate Scaleform GfX with a custom game engine, or to those looking for an in-depth technical overview of Scaleform GfX’s capabilities.

Note: Please make sure to use the latest version of GfX when following this tutorial. The tutorial works with GfX version 2.2 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 not a compatibility issue with GfX. 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 GfXPlayer application runs successfully.

2. Documentation Overview

The latest multilingual GfX documentation can be found online in the Scaleform Developer center at: <http://developer.scaleform.com/>. Free registration is required to access the site.

Current documentation includes:

- Web-based GfX SDK reference documentation: <http://developer.scaleform.com/gfx?action=doc>.
- PDF documentation: <http://developer.scaleform.com/gfx?action=doc>.
- Font Overview: Describes the font and text rendering system and provides details on how to configure both the art assets and GfX C++ APIs for internationalization.
- XML Overview: Describes the XML support available in GFX.
- Scale9 Grid: Explains how to use Scale9Grid functionality to create resizable windows, panels, and buttons.
- IME Configuration: Describes how to integrate GfX's IME support into end-user applications and how to create custom IME text input window styles in Flash.
- ActionScript Extensions: Covers GfX ActionScript extensions.

3. Installation and Build Dependencies

3.1 Installation

Download the most recent GfX and IME installers for Windows/DirectX. Run *both* installers and keep the default installation paths and options. GfX will be installed to the C:\Program Files\Scaleform\GfX SDK 3.3 directory.

The layout is as follows:

3rdParty

External libraries required by GfX such as libjpeg and zlib.

Apps\Demos

Pre-compiled binaries for these demos can be found in the Bin directory. The Visual Studio project files are on the start menu under Scaleform → GfX SDK → Demo MSVC Solutions.

3DDemo: Demonstrates rendering Flash content to texture. The GfX-based UI enables selection of different Flash movie clips and 3D models. Flash content renders to the rotating 3D model.

RenderTexture: Source for the “GfXPlayer SWF to Texture” and “GfXPlayer Texture in SWF” programs in the Scaleform → GfX SDK → Demo Examples folder.

FxPlayer: Source for the “GfXPlayer D3Dx” programs. These are SWF/GfX players that enable viewing and performance benchmarking of hardware playback of Flash content.

Projects

Visual Studio projects to build the above applications.

Apps\Samples

Sample source code for features and also this tutorial.

Bin

Contains pre-built demo binaries and sample Flash content:

3DDemo: Source Flash content for 3DDemo application.

RenderTexture: Sample user interface FLA source.

FxPlayer: Simple Flash text HUD.

IME:	Sample flash files for a custom IME text input interface.
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 GfxPlayer and demo applications.
gfxexport.exe:	GfxExport is a pre-processing tool that accelerates loading of Flash content and is covered in detail in section 8.

Note: Re-building the demo .sln Visual Studio projects will replace the binaries in the Win32 directory.

Include

Gfx headers.

Lib

Gfx libraries.

Resources

CLIK components and tools.

Doc

PDF documentation described in section 2.

3.2 Compile the Demos

In order to verify your system is configured properly to build Gfx, first build the demo solutions found under Start → All Programs → Scaleform → Gfx SDK 3.3 → Demo MSVC 8.0 Solutions → Gfx 3.3 Demos.sln. Once the .sln file is open in Visual Studio, choose the D3D9_Debug_Static configuration and build the GfxPlayer project.

Building GfxPlayer is also necessary to build third-party libraries such as libjpeg and zlib. The source is included with the distribution, but it is necessary to build the demo .sln file in order to generate binary libraries for your application to link with.

Check the modification date on the C:\Program Files\Scaleform\Gfx SDK 3.3\Bin\Win32\Msvc80\GfxPlayer\FxPlayer_D3D9_Debug_Static.exe file to confirm it was successfully rebuilt and run the program.

This program and the other GfX Player D3Dx applications on the Start → All Programs → Scaleform → GfX SDK 3.3 → Demo Examples folder are hardware-accelerated SWF players. During development, GfX Flash playback can be tested and benchmarked with these tools.

Also build the FxPlayer_D3D9_Release_Static configuration to generate libraries for the release build of the example project we will begin developing in section 3.4.

3.3 Benchmark SWF Playback in GfXPlayer

Run the Start → All Programs → Scaleform → GfX SDK 3.3 → GfX Players → GfX Player D3D9 program. Open the C:\Program Files\Scaleform\GfX SDK 3.3\Bin\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 GfX 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.

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

Zoom in on the “3D GFx 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 GFxPlayer 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 GFx 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 GFx builds do not provide optimal performance.

3.4 Compile the Sample Base

For GFx 3.0 and above, open the Tutorial solution under Scaleform\GFx SDK 3.3\Apps\Samples\Tutorial. For GFx 2.2, download and install the Tutorial from the [Scaleform GFx Demos Page](#) and open the Tutorial.sln. You can find the solutions for Visual Studio 2005 and Visual

Studio 2008 in the windows start menu in Scaleform→ GFx SDK 3.3→ 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 GFx Build Dependencies

When creating a new GFx 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 3.3\; if your libs are in a different location, you will need to replace the paths containing \$(GFXSDK) to point to the location of the libs on your system. We use "Msvc80" for these examples; if you use Visual Studio 2003 or Visual Studio 2008, you will need to use Msvc71 or Msvc90, respectively.

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

```
$(GFXSDK)\Src\GRenderer  
$(GFXSDK)\Src\GKernel  
$(GFXSDK)\Src\GFxXML  
$(GFXSDK)\Include
```

Paste the following string into the Visual Studio "Additional Include Directories" field:

```
"$(GFXSDK)\Src\GRenderer";" $(GFXSDK)\Src\GKernel";"$(GFXSDK)\Src\GFxXML";"  
$(GFXSDK)\Include"
```

The following library directories should be added to the linker search paths for the debug build configuration:

```
$(DXSDK_DIR)\Lib\x86  
$(GFXSDK)\3rdParty\expat-2.0.1\lib  
$(GFXSDK)\Lib\$(PlatformName)\Msvc80\Debug_Static\  
$(GFXSDK)\3rdParty\zlib-1.2.3\Lib\$(PlatformName)\Msvc80\Debug  
$(GFXSDK)\3rdParty\jpeg-6b\Lib\$(PlatformName)\Msvc80\Debug
```

Paste the following string into the “Additional Library Directories” field:

```
"$(DXSDK_DIR)\Lib\x86";  
"$(GFXSDK)\3rdParty\expat-2.0.1\lib";  
"$(GFXSDK)\Lib\$(PlatformName)\Msvc80\debug";  
"$(GFXSDK)\3rdParty\zlib-1.2.3\Lib\$(PlatformName)\Msvc80\Debug";  
"$(GFXSDK)\3rdParty\jpeg-6b\Lib\$(PlatformName)\Msvc80\Debug"
```

Note: Change Msvc80 to the string corresponding to your version of Visual Studio.

The corresponding release libraries should be added to the linker search paths for the release and profile build configurations:

```
$(DXSDK_DIR)\Lib\x86  
$(GFXSDK)\3rdParty\expat-2.0.1\lib  
$(GFXSDK)\Lib\$(PlatformName)\Msvc80\Release\  
$(GFXSDK)\3rdParty\zlib-1.2.3\Lib\$(PlatformName)\Msvc80\Release  
$(GFXSDK)\3rdParty\jpeg-6b\Lib\$(PlatformName)\Msvc80\Release
```

Paste the following string into the “Additional Library Directories” field (release and profile configurations):

```
"$(DXSDK_DIR)\Lib\x86";  
"$(GFXSDK)\3rdParty\expat-2.0.1\lib";  
"$(GFXSDK)\Lib\$(PlatformName)\Msvc80\Release";  
"$(GFXSDK)\3rdParty\zlib-1.2.3\Lib\$(PlatformName)\Msvc80\Release";  
"$(GFXSDK)\3rdParty\jpeg-6b\Lib\$(PlatformName)\Msvc80\Release"
```

Note: Change Msvc80 to the string corresponding to your version of Visual Studio.

Finally, add the GfX libraries and their dependencies:

- libgfx.lib
- libjpeg.lib
- zlib.lib
- imm32.lib
- winmm.lib
- libgrenderer_d3d9.lib

Add the same libraries to the release and profile build configurations.

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

4. Game Engine Integration

Scaleform provides integration layers for 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 GfX in games developed with these engines.

This section describes how to integrate GfX 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 GfX into the application to replace the default DXUT user interface with a Flash-based GfX 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 GfX 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 [GfXLoader](#) object to manage loading of all Flash content globally for the application, a [GfXMovieDef](#) to contain the Flash content, and a [GfXMovieView](#) to represent a single playing instance of the movie. Additionally a [GRenderer](#) object and a [GfXRenderConfig](#) object will be instantiated to act as the interface between GfX 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 "GTimer.h"  
#include "GfxEvent.h"  
#include "GfxPlayer.h"
```

```
#include "GFxFontLib.h"

#include "FxPlayerLog.h"
#include "GRendererD3D9.h"
```

Copy the FxPlayerLog.h file from C:\Program Files\Scaleform\GFx SDK 3.3\Apps\Demos\FxPlayer.

Several GFx 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 GFx state together in a class has the advantage that a single delete call will free all GFx objects. The interaction of these objects will be described in detail in the steps below.

```
// One GFxLoader per application
GFxLoader          gfxLoader;

// One GFxMovieDef per SWF/GFx file
GPtr<GFxMovieDef>   pUIMovieDef;

// One GFxMovieView per playing instance of movie
GPtr<GFxMovieView>  pUIMovie;

// D3D9 Renderer
GPtr<GRendererD3D9> pRenderer;
GPtr<GFxRenderConfig> pRenderConfig;
```

Step #2: Initialize GFxSystem

The first stage of GFx initialization is to instantiate a [GFxSystem](#) object to manage GFx memory allocation. In WinMain we add the lines:

```
// One GFxSystem per application
GFxSystem          gfxInit;
```

The GFxSystem object must come into scope before the first GFx call and cannot leave scope until the application is finished using GFx which is why it is placed in WinMain. GFxSystem as instantiated here uses GFx'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 GFxSystem and take no further action.

GFxSystem 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 `GFxSystem::Init()` and `GFxSystem::Destroy()` static functions instead of creating the `GFxSystem` object instance.

Step #3: Loader and Renderer Creation

The remainder of GFx 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 [GFxLoader](#) object. An application typically has only one `GFxLoader` 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. `GFxLoader` also maintains a set of configuration states such as [GFxLog](#), used for debug logging.

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

```
// Initialize logging -- GFx will print errors to the log
// stream.
gfxLoader->SetLog(GPTr<GFxLog>(*new GFxPlayerLog()));
```

`GFxLoader` reads content through the [GFxFileOpener](#) class. The default implementation reads from a file on disk, but custom loading from memory or a resource file can be accomplished by subclassing `GFxFileOpener`.

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

`GRenderer` is a generic interface that enables GFx to output graphics to a variety of hardware. We create an instance of the `D3D9` renderer and associate it with the loader. The renderer object is responsible for managing the `D3D` device, textures, and vertex buffers used by GFx. Later on, in `SetDependentVideoMode`, we will pass `GRenderer` an

IDirect3DDevice9 pointer initialized by the game, so that GfX can create DX9 resources and successfully render UI content.

```
// Create a GfX renderer and connect it with our D3D state
pRenderer = *GRendererD3D9::CreateRenderer();

// Associate the renderer with the GfXLoader
pRenderConfig = *new GfXRenderConfig(pRenderer);
gfxLoader->SetRenderConfig(pRenderConfig);
```

GfXLoader's [SetRenderConfig](#) method associates the renderer with the GfXLoader. Every GfXMovieDef created by the loader will inherit the renderer.

The above code uses the default GRendererD3D9 object supplied with GfX. Subclassing GRenderer enables better control over GfX's rendering behavior and can result in a tighter integration.

In addition to containing the pointer to the GRendererD3D9 object, GfXRenderConfig also manages various rendering parameters such as curve tolerance and EdgeAA:

```
// Use EdgeAA to improve the appearance of the interface without the
// computational expense of full AA through the video card.
pRenderConfig->SetRenderFlags(GfXRenderConfig::RF_EdgeAA);
```

EdgeAA adds subpixel triangles around the edges of shapes to create a smoother appearance, but without the computational expense of enabling full anti-aliasing on the video card. This is one advantage of vector graphics: the shape information enables anti-aliasing to be selectively applied to only the areas of the screen that stand to benefit most, such as button edges and large text characters. Although the triangle count increases, the overall performance impact is manageable because the draw primitive (DP) count does not increase.

Step #4: Load a Flash Movie

Now the GfXLoader is ready to load a movie. Loaded movies are represented as [GfXMovieDef](#) objects. The GfXMovieDef 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,
                                     GfXLoader::LoadKeepBindData |
                                     GfXLoader::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 GfxThreadTaskManager 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 [GfxMovieView](#) instance must be created from the GfxMovieDef object. GfxMovieView 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);
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 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.

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);

// 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

GFx must be given the handle to the DirectX device and presentation parameters through GRenderer in order to render. GRenderer::SetDependentVideoMode should be called after the D3D device is created and before GFx is asked to render. SetDependentVideoMode 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:

```
HWND hWND = DXUTGetHWND();  
pRenderer->SetDependentVideoMode(pd3dDevice, &presentParams,  
    GRendererD3D9::VMConfig_NoSceneCalls, hWND);
```

The SetDependentVideoMode() call passes the D3D device and presentation parameters to GFx. The GRendererD3D9::VMConfig_NoSceneCalls flag specifies that no DirectX BeginScene() and EndScene() calls will be made by GFx. 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. GRenderer can be informed of the lost device and given a chance to free its D3D resources in the corresponding OnLostDevice method in GFxTutorial:

```
pRenderer->ResetVideoMode();
```

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 GFx SDK.

Step #8: Resource Allocation and Cleanup

Because all Gfx 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;
GMemory::DetectMemoryLeaks();
```

GMemory::DetectMemoryLeaks() will print any Gfx-related memory leaks to the debugger through the Win32 OutputDebugString() function. Memory leaks will be reported unless GfxLoader and all other Gfx states have been properly deallocated.

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

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

ResetVideoMode 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 ResetVideoMode call in GfxTutorial::OnLostDevice matches the SetDependentVideoMode call in GfxTutorial::OnResetDevice.

When integrating with your own engine, try to call SetDependentVideoMode and ResetVideoMode 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,
```

```
        windowHeight, 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 Gfx 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.

Gfx 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);
    pUIMovie->Display();
}
```

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 GfxMovie::Advance to ensure the movie plays back correctly on different hardware configurations.

Step #11: Preserving Rendering States

[GfxMovieView::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 GfxMovieView::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 Gfx 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 GFx
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 GFx Flash-based UI

4.2 Scaling Modes

The calls to `GFxMovieView::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. GFx 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 GFx 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). GFx 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.

[GFxMovieView::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 `GFxMovieView` object:

```
pUIMovie->SetViewScaleMode(GFxMovieView::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(GFxMovieView::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. Sample ActionScript code can be found in `d3d9guide.fla`.

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 GfX, 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 [GfXMovieView::HandleEvent](#) passes a GfXEvent 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 GfXMovieView.

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 GfX 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)
        {
            GfXMouseEvent mevent(GfXEvent::MouseMove, 0, mx, my);
            pUIMovie->HandleEvent(mevent);
        }
        else if (pMovieButton && uMsg == WM_LBUTTONDOWN)
        {
            ::SetCapture(hWnd);
            GfXMouseEvent mevent(GfXEvent::MouseDown, 0, mx, my);
            pUIMovie->HandleEvent(mevent);
        }
        else if (pMovieButton && uMsg == WM_LBUTTONUP)
        {
            ::ReleaseCapture();
            GfXMouseEvent mevent(GfXEvent::MouseUp, 0, mx, my);
            pUIMovie->HandleEvent(mevent);
        }
    }
}
```

GFx 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 GFx 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 GFx.

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 ($\text{screen_width} / 2 - \text{movie_width} / 2$, $\text{screen_height} / 2 - \text{movie_height} / 2$) 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 [GFxMovieView::SetViewAlignment](#) these transformations do not have to be performed. As long as the mouse coordinates are relative to the coordinates given to [GFxMovieView::SetViewport](#), alignment and scaling performed by [SetViewAlignment](#) and [SetViewScaleMode](#) will be handled internally by GFx.

4.3.2 Keyboard Events

Keyboard events are also handled through [GFxMovieView::HandleEvent](#). There are two kinds of key events: [GFxKeyEvent](#) and [GFxCharEvent](#):

```
GFxKeyEvent(EventType eventType = None,
             GFxKey::Code code = GFxKey::VoidSymbol,
             UByte asciiCode = 0,
             UInt32 wcharCode = 0,
             UInt8 keyboardIndex = 0)

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

A `GFxKeyEvent` is similar to a raw scan code; a `GFxCharEvent` is similar to a processed ASCII character. In Windows, a `GFxCharEvent` would be generated in response to the `WM_CHAR` message; `GFxKeyEvents` 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 `GFxKeyEvent` 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 `GFxCharEvent` should be fired in response to the `WM_CHAR` message to pass the "cooked" ASCII value 'C' to GfX.
- Once the 'c' key is released, a `GFxKeyEvent` should be sent in response to the `WM_KEYUP` message. No `GFxCharEvent` need be sent when a key is released.
- The F5 key is pressed: A `GFxKeyEvent` is sent when the key goes down, and a second event when the key goes back up. It is not necessary to send a `GFxCharEvent` because F5 does not correspond to a printable ASCII code.

Separate `GFxKeyEvent` events are sent for key down and key up events. To enable platform independence, the key code is defined in `GFxEvent.h` to match the key codes used internally by Flash. The `GFxPlayerTiny.cpp` example and `GFxPlayer` 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(GFxMovieView *pMovie, UINT 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 GfX events will be generated and sent to pMovie.

Sending both GfXKeyEvents and GfXCharEvents is important. For example, most text boxes respond only to GfXCharEvents 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 GfXKeyEvent 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 d3d9guide fla implements this simple logic using ActionScript.

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 GfXMovieView::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)
{
```

```

        GFxMouseEvent mevent(GFxEvent::MouseMove, 0, (Float)mx, (Float)my);
        pUIMovie->HandleEvent(mevent);
        processedMouseEvent = true;
    }
    else if (uMsg == WM_LBUTTONDOWN)
    {
        ::SetCapture(hWnd);
        GFxMouseEvent mevent(GFxEvent::MouseDown, 0, (Float)mx, (Float)my);
        pUIMovie->HandleEvent(mevent);
        processedMouseEvent = true;
    }
    else if (uMsg == WM_LBUTTONUP)
    {
        ::ReleaseCapture();
        GFxMouseEvent mevent(GFxEvent::MouseUp, 0, (Float)mx, (Float)my);
        pUIMovie->HandleEvent(mevent);
        processedMouseEvent = true;
    }

    if (processedMouseEvent && pUIMovie->HitTest((Float)mx, (Float)my,
                                                GFxMovieView::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 GFx, but also passed to the 3D camera.

Resolving this issue requires determining whether the text input box has focus. Section 7.1.2 will describe how Flash ActionScript can be used to send events to C++ enabling our event handler to track focus.

5. Introduction to Localization and Fonts

5.1 Font Overview and Capabilities

Scaleform GFx 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 GFx's font and internationalization capabilities can be found on the Developer center documentation page: <http://developer.scaleform.com/gfx?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.

GFx 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 GFx's intelligent glyph-packing algorithm. GFx 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, GFx 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\Samples\FontConfig` and drag the `sample.swf` file onto an open GFxPlayer D3D9 window.

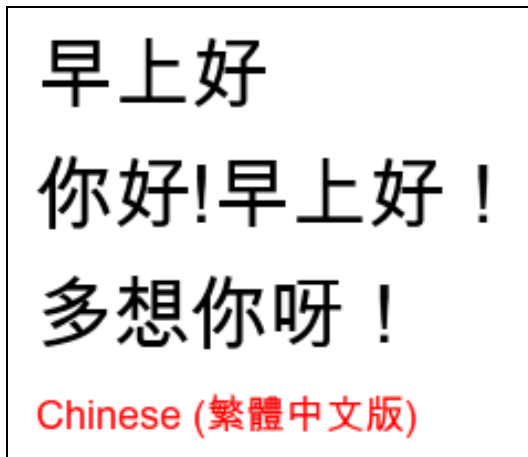


Figure 4a: Small Chinese characters

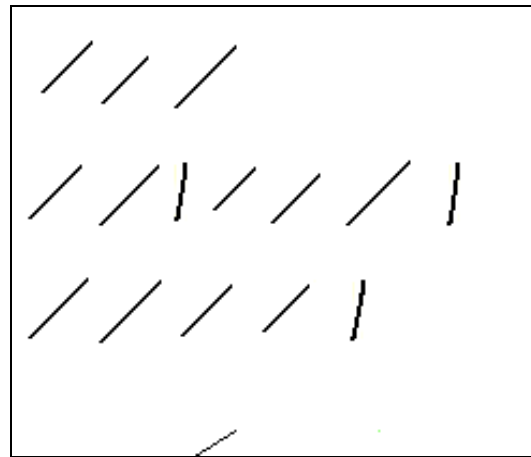


Figure 4b: Wireframe representation

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

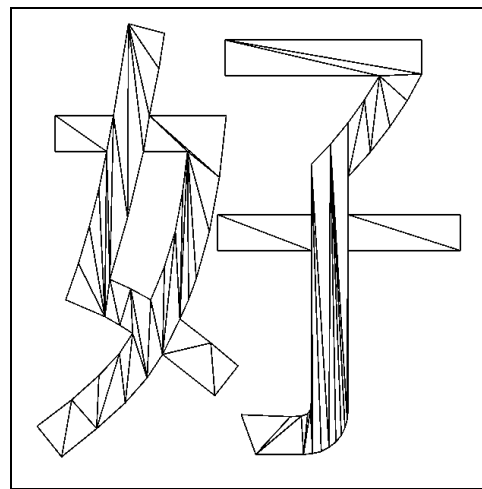


Figure 4d: Wireframe representation

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, GFx 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

5.2 Font Example: Embedding Chinese Characters

The following example demonstrates how to support Chinese text input in the text input box added to ShadowVolume. If your application does not require Asian font support, skip ahead to section 6.

Run the ShadowVolume application and click the “Change Mesh” button to open the text input box. Switch to a Chinese IME and type 你好 into the box:



Figure 6a: Chinese text input

Switch to the Microsoft Chinese Pinyin IME type “nihao” and then press space twice to output the characters. The characters appear as empty boxes because the text box font is missing the Chinese

character glyphs:

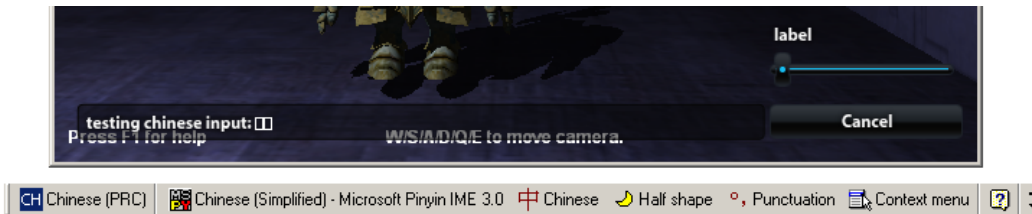


Figure 6b: Empty squares representing missing glyphs

Also note that GFx printed a warning to the console that missing glyphs were encountered:

```
Missing "Myriad Pro" glyph '`' (20320) in "_level0.hud.text_MeshPath.field".
Font has 114 glyphs, ranges 0x20-0x7e, 0x2c6, 0x2dc, 0x2013-0x2014,
0x2018-0x201a (truncated).
Search log:
  Searching for font: "Myriad Pro" [Bold]
  Movie resource: "Myriad Pro" [Bold] found.
```

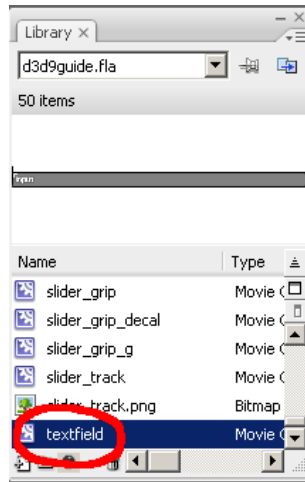
There are several ways to provide the missing glyphs to GFx:


1. Embed the missing Chinese font glyphs in the d3d9guide.swf file.
2. Embed Chinese fonts in a shared font file, gfxfontlib.swf, enabling the font to be shared by all loaded Flash files.
3. Obtain fonts from the operating system through GfxFontProviderWin32.
4. Obtain fonts directly from a TrueType TTF font through GfxFontProviderFT2.

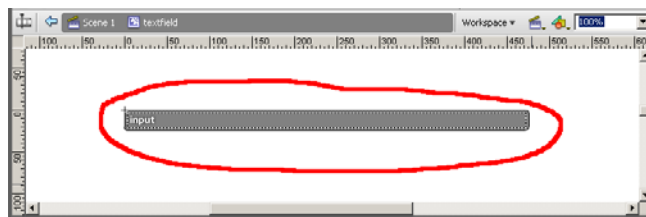
To keep this example simple, we choose the first option: Embed the missing font. The second option is recommended for portability, as keeping the font data with the game package enables playback on a system that does not have the required font or does not have operating system font support, such as most game consoles. The [Font and Text Configuration Overview](#) document describes all of these options in detail.

Step #1: Open d3d9guide.fla in Flash Studio.

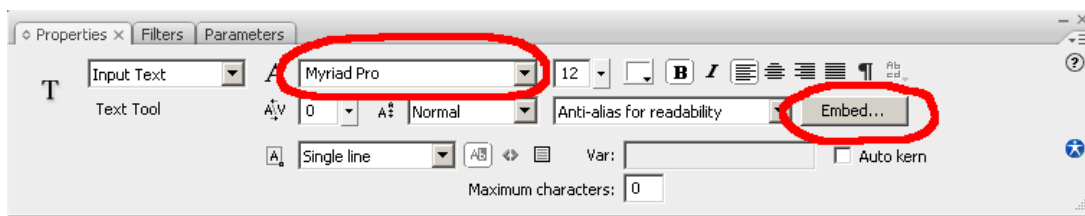
Step #2: Double click the textfield item at the bottom of the library pane. (The library pane is typically located in the lower right of the screen.)



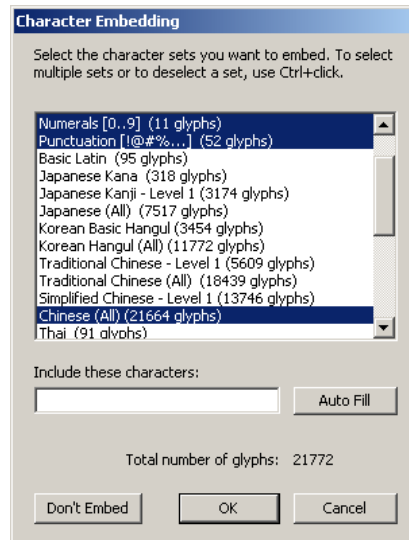
Step #3: Double click with the black arrow tool  on the text input box that is now on the stage.



Step #4: Choose a Chinese SimSun font from the font dropdown box in the properties dialog that is typically positioned at the bottom of the screen. Then click the “Embed” button to cause Flash Studio to export the character glyphs to the movie.



Step #5: Use CTRL+Click to select “Chinese (All) (21664 glyphs)” from the Character Embedding dialog, taking care to maintain the original selection which includes the Latin characters, numerals, and punctuation.



Step #6: Save the file and export a new SWF movie by pressing Ctrl+Alt+Shift+S or choosing File → Export → Export Movie...

Note: Make sure to save the Flash file in the Flash 8 format by going through the File → Save As... dialog. Also, when exporting to a SWF choose to export a Flash 8 SWF with ActionScript 2.0.

To confirm that embedding was successful, check the size of the new d3d9guide.swf file. It should come to about 9MB with the embedded SimSun font.

Restart ShadowVolume and type 你好 again. With the embedded fonts Chinese input will now appear correctly:



Figure 7: Successful Chinese text input

Increase the window size to see how GFx keeps the characters looking crisp and clear at any screen resolution. This is very difficult to achieve with most other game font systems.

Note: This example follows the naïve approach of embedding font directly in the SWF file in the interests of simplicity and ease of demonstration. This is *not* the most efficient way to manage fonts. GFx provides a flexible font mechanism enabling shared fonts and localization, which is described in detail in the [Font and Text Configuration Overview](#) document. In order to make full use of GFx's font capabilities in a release-quality application please follow the more advanced techniques outlined in the [Font and Text Configuration Overview](#) document.

6. IME Overview

The example of the previous system used the Windows default Input Method Editor (IME) to enter text. The Windows default IME character selection window did not follow the cursor as we typed. It was rendered with a fixed gray style:



Figure 8: Default IME window

GFx provides full in-game IME capabilities, enabling gamers to use their favorite text input methods such as Google Pinyin and Sogo Pinyin to easily chat with their friends. The same familiar IME character selection logic is used, but GFx renders the IME UI according to a custom style defined by the artist in Flash studio. For example, an action game might choose a metallic style like the one shown below, and a MMORPG game might use a wood texture as the IME interface style. The IME interface is rendered directly into the scene as triangles:

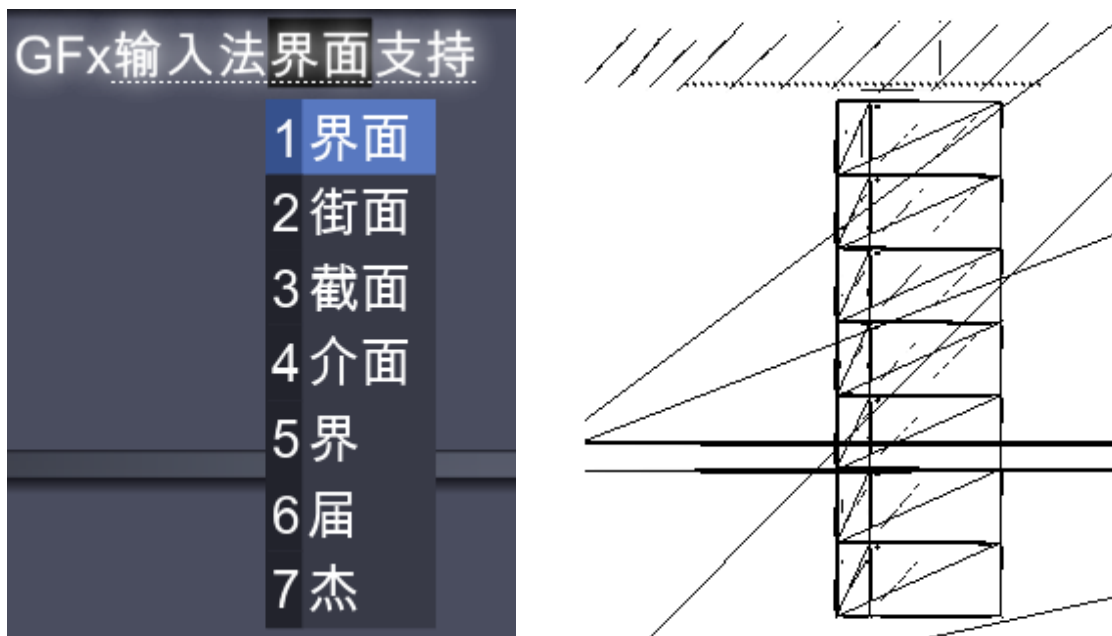


Figure 9: GFx IME window and wireframe representation demonstrating the window is rendered directly into the game environment as triangles. Notice the glow effects applied to the text being edited.

Rendering the IME window in-game as triangles enables IME in full screen games. The standard IME window would either not work or flicker.

An IME sample is available in the Demo Examples at Start Menu->Programs->Scaleform->GFX SDK 3.3->Demo Examples->IME Demo. This sample can be used to quickly evaluate GFX's IME capabilities and to test compatibility with third-party IMEs. All Microsoft system IMEs are supported, as are third-party IMEs that fully implement the Microsoft Text Services Framework (TSF) API. As many non-Microsoft IMEs do not properly implement TSF, some third-party IMEs may have issues. Should you encounter a third-party IME that is not compatible with GFX, please email a link to the IME to our support staff at support@scaleform.com.

Full IME documentation including the game integration process can be found in the [Input Method Configuration Overview](#) document.

7. 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. Gfx provides a C++ interface to directly manipulate ActionScript variables and arrays, as well as directly invoke ActionScript subroutines. Gfx also provides a callback mechanism that enables ActionScript to pass events and data back to the C++ program.

7.1 ActionScript to C++

Gfx 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 `GfxLoader` 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 [GfxValue](#) arguments (covered in section 7.1.2), 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`.

7.1.1 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 [GfxFSCommandHandler](#) and registering the class with either the `GfxLoader`

or with individual GfxMovieView objects. If a command handler is set on GfxMovieView, 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\Section7.1.

First, subclass GfxFSCommandHandler:

```
class OurFSCommandHandler : public GfxFSCommandHandler
{
    public:
    virtual void Callback(GfxMovieView* pmovie,
                        const char* pcommand, const char* parg)
    {
        GfxPrintf("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 GfxLoader object in GfxTutorial::InitGfx():

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

Registering the handler with the GfxLoader causes every GfxMovieView 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 d3d9guide.swf in Flash Studio and open the ActionScript Panel (F9). Compare the events printed to the screen with the fscommand() calls made from the ActionScript block for Symbol Definition(s) → hud → var : Frame 1:

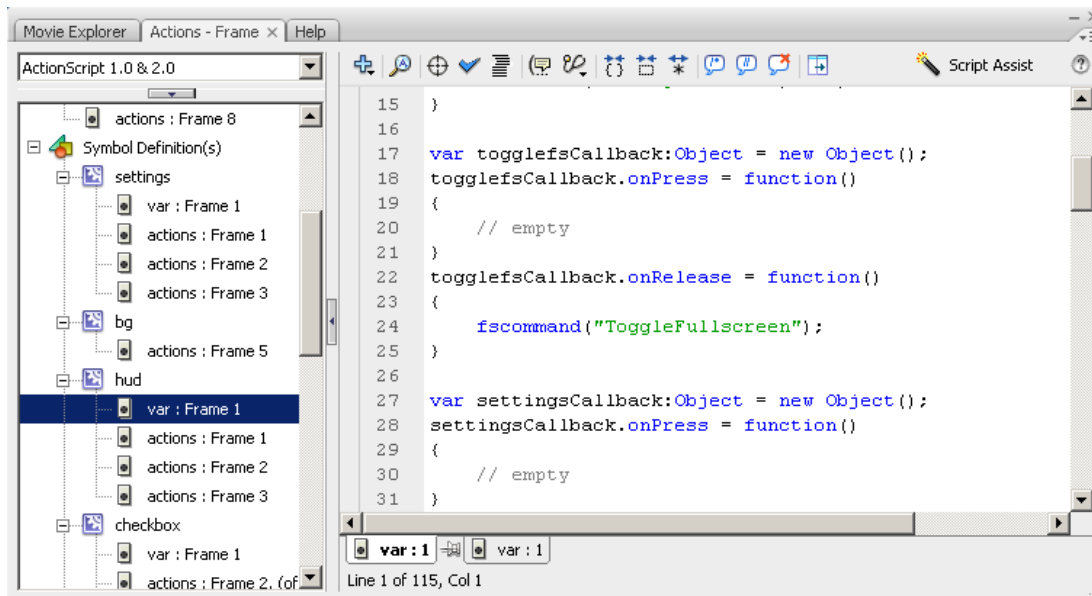


Figure 10: An ActionScript fscommand() call

As an exercise, change fscommand("ToggleFullscreen") to fscommand("ToggleFullscreen", this.UI._visible) to print the value of the this.UI._visible value to C++. Export the flash movie (CTRL+ALT+Shift+S) and replace d3d9guide.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.

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

And register the handler with GfxLoader:

```
GPtr<GfxExternalInterface> pEIHandler = *new OurExternalInterfaceHandler;
```

```
gfxLoader.SetExternalInterface(pEIHandler);
```

An external interface call will be made from ActionScript when the text input box gains or loses focus. Press F9 and look at the ActionScript for Symbol Definitions() → hud → var : Frame 1:

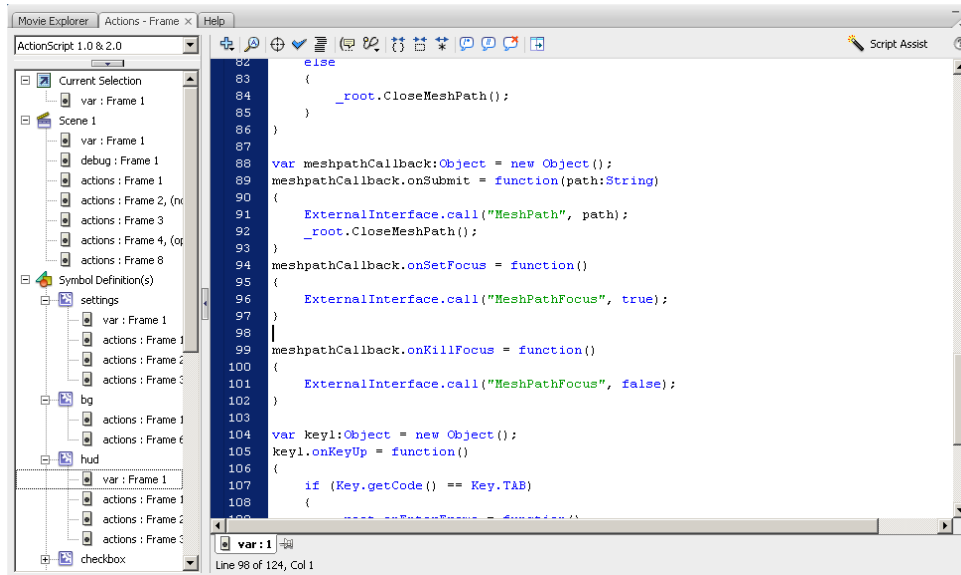


Figure 11: The MeshPathFocus callback is called whenever the text input box gains or loses focus

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:

```
ExternalInterface: MeshPathFocus, 1 args: false
Focus change:
    _level0.hud.text_MeshPath.field => null
ExternalInterface: MeshPathFocus, 1 args: true
Focus change:
    null => _level0.hud.text_MeshPath.field
```

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)
{
    if (args[0].GetType() == GfxValue::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.

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
GFxValue 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(GFxFMovie::SA_Value,
                           "_root.SceneData", 0, sceneData, 3);

```

The code in Tutorial\Section7.1 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++. Starting from Section 7.2, the d3d9guide.swf/fla files use ActionScript to initialize the dropdown list.

7.2 C++ to ActionScript

Section 7.1 explained how ActionScript can call into C++. This section describes how to communicate in the other direction using GfX functions that enable the C++ program to initiate communication with the playing movie. GfX supports C++ functions to get and set ActionScript variables as well as invoke ActionScript subroutines.

7.2.1 Manipulating ActionScript Variables

GfX supports [GetVariable](#) and [SetVariable](#), which enable direct manipulation of ActionScript variables. The code in Tutorial\Section7.2 has been modified to load the yellow HUD display (fxplayer.swf) used by the GFxPlayer program and increment a counter whenever F5 is pressed:

```

void GFxTutorial::ProcessEvent(HWND hWnd, UINT 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",
                                   GFxValue((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 [GFXMovie](#) 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 generated 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 [GFXMovie::Invoke](#) to run an ActionScript routine to change the text (more on this in section 7.2.2). *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 `GFX` must check the value of that variable on every frame.



Figure 12: `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 GfxValue 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 GfxMovie::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.

7.2.2 Executing ActionScript Subroutines

In addition to modifying ActionScript variables, ActionScript code can be triggered through the [GfxMovie::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 GfxMovie::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.

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

```
...

else if (wParam == VK_F6)
{
    const char *retval = pHUDMovie->Invoke("_root.OpenMeshPath", "");
    GfxPrintf("_root.OpenMeshPath returns '%s'\n", retval);
}
else if (wParam == VK_F7)
{
    const char *retval = pHUDMovie->Invoke("_root.CloseMeshPath", "");
    GfxPrintf("_root.CloseMeshPath returns '%s'\n", retval);
}

...
```

Notice that the ActionScript code for `openMeshPath` is placed in frame 1 to ensure that the ActionScript routine is loaded as soon as the first frame of the movie is played. 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 Gfx 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 [GfxMovieView::Advance](#) is made, or if [GfxMovieDef::CreateInstance](#) is called with `initFirstFrame` set to true.

This example used the `printf` style of `Invoke`. Other versions of the function use `GfxValue` 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`.

7.3 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 GfxMovieView 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 GfxMovieView. As long as the SWF movies are in the same GfxMovieView, they share the same variable space, eliminating the need for C++ code to initialize each movie every time it is loaded.

In the MMOG example, the inventory could be represented as an ActionScript array named `_global.inventory`. When one of the SWF interfaces is loaded it will draw its items based on the contents of this array. When one of the SWF interfaces is unloaded with ActionScript's unloadMovie method, the `_global.inventory` array remains available to the other interfaces.

As an example, we will create a container.swf 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. Movies can be either loaded into a named object or into a specific numbered level:

```
//  
// Load a file into a specific level  
  
function LoadFlashLevel(url, level)  
{  
    trace("LoadFlashLevel(" + url + ", " + level + ")\n");  
    mclLoader.loadClip(url, level);  
}
```



```

    }

    //
    // Load a file into a named object

    function LoadFlash(url, objectName)
    {
        trace("LoadFlashLevel(" + url + ", " + objectName + ")\n");
        var container:MovieClip = createEmptyMovieClip(objectName,
                                                         getNextHighestDepth());
        mclLoader.loadClip(url, container);
    }

```

Each numbered “level” can contain a single movie. Movies in different levels can share data and access each other’s variables. The level is related to the Z-order of the movie clips, enabling the UI designer to choose which clips appear in front and which clips appear behind. Movies in different levels can share data and access each other’s variables.

Loading movies with LoadMovieLevel into specific levels has the advantage that the Z-order is implicitly defined based on the level number. Variables in that movie can be accessed as `_levelN.variableName` (e.g., `_level6.counter`).

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. A movie clip that references its own internal variables with `_root` will work normally if it is loaded into the base of a specific level with LoadMovieLevel.

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 GfxMovieView 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 or LoadMovieLevel 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), MovieClipLoader's 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:

```
// Define callback functions to report events such as:
// 1. Started loading movie
// 2. Loading a movie failed
// 3. Loading a movie finished
// 4. Progress updates
//
// These callbacks are necessary because a movie does not
// necessarily load as soon as LoadMovie() returns.
//
// Currently the callbacks only print debug information.
// An application that needs to act on these events should
// either make ExternalInterface calls to notify the C++
// application or handle the events in ActionScript.

var mclListener:Object = new Object();

mclListener.onLoadError = function(target_mc:MovieClip, errorCode:String,
                                   status:Number)
{
    trace("Error loading image: " + errorCode + " [" + status + "]");
};

mclListener.onLoadStart = function(target_mc:MovieClip) : Void
{
    trace("onLoadStart: " + target_mc);
};

mclListener.onLoadProgress = function(target_mc:MovieClip,
                                     numBytesLoaded:Number,
                                     numBytesTotal:Number) : Void
{
    var numPercentLoaded:Number = numBytesLoaded / numBytesTotal * 100;
    trace("onLoadProgress: " + target_mc + " is " +
          numPercentLoaded + "% loaded");
};

mclListener.onLoadComplete = function(target_mc:MovieClip,
                                     status:Number) : Void
```

```

{
    trace("onLoadComplete: " + target_mc);
};

// Register the listener with the MovieClipLoader
mclLoader.addListener(mclListener);

```

The code in Tutorial\section7.3 has been modified to initially load only container.swf instead of d3d9guide.swf and fxplayer.swf. Pressing F8 loads the HUD on demand and pressing F9 loads the main UI:

```

void GfxTutorial::ProcessEvent(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam,
                               bool *pbNoFurtherProcessing)
{
    ...

    else if (wParam == VK_F8)
    {
        const char *retval = pShellMovie->Invoke("_root.LoadFlashLevel",
                                                  "%s, %d", "fxplayer.swf", 5);
        GfxPrintf("_root.LoadFlash returns '%s'\n", retval);
    }
    else if (wParam == VK_F9)
    {
        const char *retval = pShellMovie->Invoke("_root.LoadFlashLevel",
                                                  "%s, %d", "d3d9guide.swf", 6);
        GfxPrintf("_root.LoadFlash returns '%s'\n", retval);
    }
    else if (wParam == VK_F2)
    {
        const char *retval = pContainerMovie->Invoke("_root.UnloadFlashLevel",
                                                      "%d", 5);
        GfxPrintf("_root.UnloadFlash returns '%s'\n", retval);
    }
    else if (wParam == VK_F3)
    {
        const char *retval = pContainerMovie->Invoke("_root.UnloadFlashLevel",
                                                      "%d", 6);
        GfxPrintf("_root.UnloadFlash returns '%s'\n", retval);
    }
}

```

The above code loads the HUD into level 5 and the main interface into level 6. Flash ActionScript levels are related to the Z order of the scene. Within a single GfxMovieView, movies in a higher level

render on top of movies with a lower level. Container.swf, the first movie loaded, is placed in level 0 by default.

Variables in levels can be referenced from other levels with the `_level` keyword. For example, the main interface in `_level6` can directly access the HUD's text field by changing `_level5.MessageText.text`. Alternatively the two movies can share data in Flash's global variable space through the `_global` keyword (e.g., both movies can access `_global.counter`). This has the advantage that when either movie is unloaded, variables in the `_global` namespace will not be lost. Movies can be swapped in and out of memory and see a consistent `_global` namespace. For more information on `_level`, `_global`, and the ActionScript variable namespace, please refer to the Adobe Flash documentation.

Run the application and press F8 to load the HUD. Then press F9 to load the main interface. Notice how there is a delay loading the main interface. This is because of the time required to load the embedded Chinese font. This process can be accelerated by pre-processing the SWF file into a GfX file (section 8) and by leveraging GfX's multithreaded loading capabilities to load a shared font file in the background (see *Font and Text Configuration Overview* on the developer center website).

Pressing F8 to bring up the HUD and then F5 to increment the counter added in Section 7.2 no longer works. The `SetVariable` method refers to `_root.counter` but the HUD is now loaded into the `_level5` namespace so the counter code should be changed as follows:

```
void GFxTutorial::ProcessEvent(HWND hWnd, UINT 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("_level5.counter");
            counter++;
            pHUDMovie->SetVariable("_level5.counter",
                GFxValue((double)counter));

            char str[256];
            sprintf_s(str, "testing! counter = %d", counter);
            pHUDMovie->SetVariable("_level5.MessageText.text", str);
        }
    }
    ...
}
```

The `_level5.counter` can also be accessed directly from ActionScript in `d3d9guide.swf`, which is loaded into level 6, enabling both interfaces to communicate directly without C++ code in the middle.

Run the application, press F8 to load the HUD and F5 to increment the counter. Press F2 to unload the HUD, then F8 to reload the HUD and continue incrementing the counter with F5. When the HUD was unloaded, the value of the counter was lost and counting restarted from zero. This is because the counter variable was stored in `_level5`. Change `_level5.counter` to `_global.counter` and try again. By storing the information in `_global`, it is preserved even when movies are loaded and unloaded.

8. 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 precomputed 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 d3d9guide.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\Section8.

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 GFxLoader::CreateMovie.

9. Next Steps

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

- In-game Flash render-to-texture (e.g., *Doom 3* consoles). See the GfXPlayer 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: <https://developer.scaleform.com/gfx?action=faq>.
- Scale9 window support documented in the Scale9Grid Overview on the documentation page: <http://developer.scaleform.com/gfx?action=doc>.
- 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 [GfXFileOpener](#).
- Custom images and textures through [GfXImageCreator](#) and [GfXImageLoader](#).
- Rendering 3D objects inside of GfX Flash movies.
- Controlling text rendering with [GfXFontCacheManager](#).
- Localization with [GfXTranslator](#).