Autodesk®
Scaleform®

Scaleform 集成應用指南

本文檔通過 DirectX 9 實例介紹了基本的 Scaleform 使用方法和 3D 引擎的集成應用。

作者： Ben Mowery
版本： 3.02
最近更新： 2011 年 4 月 14 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Autodesk Scaleform 聯繫方式：

文檔	Scaleform 4.2 集成應用指南
地址	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
網站	www.scaleform.com
郵箱	info@scaleform.com
電話	(301) 446-3200
傳真	(301) 446-3199

目錄

1 引言

Scaleform 作爲一個視覺化 UI 設計的中間件解決方案，其高性能已被充分證明，有了它，開發者通過 Adobe Flash Studio 可以高效率、低成本地創建富有現代感的 GPU 加速動畫用戶介面和向量圖形，而無需學習新工具的使用或者動畫處理方法。Scaleform 爲直接在 Flash Studio 創作遊戲用戶介面開闢了一條視覺化開發途徑，能夠無縫銜接工作中的各個環節。

除了普通的 UI 介面顯示，開發者還可以利用用 Scaleform 在 3D 場景中顯示 Flash 動畫，將 Flash 畫面映射到 3D 表面。同樣，3D 物件和視頻也可以在 Flash UI 當中顯示。由此，Scaleform 即可以作爲單獨的 UI 設計工具，也可以用來增強遊戲前端介面設計功能。

本使用指南涵蓋了 Scaleform 的安裝和使用等相關內容。其中提供了一個基於 Flash 用戶介面的增強型 DirectX ShadowVolume SDK 實例作爲詳細描述。

注釋：Scaleform 已被大多數主流遊戲引擎所集成。支援 Scaleform 的遊戲引擎中只需少量的代碼便可以使用。本指南主要目的是告知那些需要在各自的遊戲引擎中集成 Scaleform 的工程師，如何去實現集成，以及讓那些需要深入瞭解 Scaleform 性能的人員瞭解更多技術相關細節內容。

注釋：請確保參考本指南時，使用最新版本的 Scaleform。本指南適用於 Scaleform 4.0 或者更高版本。

注釋：本用用指南元件可能與一些較早的顯卡不相容。這是由於應用程式中所用的方法依賴於 DirectX SDK ShadowVolume 代碼與 Scaleform 不相容。當使用指南中的方法時是否會出現“cannot create renderer”錯誤提示，可以通過檢查 Scaleform Player 應用程式是否能順利執行來作判斷。

2 文檔概述

最新的多語種 Scaleform 文檔可以在 Scaleform 開發者中心下載到，網址爲：https://gameware.autodesk.com/scaleform/developer/。免費註冊後即可登錄站點。

當前文檔包括：

- Web 格式 Scaleform SDK 參考文檔。
- PDF 文檔。
- 字體概述：描述字體和文本渲染系統，並詳細介紹了藝術資源和 Scaleform C++ API 介面的配置，以實現適應國際化使用。
- XML 概述：描述了 Scaleform 中支援的 XML 元素。
- Scale9 Grid：解釋了如何利用 Scale9Grid 功能函數創建可變視窗、面板和按鈕。
- IME 配置：描述了如何在終端應用中集成 Scaleform IME，以及如何如何在 Flash 創建和設置 IME 文本輸入視窗。
- 動態擴展腳本：覆蓋了 Scaleform 動態腳本擴展 。

3 組建安裝和關聯編譯

3.1 安裝

下載最新 Windows 平臺的 Scaleform 安裝包。運行兩個安裝程式,保持其預設安裝路徑和選項。如有需要,Scaleform installer 將更新 DirectX。Scaleform 默認安裝目錄爲 C:\Program Files\Scaleform\GFx SDK 4.2。

目錄結構如下所示：

3rdParty

Scaleform 需要的擴展庫文件如 libjpeg 和 zlib。

Projects

包含了建立上述應用的 Visual Studio 工程文件。

App\Samples

原始程式碼實例和其他演示樣板

Apps\Tutorial

此教程的原始程式碼及專案。

Bin

包含了已編譯的演示二進位文件和 Flash 實例：

FxPlayer:	簡單的 Flash 文本 HUD。
Samples:	各種 Flash 實例的 FLA 原文件，包括用戶介面元素如按鈕、編輯框、鍵區、功能表和旋轉計數等。
Video Demo：	Scaleform 視頻示例和文件
Win32:	Scaleform Player 及其演示文件的預編譯二進位文件。
AMP:	用於運行 AMP 工具的 Flash 文檔。
GFxExport:	GFxExport 是一個預編譯工具，具有加速導入 Flash 內容的功能，詳細情況見第 7 小節

Include

Scaleform 便利頭文。

Lib

Scaleform 庫文件。

Resources

CLIK 元件和工具

Doc

第 2 節所描述的 PDF 文檔。

3.2 編譯演示文件

爲了驗證系統是否配置完成，能夠編譯 Scaleform。首先編譯如下演示文件，在 C:\Program Files (x86)\Scaleform\GFx SDK 4.1\Projects\Win32\Msvc90\Demos\GFx 4.2 Demos.sln 目錄中找到 Demos.sln 文件，在 Visual Studio 中打開.sln 文件，選擇 D3D9_Debug_Static 設置屬性並編譯該 GFxPlayer 工程。

檢查位於目錄 GFx SDK 4.2\Bin\Win32\Msvc90\GFxPlayer\GFxPlayer_D3D9_Debug_Static.exe 文件的更新時間，確認已成功編譯並運行該程式。

此程式和其他位於示例文件目錄 Start → All Programs → Scaleform → GFx SDK 4.2 → Demo 的 GFx Player D3Dx 應用程式都爲硬體加速 SWF 播放器。開發過程中，能用該工具來測試和定位 Scaleform Flash 重播功能。

3.3 在 Scaleform Player 中 SWF 重播定位

運行 Start → All Programs → Scaleform → GFx SDK 4.2 → GFx Players → GFx Player D3D9 程式。打開 C:\Program Files\Scaleform\GFx SDK 4.2\Bin\Data\AS2\Samples\SWFToTexture 目錄，拖放 3DWindow.swf 到應用程式中。

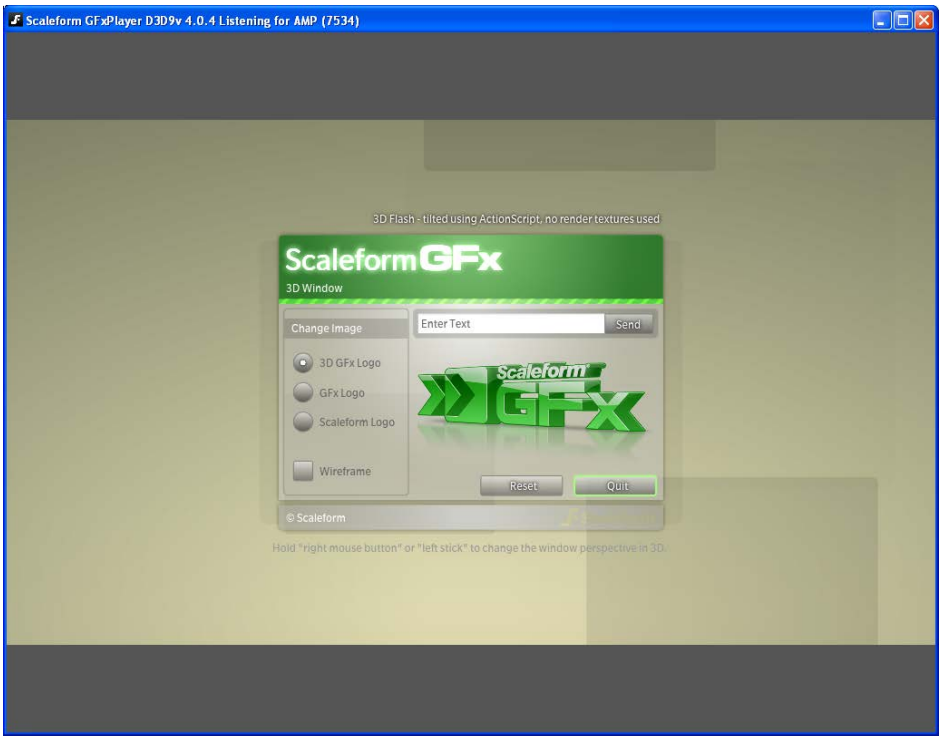

圖 1： Flash 實例的硬體加速重播

按 F1 快捷鍵顯示幫助視窗。嘗試如下選項：

1. 選擇 CTRL+F 組合鍵查看性能指標。當前 FPS 值將會出現在標題欄中。
2. 選擇 CTRL+W 組合鍵進入線性幀模式。注意 Scaleform 中如何通過將 Flash 內容轉換爲三角元素來優化硬體重播功能。再一次選中 CTRL+W，則離開線性幀模式。
3. 放大圖像操作，按下 CTRL 鍵並點擊滑鼠左鍵，然後將滑鼠上下移動即可。
4. 圖像變形操作，按下 CTRL 鍵並點擊滑鼠右鍵，然後移動滑鼠即可。
5. 選擇 CTRL+Z 組合鍵返回到普通視圖。
6. 選擇 CTRL+U 功能鍵進入全屏重播模式。
7. 選擇 F2 快捷鍵分析動畫參數，包括記憶體使用情況等。

注意到如何銳化按鈕邊角等邊緣曲線部分，使得在近距離觀看時仍顯得曲線平整。當視窗放大或縮小時，Flash 也隨著放大和縮小。向量圖的優勢之一是可以放大縮小到任何比例以適應任何解析度。而點陣圖通常需要在 800x600 和 1600x1200 不同解析度下準備兩張不同尺寸的圖像。

Scaleform 同時也支援傳統的點陣圖顯示，如螢幕右側中間的“Scaleform”圖示。任何設計師能夠在 Flash 中創作的絕大多數效果，在 Scaleform 中都能夠描繪出來。

放大“3D Scaleform Logo”單選按鈕，選擇 CTRL+W 組合鍵切換到線性幀模式。注意到圓弧形被柵格化成三角形狀。重復多次選擇 CTRL+A 組合鍵進入反鋸齒模式。在邊緣反鋸齒（EdgeAA）模式，三角

圖元點陣填充原型邊緣來實現反鋸齒效果。這就是全屏反鋸齒（FSAA）視頻卡所帶來的超強性能。該操作需要四倍的視頻幀緩存區和四倍的圖元渲染運算消耗來實現 AA 效果。Scaleform 的 EdgeAA 專利技術利用了向量物件表示法作爲反鋸齒應用，該效果得益於那些絕大多數典型的彎曲邊緣和大尺寸文本顯示區域。儘管，三角計數將增加，性能影響是可控的，因爲圖形元（DP）的數量仍然是個常數。顯然，使用視頻卡的反鋸齒功能和 EdgeAA 功能需要更多消耗，爲了高效運作，可以將這些屬性設置禁止使用或者適當調整。

Scaleform Player 工具用來調試 Flash 文件並測試性能指標。打開任務管理器查看 CPU 使用率。CPU 使用率隨著 Scaleform 渲染幀的數量的增多而增加。選中 CTRL+Y 組合鍵查看顯示刷新的幀速率，典型情況下爲 60 幀每秒。需要注意降低 CPU 使用率至關重要。

注釋：當測試您的應用工程時，務必運行發佈（release ）編譯模式而不要運行調試（debug）編譯模式的 Scaleform，由於調試（debug）編譯模式的 Scaleform 不提供性能優化功能。

3.4 編譯基本實例

如果使用 Scaleform 4.2 的话，把 Scaleform\GFx SDK 4.2\Apps\Tutorial 中的 SLN 项目文件打开。可以在 Scaleform-> GFx SDK 4.2-> Tutorial 的 Windows 开始菜单中找到用于 for Visual Studio 2005 和 Visual Studio 2008 开发平台的解决方案。確保工程文件配置設置爲 “Debug” 模式並運行應用程式。

圖 2：默認用戶介面下的 ShadowVolume 應用畫面

3.5 *Scaleform 編譯關聯項*

但新建一個 Scaleform 工程時，在編譯之前 Visual Studio 中有些必須設置的選項。本 Tutorial 應用程式已經在適當的位置有相關的參考索引，只需按照其步驟執行。請牢記$(GFXSDK)是基本 SDK 安裝目錄的環境變數。默認路徑爲 C:\Program Files\Scaleform\GFx SDK 4.2\；如果你的庫文件在不同的路徑，需要改$(GFXSDK)環境變數指向系統中庫文件所在的路徑。在這些例子中我們將使用“Msvc90”；如果你使用 Visual Studio 2008 或者 Visual Studio 2010，則需要對應的使用 Msvc90 或 Msvc10 參數。

將 Scaleform 添加到工程所在的目錄，包括調試模式和發佈模式的目錄下：

$(GFXSDK)\Src
$(GFXSDK)\Include

將下列文字粘貼到 Visual Studio 的“Additional Include Directories”區域。

如果在用 AS3，请务必直接将必需的 AS3 类注册文件包含在您的应用程序中的 “GFx/AS3/AS3_Global.h” 之中。开发者可以将此文件自定义为排除不需要的 AS3 类。有关更多详情，请参阅文档 Scaleform LITE 自定义。

將以下庫文件所在的目錄路徑添加到調試編譯配置中的路徑搜索中：

$(DXSDK_DIR)\Lib\x86
$(GFXSDK)\3rdParty\expat-2.1.0\lib
$(GFXSDK)\Lib\$(PlatformName)\Msvc90\Debug_Static\
$(GFXSDK)\3rdParty\zlib-1.2.7\Lib\$(PlatformName)\Msvc90\Debug
$(GFXSDK)\3rdParty\jpeg-8d\Lib\$(PlatformName)\Msvc90\Debug

將下列文本粘貼到“Additional Library Directories”區域：

"$(DXSDK_DIR)\Lib\x86";
"$(GFXSDK)\3rdParty\expat-2.1.0\lib";
"$(GFXSDK)\Lib\$(PlatformName)\Msvc90\debug";
"$(GFXSDK)\3rdParty\zlib-1.2.7\Lib\$(PlatformName)\Msvc90\Debug";
"$(GFXSDK)\3rdParty\jpeg-8d\Lib\$(PlatformName)\Msvc90\Debug"

注釋：改變 Msvc90 參數以符合所用的 Visual Studio 版本。

相應的將發佈版本的庫文件添加到搜索路徑中，以便發佈和編譯資訊配置：

$(DXSDK_DIR)\Lib\x86
$(GFXSDK)\3rdParty\expat-2.1.0\lib
$(GFXSDK)\Lib\$(PlatformName)\Msvc90\Release\
$(GFXSDK)\3rdParty\zlib-1.2.7\Lib\$(PlatformName)\Msvc90\Release
$(GFXSDK)\3rdParty\jpeg-8d\Lib\$(PlatformName)\Msvc90\Release

將下列文本粘貼到“Additional Library Directories”區域（發佈和配置資訊）：

"$(DXSDK_DIR)\Lib\x86";
"$(GFXSDK)\3rdParty\expat-2.1.0\lib";
"$(GFXSDK)\Lib\$(PlatformName)\Msvc90\Release";
"$(GFXSDK)\3rdParty\zlib-1.2.7\Lib\$(PlatformName)\Msvc90\Release";
"$(GFXSDK)\3rdParty\jpeg-8d\Lib\$(PlatformName)\Msvc90\Release"
"$(GFXSDK)3rdParty\libpng\Lib\$(PlatformName)\Msvc90\Release"
注釋：改變 Msvc90 參數以符合所用的 Visual Studio 版本。

最後，添加 Scaleform 庫和相關文件：

libgfx.lib
libAS2.lib
libAS3.lib
libgfxexpat.lib
libjpeg.lib
zlib.lib
libpng.lib
libgfxrender_d3d9.lib
libgfxsound_fmod.lib

添加相同的库到发布和编译配置设置中。

确保同一个应用程序在调试模式和发布模式下均进行编译和链接。作为参考，更改过的.vcproj 文件中包含了 Scaleform include 目录信息，链接设置位于 Tutorial\Section3.5 目录。

3.5.1 AS3_Global.h and Obj\AS3_Obj_Global.xxx 文件

開發者必須注意:AS3_Global.h 與 Obj\AS3_Obj_global.xxx 是完全無關的檔。
AS3_Obj_Global.xxx 檔包含對所謂“全域”ActionScript 3 物件的實現。每個 swf 檔均包含至少一個稱為“腳本”(script) 的物件,這是一個全域物件。還有一個類 GlobalObjectCPP,它是用 C++ 實現的所有類的一個全域物件。這是 Scaleform VM 實現所特有的。

AS3_Global.h 有一個完全不同的用途。此檔包含 ClassRegistrationTable 陣列。此陣列的用途是引用實施相應 AS3 類的 C++ 類。沒有此引用,代碼就會被一個連結程式排除在外。因此,必須在可執行程式中定義 ClassRegistrationTable,否則就會收到一個連結程式錯誤。為此,我們的每個演示播放機均包含了 AS3_Global.h。

將 ClassRegistrationTable 置入一個 include 檔並要求開發者將其包含在內的全部目的就是允許自訂 ClassRegistrationTable(出於有可能減小代碼大小的目的)。達此目的的最佳方法是製作 AS3_Global.h 的

一個副本,注釋掉不需要的類(然後,這些類就不會被連結進去),並把自訂的版本包含在您的應用程式中。

不過,有一個與此優化相關的陷阱。由於 AS3 VM 中的名稱解析在運行時發生,因而就有可能找不到從表中解釋掉的需要的類。因此,如果想要消除“不必要的”類,請確保您的應用程式在此操作之後仍然能夠正常工作。

4 遊戲引擎集成

Scaleform 提供了集成層，支援爲大多數主流 3D 遊戲引擎：包括 Unreal® Engine 3, Gamebryo™, Bigworld®, Hero Engine™, Touchdown Jupiter Engine™, CryENGINE™, 和 Trinigy Vision™ 引擎。這些遊戲在開發中將 Scaleform 集成到其引擎只需少量或者無需編寫代碼。

本節描述了如何將 Scaleform 集成到自定義 DirectX 應用當中。DirectX ShadowVolume SDK 實例是一個標準的 DirectX 應用程式，擁有典型遊戲所具有的應用過程和遊戲環節。如前一節所述，該應用程式採用基於 DXUT 的 2D 介面覆蓋來渲染 3D 場景。

本使用指南覆蓋了 Scaleform 集成到應用程式中用基於 Flash 的 Scaleform 介面代替默認 DXUT 用戶介面的整個過程。

DXUT 框架不顯露應用程式的隱含渲染環，而顯露了底層抽象層的細節調用過程。需理解這些與標準 Win32 DirectX 渲染環相關的步驟，則對比 Scaleform SDK 附帶的 GFxPlayerTiny.cpp 實例原始檔案。

4.1 *Flash 表現手法*

集成過程的第一步爲將 Flash 動畫在 3D 背景中表現出來。這包含了實例中由一個 GFx::Loader 物件管理應用工程中 Flash 內容的全局調用，GFx::MovieDef 中包含了 Flash 內容，GFx::Movie 物件顯示了動畫的一個播放實例。另外，Render::Renderer2D 物件和 Render::HAL 物件將實例化，作爲本實例 DirectX 中 Scaleform 和專用渲染 API 函數之間的介面。我們還討論了如何根據釋放的設備事件有效分配資源，如何處理全屏/視窗轉換。

本節 ShadowVolume 實例版本修改步驟在 4.1 小節中有描述。文檔中所示的以下相關代碼僅供參考，並不全面。

步驟 #1: 添加頭文件

爲 ShadowVolume.cpp 添加必要的頭文件。

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

某些 Scaleform 物件在視頻渲染中需要用到，被封裝爲一個新的類添加到 GFxTutorial 應用程式中去。爲使代碼保持簡介，保持 Scaleform 狀態在統一在一個類中具有一定優勢，只需一個刪除調用就可以釋放所有的 Scaleform 物件。在以下步驟中將會對這些物件之間的相互作用做詳細介紹。

```
//每個應用套裝程式含一個 GFx::Loader
Loader           gfxLoader;

//每個 SWF/GFx 文件包含一個 GFxMovieDef
Ptr<MovieDef>   pUIMovieDef;

//每個視頻播放實例包含一個 GFx::Movie
Ptr<Movie>  pUIMovie;

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

步驟#2：初始化 GFx::System

Scaleform 初始化的第一步爲初始化 GFx::System 物件來分配 Scaleform 記憶體。在 WinMain 中我們添加如下代碼行：

```
//每個應用套裝程式含一個 GFx::System 物件
GFx::System           gfxInit;
```

GFx::System 物件必須在第一個 Scaleform 調用時被獲取，在 Scaleform 結束前不能被釋放，這就是其位於 WinMain 函數頭的原因。GFx::System 作爲初始化，這裏使用 Scaleform 的默認存儲空間，但是可以被應用程式的自定義存儲空間所覆蓋。本 Tutorial 的目的爲簡化 GFx::System 物件並未對其深入操作。

GFx::System 在應用程式結束時必須被釋放，意味著其不能作爲一個總體變數。當 GFxTutorial 物件被釋放時 GFx::System 物件也立即銷毀。

根據特別應用所建立的體系結構，相對於創建一個 GFx::System 物件實例，調用 GFx::System::Init() 和 GFx::System::Destroy() 靜態函數來的更加簡便。

步驟#3: 導入和創建渲染器

其餘的 Scaleform 初始化在應用工程的 WinMain 函數在其自身 InitApp() 初始化完成之後執行。在調用 InitApp()函數之後添加以下代碼：

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

GFxTutorial 包含了一個 GFx::Loader 物件。一個應用程式通常只有一個 GFx::Loader 物件，該物件用來導入 SWF/GFx 內容以及在資源庫中存儲 SWF/GFx 內容，以便資源的重用。相互獨立的 SWF/GFx 文件可以共用圖像和字體等資源以節省記憶體空間。GFx::Loader 還包含了狀態配置集，如 GFx::Log，作爲調試腳本。

GFxTutorial::InitGFx()函數的第一步爲設置 GFx::Loader 物件狀態。GFx::Loader 物件傳遞調試跟蹤資訊給 SetLog 中提供的控制碼。調試輸出資訊包含了 Scaleform 功能函數執行出錯資訊，輸出到腳本當中，這些資訊起到很好的作用。本例中我們採用 GFxPlayerLog 控制碼的默認方式，其將消息直接輸出到控制臺視窗，但是，GFx::Log 子集同時可完成與遊戲引擎調試腳本資訊的整合。

```
// 初始化腳本—Scaleform輸出錯誤資訊到腳本文件
// 資料流程
gfxLoader->SetLog(Ptr<Log>(*new PlayerLog()));
```

GFx::Loader 通過 GFx::FileOpener 類來讀取文件內容。默認方式下從光碟文件中讀取資料，同時GFx::FileOpener子集支援從記憶體或者其他資源檔案讀取資料的自定義方式。

```
// 導入默認文件
Ptr<FileOpener> pfileOpener = *new FileOpener;
gfxLoader->SetFileOpener(pfileOpener);
```

Render::HAL是一個通用介面，使Scaleform輸出圖形到各種硬體設備。我們創建一個D3D9渲染器實例並與導入器相關聯。渲染物件負責管理D3D設備、紋理和頂點緩存，這些Scaleform將會用到，在InitHAL模式下，我們需要爲Render HAL物件傳遞一個IDirect3Device9指標，這個指標由遊戲初始化，通過它Scaleform能夠創建DX9資源並順利渲染UI元素。

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

上述代碼用到了 Scaleform 的 Render::HAL 子集中提供的 Render::D3D9::HAL 物件，更好的控制 Scaleform 渲染行爲和緊密集成。

步驟#4：導入 Flash 動畫

接下來 GFx::Loader 已經可以導入動畫了。被導入的動畫表示爲 GFx::MovieDef 物件。這些 GFx::MovieDef 物件包含了動畫的所有共用資料，如幾何圖形和紋理。但不包含單個實例的資訊，如單個按鈕的狀態、動態腳本變數、當前動畫幀等資訊。

```
// 導入動畫
pUIMovieDef = *gfxLoader.CreateMovie(UIMOVIE_FILENAME,
                                     Loader::LoadKeepBindData |
                                     Loader::LoadWaitFrame1, 0);
```

LoadKeepBindData變數指向了系統記憶體中的紋理圖像內容副本，在應用中需要重新創建D3D設備時將會用到。該變數在遊戲控制系統和紋理已知且不會丟失的情況下不是必須元素。

LoadWaitFrame1指令通知CreateMovie函數直到第一幀導入後才返回值。這在使用GFx::ThreadTaskManager時非常重要。

最後一個參數是可選的，並說明了所用的記憶體區。有關創建和使用記憶體區的資訊，請參閱 Memory System Overview 。

步驟#5：動畫實例創建

在描繪動畫前，必須從 GFx::MovieDef 物件創建 GFx::Movie 實例。GFx::Movie 包含了單個運行實例的相關狀態資訊，如當前幀、動畫時間、按鈕狀態和動態腳本變數等。

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

傳遞給 CreateInstance 函數的參數爲第一幀是否導入狀態。如果該狀態爲 false，則能夠 在位於第一幀的動態腳本執行前改變 Flash 和動態腳本的狀態。最後一個參數是可選的，並說明了所用的記憶體區。有關創建和使用記憶體區的資訊，請參閱 Memory System Overview 。

動畫實例一旦創建，第一幀調用 Advance()函數初始化，只有當傳遞 false 給 CreateInstance 函數時才需要這個步驟。

```
// 指向動畫第一幀
```

```
pUIMovie->Advance(0.0f, 0, true);

// 記錄當前時間，測算當前幀到下一幀的時間消耗
MovieLastTime = timeGetTime();
```

傳遞給 Advance 函數的第一個參數爲時差，第二個參數爲當前幀到下一幀的時間間隔。記錄當前系統時間用來計算當前幀和下一幀的時差。

爲了在 3D 場景中表現動畫透明效果：

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

沒有上述函數調用，動畫在渲染時將會用 Flash 文件中已定義的背景色覆蓋 3D 場景。

步驟#6：設備初始化

Scaleform 在渲染时必须从 Render::D3D9::HAL 获取 DirectX 设备的句柄和参数描述信息。在 D3D 设备创建和 Scaleform 被要求执行渲染前需要调用 Render::D3D9::HAL::InitHAL 函数。當視窗大小改變或者全屏/視窗切換使 D3D 設備控制碼發生變化時需要重新調用 InitHAL 函數。

ShadowVolume 中的 OnResetDevice 功能函數在設備創建並初始化或者設備重定時由 DXUT 框架調用。以下代碼對應於 GFxTutorial 中的 OnResetDevice 方法：

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

InitHAL()函数的调用传递 D3D 设备和参数描述信息到 Scaleform。HAL_NoSceneCalls 函數參數表示 Scaleform 不會調用 DirectX 中的 BeginScene() 和 EndScene()函數。這個設置是必不可少的，因爲 ShadowVolume 實例應用程式中在 OnFrameRender 函數中已經調用了該兩個函數。

步驟#7：設備丟失

當視窗大小改變或應用程式切換爲全屏模式時，D3D 設備將丟失。所有 D3D 介面資訊包或頂點緩存和紋理必須重新初始化。ShadowVolume 在 OnLostDevice 函數回收中釋放介面資訊。Render::HAL 獲得設備丟失資訊後將用 GFxTutorial 中的 OnLostDevice 方法釋放其 D3D 資源。

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

本步驟和前面一個步驟解釋了 DXUT 框架回收系統中初始化和設備丟失的相關操作。需要一個基本的 Win32/DirectX 渲染環例子，請查看 Scaleform SDK 中的 GFxPlayerTiny.cpp 實例源代碼。

步驟#8：資源分配和回收

由於所有 Scaleform 物件都包含在 GFxTutorial 物件當中，這樣資源回收操作變得很簡單，只需在 WinMain 函數結尾處刪除整個 GFxTutorial 物件即可：

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

另外一個需要考慮的問題是如頂點緩存等 DirectX 9 資源回收。這在 Scaleform 中得到很好的處理，但在遊戲主迴圈中發生資源分配和回收需要瞭解 InitHAL() 和 ShutdownHAL()函數所起的作用。

在 DirectX 9 實現中，InitHAL 分配 D3DPOOL_DEFAULT 資源，包括頂點緩存。當與您自己的引擎集成時，設法把 InitHAL 放置到合適的位置來分配 D3DPOOL_DEFAULT 資源。

ShutdownHAL 將釋放 D3DPOOL_DEFAULT 資源。應用程式使用 DXUT 框架，包括 ShadowVolume，應當在 DXUT 中的 OnResetDevice 調用中分配 D3DPOOL_DEFAULT 資源，而在 OnLostDevice 調用中釋放資源。GFxTutorial::OnLostDevice 方法中的 ShutdownHAL 調用需與 GFxTutorial::OnResetDevice 中的 InitHAL 調用相配合。

當與您自己的引擎集成時，設法調用 InitHAL 和 ShutdownHAL 函數和其他相關函數來分配和釋放 D3DPOOL_DEFAULT 資源。

步驟#9：視窗設置

必須在螢幕上確定一個特定的視窗以便描繪動畫。在本實例中，動畫佔據了整個視窗。由於螢幕的解析度可以改變，每次在 D3D 設備重定時我們也將重定視窗，可添加如下代码到 GFxTutorial::OnResetDevice 來完成本操作：

```
// 使用視窗用戶端大小尺寸作爲視窗大小。
RECT windowRect = DXUTGetWindowClientRect();
DWORD windowWidth = windowRect.right - windowRect.left;
DWORD windowHeight = windowRect.bottom - windowRect.top;
pUIMovie->SetViewport(windowWidth, windowHeight, 0, 0,
```

```
                                windowWidth, windowHeight);
```

SetViewport 函數中的前兩個參數描述了使用的畫面緩存區大小，通常爲 PC 應用程式的視窗大小。其餘四個參數描述了 Scaleform 可以描繪的窗口大小。

畫面緩存區大小參數與 OpenGL 和其他平臺相容，不同平臺間具有不同的坐標系數或畫面緩存區的大小資訊不可獲取。

Scaleform 提供了一系列函數用於控制 Flash 畫面在視窗範圍內的縮放和移動。在 4.2 小節中應用程式準備好可以執行時我們可以檢查這些功能選項。

步驟#10：描繪 DirectX 場景

描繪操作在 ShadowVolume 的 OnFrameRender()函數中實現。所有的 D3D 描繪調用都是在函數 BeginScene() 和 EndScene()之間執行。我们调用 EndScene()之前将调用 GFxTutorial::AdvanceAndRender()。

```
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()函數將動畫向前推進 deltaTime 秒。動畫的播放速度由應用程式所在的當前系統時間來控制。必須提供即時時鐘給 GFx::Movie::Advance，確保動畫能夠在不同的的硬體配置中能夠正確重播。

步驟#11：保留描繪狀態

Render::Renderer2D::Display 調用 DirectX 在 D3D 設備上描繪動畫幀。考慮到性能因素，D3D 設備狀態資訊，如混合模式和紋理存儲設置等資訊將不被保留，因此調用 Render::Renderer2D::Display 後 D3D 的設備狀態資訊將有所不同。有些應用程式可能受到負

面影響。最直接的做法是在調用之前先保存設備狀態資訊，調用之後再恢復這些資訊。遊戲引擎在 Scaleform 渲染後重新初始化必要的狀態資訊，這樣能達到優越的性能。本指南中用到了 DX9 狀態塊函數來簡單的操作保存和恢復狀態資訊。

DX9 狀態塊在應用程式整個執行過程中被分配，並在 GFxTutorial::AdvanceAndRender()函數調用之前和之後被使用。

```
// 調用GFx前保存DirectX狀態資訊
g_pStateBlock->Capture();

// 描繪畫面幀，推進時間計數
gfx->AdvanceAndRender();

// 恢復DirectX狀態資訊，避免干擾遊戲描繪狀態
g_pStateBlock->Apply();
```

步驟#12：禁止默認 UI

最後一步爲禁止 DXUT 默認用戶介面。這由注釋相關部分代碼的方法來完成，可參考 4.1 小節 ShadowVolume.cpp 文件。將該文件與之前小節的代碼相對比查看改變部分。所有與 DXUT 相關的修改都有如下注釋：

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

現在，我們在 DirectX 應用中已經擁有了一個硬體加速功能的 Flash 動畫。

圖 3：GFx Flash 用戶介面的 ShadowVolume 應用程式

4.2 縮放模式

GFx::Movie::SetViewport函數調用使得視窗與螢幕解析度保持一致。若螢幕的縱橫比與Flash畫面的縱橫比例不協調，則Flash畫面將扭曲。Scaleform提供如下函數：

- 保持畫面縱橫比例或者自由延伸.。
- 移動畫面至中間、角落或者視窗邊沿。

這些函數在4:3或者寬屏顯示中繪製同個圖形非常有用。Scaleform的優勢之一就是其可伸縮向量圖使得能夠自由的與各種解析度的顯示尺度相適配。而傳統的點陣圖在不同解析度下需要設計不同尺寸的點陣圖分別顯示。例如，一個設置成爲低解析度的800x600點陣圖，另外一個設置成高解析度的1600x1200點陣圖。Scaleform可以把同一個向量圖縮放到任何比例的解析度尺寸。除此之外，在某些遊戲場景元素描繪中點陣圖更加合適，Scaleform也完全支援。

GFx::Movie::SetViewScaleMode定義了縮放操作。爲了保證畫面能夠與視窗相適應而不導致原始尺寸比列變形，可調用GFxTutorial::InitGFx()函數與其他調用相結合來創建GFx::Movie物件：

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

SetViewScaleMode函數的參數含義在在線文檔中有闡述並說明如下：

SM_NoScale	將Flash畫面恢復到原始尺寸
SM_ShowAll	維持原始比例，縮放到視窗大小
SM_ExactFit	不保持原始比例，縮放到視窗大小。填充整個視窗，但可能會發生變形。
SM_NoBorder	縮放到視窗大小，填充整個視窗，並保持原始比例，增加畫面剪輯。

SetViewAlignment能夠充分地控制畫面在視窗中的位置。當需要維持原始縱橫比例，使用SM_NoScale 或SM_ShowAll參數時，視窗的某些部分將空白。在此情況下選擇SetViewAlignment參數來確定畫面在視窗中的對齊位置，如下代碼表示按鈕的介面垂直居中並右對齊：

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

試著改變SetViewScaleMode 和SetViewAlignment的參數，並調整視窗尺寸，查看應用程式的變化情況。

SetViewAlignment函數只在SetViewScaleMode函數設置爲SM_NoScale默認參數時才起作用。更多複雜的關於對齊、縮放和移位元需求，Scaleform支援動態腳本擴展定位動畫位置和尺寸大小，這些實例動態腳本在文件d3d9guide.fla中有所提供。

也可以通过ActionScript代替C++进行缩放大小和对齐方式参数设置。SetViewScaleMode 和SetViewAlignment可以修改ActionScript Stage类所描述的属性(State.scaleMode, Stage.align)。

4.3 輸入事件處理

ShadowVolume 管線做了修改用來描繪 Scaleform 中的 Flash；我們現在需要與 Flash 播放動畫交互。例如，滑鼠移動到按鈕上方使得按鈕能突出顯示，在文本框輸入文本能顯示新的字元。

GFx::Movie::HandleEvent 傳遞 GFx::Event 物件來描述事件類型和其他資訊，如按鍵或者滑鼠動作。應用程式簡單得構造輸入時間，並傳遞給對應視圖的 GFx::Movie。

4.3.1 滑鼠事件

ShadowVolume 在 MsgProc 調用中獲得 Win32 輸入事件。爲 GFxTutorial::ProcessEvent 增加一個調用來運行相關代碼使得 Scaleform 能夠處理過程事件。以下代碼表示的動作有 WM_MOUSEMOVE, WM_LBUTTONDOWN 和 WM_LBUTTONUP：

```
void ProcessEvent(HWND hWnd, unsigned uMsg, WPARAM wParam, LPARAM lParam,
                  bool *pbNoFurtherProcessing)
{
      int mx = LOWORD(lParam), my = HIWORD(lParam);
      if(pUIMovie)
```

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

Scaleform 定位滑鼠焦點位於視窗的左上方，而與畫面的原始尺寸無關。下面例子就闡明了本含義：

實例#1: 視窗與螢幕尺寸匹配

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

本例中無需任何轉換：由於畫面被定位在座標(0, 0)位置，滑鼠在畫面視窗中的位置也與左上方相關聯。窗口的尺度與動畫原始解析度尺度的對應由 Scaleform 內部實現。

實例#2：視窗比螢幕小，定位于螢幕的左上角

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

同樣，本例中也無需做任何轉換。由於視窗被縮小，按鈕的尺寸和位置也隨之改變。但是不管是 HandleEvent 事件還是視窗螢幕的基準仍定位在視窗的左上角，無需任何轉變。視窗尺寸縮放和動畫原始尺寸由 Scaleform 內部處理。

實例#3：視窗比螢幕小居中顯示

```
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);
```

在本例子中螢幕視窗的轉換需要對應基準座標。動畫不在位置(0,0)，新的位置位於座標爲(screen_width / 2 – movie_width / 2, screen_height / 2 – movie_height / 2)處，必須根據螢幕座標計算準確的座標位置。

注意如果 Flash 畫面通過 GFx::Movie::SetViewAlignment 居中或者用其他方式對齊，無需執行轉換。只要滑鼠動作與 GFx::Movie::SetViewport 相關聯。SetViewAlignment 和 SetViewScaleMode 的對齊和縮放操作將由 Scaleform 內部來處理。

4.3.2 鍵盤事件

鍵盤事件也由 GFx::Movie:HandleEvent 來處理。有兩類鍵盤事件：GFx::KeyEvent 和 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)
```

GFx::KeyEvent 類似於原始的掃描碼；而 GFx::CharEvent 類似於 ASCII 字元處理器。在 Windows 作業系統中，GFx::CharEvent 事件由 WM_CHAR 消息產生；GFx::KeyEvents 事件由 WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN 和 WM_KEYUP 消息產生。以下爲相關的幾個例子：

- 同時按住 ‘c’ 鍵和 SHIFT 鍵盤:：一個 GFx::KeyEvent 事件即產生，對應的 WM_KEYDOWN 消息描述的過程爲：
  - ‘c’ 鍵已點擊，獲得了鍵值
  - 鍵盤已被按下
  - SHIFT 鍵有效

- 同時 GFx::CharEvent 應該觸發回應 WM_CHAR 消息並傳遞 “觸發”的 ASCII 碼字元‘C’ 給 Scaleform。

- 一點‘c’鍵被釋放，將產生 GFx::KeyEvent 事件對應消息 WM_KEYUP。當鍵釋放時無需傳遞 GFx::CharEvent 事件。

- 按下 F5 鍵盤： 當鍵盤按下時隨即產生 GFx::KeyEvent 事件，當鍵彈起產生第二個事件。由於 F5 不代表任何可列印的 ASCII 碼，所以整個過程不需要產生 GFx::CharEvent 事件。

在鍵盤按下和彈起時產生獨立的 GFx::KeyEvent 事件。爲了保持平臺無關性，各個鍵值在 GFx_Event.h 頭文件中有統一定義以配合 Flash 內部使用。GFxPlayerTiny.cpp 例子中和 Scaleform Player 程式中均包含了鍵值，將 Windows 的按鍵碼轉換成 Flash 內部使用的鍵值。本節所描述的最終代碼還包含 ProcessKeyEvent 函數，可在與自定義 3D 引擎集成中重用：

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

從 Windows 的 WndProc 函數調用相關消息，包括 WM_CHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN 和 WM_KEYUP 消息。相應地觸發 Scaleform 事件並傳遞給 pMovie。

發送 GFx::KeyEvent 和 GFx::CharEvent 事件是非常重要的。例如，大多數文本框需回應 GFx::CharEvent 事件，因爲文本框需要獲取可列印 ASCII 字元。同樣的，文本框也需獲取 Unicode 字（比如用中文輸入框 IME），. 在 IME 輸入情況下，原始鍵值是沒有用的，只有最終的字元（通常由 IME 中的若干個鍵值組合而成）才輸入到文本框。於此相反，下拉清單需要通過 GFx::KeyEvent 事件截取 Page Up 和 Page Down 鍵，由於該建不是用來顯示字元，而是用來做控制用。

該功能的相關代碼在 4.3 小節中有詳細描述。運行程式並移動滑鼠至按鈕上方。按鈕將突出顯示，那些無需與 3D 引擎集成的實例將能夠很好運行。點擊“Settings” 出現 DX9 配置介面，無需任何 C++代碼，因爲 d3d9guide.fla 用動態腳本來實現此簡單的邏輯功能。

同時命令鍵也可以傳遞給 Flash，空白鍵控制 UI 是否可見。固定 UI 觀測應用程式的幀速率並測算 Scaleform 在應用程式中的性能。與 DXUT 介面對比幀速率查看在 DXUT 和 Scaleform 中的性能比較。務必用發佈編譯方式的程式來做對比，不要用調試編譯模式的程式。通常 Scaleform 不會對應用程式產生多大影響，也不會比 DXUT 速度要慢很多。請記住 Scaleform 介面支援全部功能的 Flash 動畫（如文本輸入框的隱現）、動畫視窗轉換，同時能夠任何比例地縮放按鈕、文本和其他 UI 元素。

需查看脚本代码中的键盘处理代码，点击“Change Mesh”并输入到文本框。还有些小问题在文中接下来部分将予以解决。注意到当点击“Change Mesh”按钮后画面中开打一个文本输入框。则在 Flash 中通过矢量动画很容易实现，但是在传统的位图界面中就比较困难。在位图中实现此动画效果需要添加额外代码，使动画导入比较缓慢，渲染开销较大，且使用代码非常枯燥。

4.3.3 點擊測試

運行應用程式並移動滑鼠，同時按下滑鼠左鍵改變指標方向進入 3D 的世界。然後移動滑鼠至其中一個用戶介面元素的上方並重復此操作。儘管滑鼠點擊在介面中產生了預期的回應，仍然使鏡頭在移動。

在 UI 和 3D 中的焦点控制可以用 GFx::Movie::HitTest 来解决。此函数可以确定视窗是否产生一个 Flash 中的渲染事件。在处理完一个鼠标事件后，改变 GFxTutorial::ProcessEvent 以调用 HitTest，若事件在用户 UI 界面元素上发生，通知 DXUT 框架不必传递该事件到镜头以处理：

```
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 鍵盤焦點

運行應用程式點擊“Change Mesh”按鈕打開文本輸入框。輸入文字到輸入框，可以看到鍵盤的輸入字元，代碼如 4.3.2 中所示。但是，輸入 W,S,A,D 和 Q 到輸入文本框，同時也移動了 3D 視鏡。鍵盤事件被 Scaleform 處理，但也同時傳遞給了 3D 視鏡。

要解決這個問題需要判斷是否輸入文本框已獲得焦點。6.1.2 小節將描述動態腳本如何用來發送事件給 C++以確定事件處理焦點。

5 文字本地化和字體介紹

5.1 字體概述和容量

Scaleform 提供了一套高效、易擴展的字體和本地化系統。多樣化的字體、點陣大小和風格能同時有效地顯示出來，且佔用較低的記憶體空間。字體資料可以從內嵌字體資料的 Flash 文件、共用字體庫、作業系統和直接 TTF 字體庫中獲取。基與向量圖的字體壓縮資料在大尺寸亞洲文字應用中減少了記憶體空間。字體的支援具有跨平臺功能，在控制臺系統、Windows 和 Linux 中都能夠支援。Scaleform 關於字體和國際化覆蓋範圍的完整文檔可在開發中心文檔頁面中找到：
http://gameware.autodesk.com/scaleform/developer/?action=doc。

通常字體的描繪手段爲爲每個文字的不同的字體準備不同的紋理外觀，映射到不同的字體使用中去。附加的字體紋理需要不同的尺寸和風格。如拉丁文文字總的資料存儲量還可以接受，但是亞洲文字有 5000 個不同的文字，實現起來就不大方便。需要大量的記憶體和處理時間。同時描繪不同文字大小和風格更加不可能。

Scaleform 用動態字體緩存來解決該問題。字元根據顯示需要放入到緩存，緩存中文字資料隨時可更新和替換。不同的文字尺寸和字體風格可以共用公共緩存，Scaleform 用到了智慧的輪廓壓縮演算法來實現共用。Scaleform 使用向量字體，這意味著只需單個 TTF 字體存儲在記憶體，可以用來表示各種大小的字體。另外，Scaleform 還支援“仿斜體”和“仿粗體”功能，使得單個字體向量能夠按照要求表示爲斜體和粗體，節省額外記憶體開銷。

超大字體可以通過柵格鑲嵌直接描繪出來。這解決了在遊戲開始螢幕標題中超大文字顯示的問題。在 Bin\Data\AS2\Samples\FontConfig 目錄中找到字體配置實例，將 sample.swf 文件拖放到打開的 Scaleform Player D3D9 窗口。

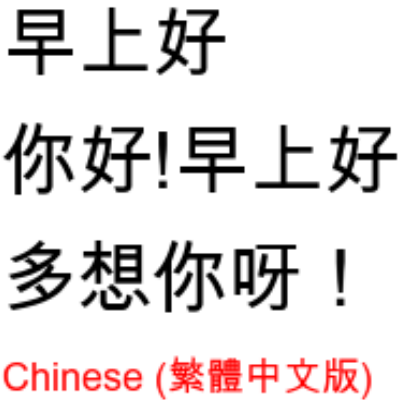

圖 3a：中文小字體

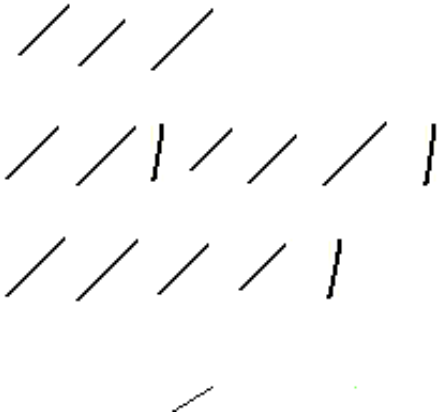

圖 4b：線狀描繪

選擇 CTRL+W 組合鍵顯示這些文字的線狀描繪，可以看到每個文字用兩種紋理映射而成。這裏字體比較小，所以用點陣圖來描繪比較有效。

當增加視窗尺寸而文字保持在線狀模式。文字將從紋理映射轉換成純色覆蓋模式：

圖 4c：大號中文字

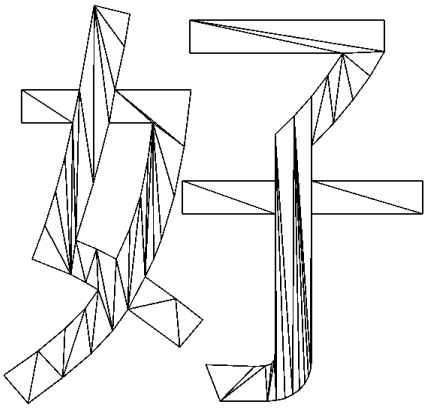

圖 4d：線狀描繪

大號字中，使用純色覆蓋更加有效。對大的點陣圖操作需要消耗大量記憶體。只有圖元點需要著色，在字元的空白處避免浪費處理器資源。圖 4c 和圖 4d 中，Scaleform 將字元通過一個固定的閾值，檢測其尺寸大小，並將其柵格化，當作幾何圖形來處理，而不作爲點陣圖來處理。

字體柔性陰影、模糊和其他效果都被支援。只要簡單的將對應的濾鏡作用于 Flash 文本域來產生預期的效果。更多細節請參考字體和文本設置概述，前面有對其的鏈結。例子可以從開發中心下載，文件爲 gfx_2.1_texteffects_sample.zip。


Isotropic shadow with offset
Text Shadow
文本陰影
Anisotropic glow, BlurX=2, BlurY=7
Text Glow 1
文本煥發 1
Isotropic glow, BlurX=BlurY=4, Knockout
TEXT GLOW 2
文本煥發 2
Rotating glow
Text Glow 3
文本煥發 3
Changing blurriness
Text Glow 4
文本煥發 4

圖 5：文本效果

6 C++、Flash 和動態腳本介面

Flash 的動態腳本語言支援交互動畫的創建。按鈕點擊、到達特定幀或者導入畫面等事件都能執行代碼來動態改變動畫內容、控制動畫流程，甚至啓動附加的動畫。動態腳本功能強大，以至於能夠在 Flash 中創建完整的小遊戲。與大多數編程語言類似，動態腳本支援變數和子函數。Scaleform 提供 C++介面直接操作動態腳本變數、陣列以及直接調用動態腳本副程式。Scaleform 還提供了調用回收機制，使得動態腳本能夠傳遞事件和資料給 C++程式。

6.1 *動態腳本到 C++*

Scaleform 提供了兩種機制，使得 C++應用程式能夠從動態腳本接收事件：FSCommand 和 ExternalInterface 函數。FSCommand 和 ExternalInterface 都在 GFx::Loader 中註冊一個 C++事件控制碼來接收事件資訊。FSCommand 事件由動態腳本的 fscommand 函數發出，接收兩個字串參數，無返回值。ExternalInterface 事件當動態腳本調用 flash.external.ExternalInterface.call 函數時被觸發，接收一個 GFx::Value 參數列表（在 6.1.2 中有描述），返回值給調用者。

由于在灵活性上的局限性，FSCommands已经不在推荐使用，由ExternalInterface替代。在这里仍然予以详细介绍，因为在一些遗留代码中还会遇到fscommands。而且，gfxexport可以产生一个所有在SWF文件中的fscommands使用报告，包括了–fstree， –fslist 和 –fsparams选项。这项功能用ExternalInterface是无法实现的，所以有些情况下fscommands仍然有其用途。

6.1.1 FSCommand 回收

動態腳本 fscommand 函數傳遞命令和資料給主應用程式。該函數在動態腳本的典型應用如下所示：

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

任何非字串參數傳遞給 fscommand 函數，如布林型或者整型，將被轉換成字串。而 ExternalInterface 可直接接收整型參數。

此處傳遞兩個字串給 Scaleform FSCommand 控制碼。一個應用程式通過 GFx::FSCommandHandler 子類註冊 FSCommand 控制碼同時註冊 GFx::Loader 或者單個 GFx::Movie 物件相關的類。若設置一個命令句柄到 GFx::Movie，則能夠接收到 fscommand 函數在動畫實例中的調用返回值。GFxPlayerTiny 實例展現了這個過程(搜索“FxPlayerFSCommandHandler”)，我們將添加類似的代碼到 ShadowVolume。本節詳細代碼步驟位於 6.1 小節。

首先，GFx::FSCommandHandler 子類：

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

Callback 方法接收傳遞給動態腳本中的 fscommand 的兩個字串參數，同時接收了指向調用 fscommand 的特殊動畫實例的指標。

接下來，在 GFxTutorial::InitGFx()中的 GFx::Loader 物件創建後註冊控制碼：

```
// 註冊 FSCommand 控制碼
Ptr<FSCommandHandler> pcommandHandler = *new OurFSCommandHandler;
gfxLoader->SetFSCommandHandler(pcommandHandler);
```

通過 GFx::Loader 註冊控制碼使得每個 GFx::Movie 繼承該控制碼。SetFSCommandHandler 可以在每個實例中被調用來跳過這些默認設置。

我們的自定義控制碼簡單的將每個 fscommand 事件列印到調試控制臺窗口。運行 ShadowVolume 並點擊“Toggle Fullscreen”按鈕。注意到只要用戶介面事件發生就會列印出相關資訊：

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

在 Flash Studio 中打開 d3d9guide.swf 文件，打開動態腳本面板(F9)。將螢幕上列印出的事件資訊與位於 ActionScript block for Symbol Definition(s) → hud → var : Frame 1
中的 fscommand 調用做對比：

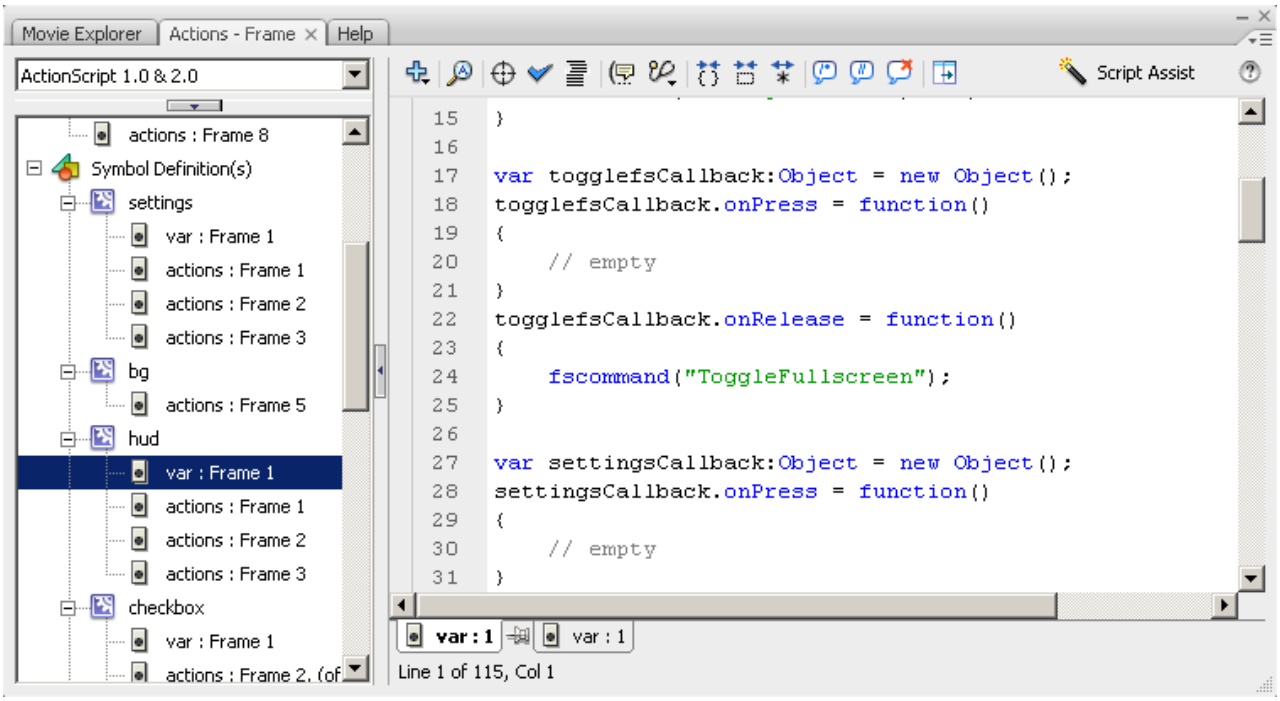

圖 6：動態腳本 fscommand()調用

作爲演示，將 fscommand(“ToggleFullscreen”) 改變爲 fscommand(“ToggleFullscreen”, this.UI._visible) 來列印 C++可視的 this.UI._visible 值。導出 flash 動畫(CTRL+ALT+Shift+S)並替換 d3d9guide.swf 文件。

當 FSCommand 事件發生時啓動相關代碼，用戶介面上的按鈕可集成到 ShadowVolume 實例中來。例如：增加以下代碼行，使 fscommand 控制碼進入全屏模式：

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

DXUT 函數 OnFrameMove 在一幀被描繪時調用。在 OnFrameMove 函數返回後增加如下相應代碼：

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

通常，事件控制碼不應該被阻塞，需要儘快地返回給調用函數。时间句柄通常只可以在 Advance 或 Invoke 调用函数中被使用。

6.1.2 ExternalInterface

Flash ExternalInterface 调用方法与 fscommand 类似，但更具有優越性，因爲它提供了更多靈活的參數並可返回值。

註冊一個 ExternalInterface 類與註冊一個 fscommand 控制碼類似：

```
class OurExternalInterfaceHandler : public ExternalInterface
{
  public:
      virtual void Callback(Movie* pmovieView,
                            const char* methodName,
                            const Value* args,
                            unsigned argCount)
      {
            Printf("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_Number:
                        GFxPrintf("%3.3f", args[i].GetNumber());
                        break;
                    case Value::VT_String:
                        GFxPrintf("%s", args[i].GetString());
                        break;
                    default:;
                        GFxPrintf("unknown");
                        break;
                  }
                  GFxPrintf("%s", (i == argCount - 1) ? "" : ", ");
            }
            GFxPrintf("\n");
      }
};
```

用 GFx::Loader 註冊控制碼：

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

當文本輸入框獲得或者失去焦點時，可以從動態腳本創建一個外部介面調用。選擇 F9 鍵查看 ActionScript for Symbol Definitions() → hud → var : Frame 1：

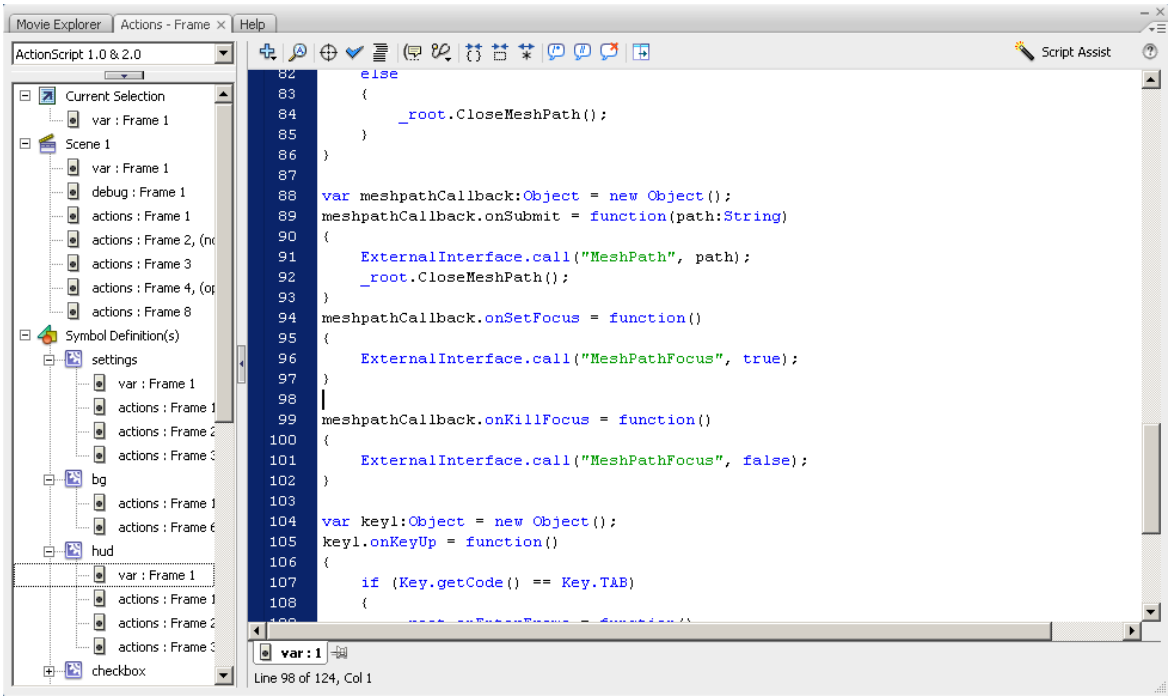

圖 7：MeshPathFocus 當文本輸入框獲得或者失去焦點時被調用。

外部介面調用將觸發外部介面控制碼並傳遞輸入文本框(true 或者 false)和專用命令字串 “MeshPathFocus” 的焦點狀態資訊。打開輸入文本框，點擊文本區，然後點擊螢幕空白處以從文本區移開焦點。控制臺將輸出如下資訊：

```
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
```

事件控制碼將檢測何時獲得焦點或者何時失去焦點並傳遞該資訊到 GFxTutorial 物件：

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

若文本框獲得焦點，GFxTutorial::ProcessEvent 將只傳遞鍵盤事件給動畫。當一個鍵盤事件傳遞給文本框時，將產生一個標記，防止傳遞到 3D 引擎：

```
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;
      }
}
```

空格(ASCII 碼爲 32)和 tab(ASCII 碼爲 9)對應於“Toggle UI” 和 “Settings” 按鈕，總是被傳遞。

爲了使用戶能夠改變描繪的網格，點擊“Change Mesh”按鈕來打開文本框，輸入新網格名，按下回車鍵。當回車鍵被按下時，文本框將調用動作腳本事件控制碼，其調用了用新名字命名的外部介面。調用 OurExternalInterfaceHandler::Callback 中的代碼如下所示：

```
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));
}
```

在全屏模式，實際工作在 DXUT 的 OnFrameMove 調用函數中完成。代碼建立在 DXUT 介面的事件控制碼基礎之上。

6.2 C++ *到動態腳本*

6.1 小節介紹了動態腳本如何被 C++調用。本節描述了使用 Scaleform 函數如何與其他單元通信，使得 C++程式初始化與播放動畫的通信。Scaleform 支援 C++函數讀取和設置動態腳本變數並調用動態腳本副程式。

6.2.1 操作動態腳本變數

Scaleform 支援 GetVariable 和 SetVariable 函數，通過這兩個函數可以直接操作動態腳本變數。6.2 小節中的對應代碼修改用來導入黃色 HUD 顯示(fxplayer.swf)，該顯示是 Scaleform Player 程式使用並在按下 F5 鍵時增加計數功能：

```
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 返回變數_root.counter 的值。起初該變數不存在，GetVariableDouble 返回零。計數器增加計數，新的值通過 SetVariable 保存到_root.counter 中。當前文檔關於 GFx::Movie 列表中包含了 GetVariable 和 SetVariable 參數的不同變數值。

fxplayer Flash 文件具有兩個動態文本區，能夠通過應用程式設置專用文本。MessageText 文本區位於螢幕中間，HUDText 變數位於螢幕的左上角。根據_root.counter 變數的值來產生一個字串，SetVariable 用來更新消息文本。

性能注釋：改變 Flash 動態文本區的首先方法爲設置 TextField.text 變數或者調用 GFx::Movie::Invoke 運行動態腳本程式來改變文本（更多資訊參考 6.2.2）。禁止將動態文本區與專用變數綁定後通過改變該變數來改變文本。儘管這樣做是可行的，但是有一個缺陷，因爲 Scaleform 在每個幀都要檢查變數的值。

圖 8：SetVariable 改變 HUD 文本值。

上面用法中將變數直接當成字串或者數位來處理。在線文檔中描述了使用新型 GFx::Value 物件句法有效地處理變數，直接將變數作爲整數處理，取消了字串到整數的轉換過程。

SetVariable 函數擁有一個可選使用的第三個參數，參數表示 GFx::Movie::SetVarType 類型，用來定義“粘貼”分配。這個參數在變數被分配但在時間軸上還未被創建時起到作用。例如：假設文本域 _root.mytextfield 直到動畫的第 3 幀才被創建。如果 SetVariable(“_root.mytextfield.text”, “testing”, SV_Normal)在動畫剛被創建的第 1 幀就被調用，則分配不起作用。如果調用由 SV_Sticky 產生（預設值），則請求排入佇列，直到第 3 幀中_root.mytextfield.text 的值有效時執行。通過 C++來初始化動畫將變得更加容易。

SetVariableArray 通過單個操作傳遞整個變數的陣列到 Flash。該功能可以用來動態移動下拉清單控制操作。以下代碼處於 GFxTutorial::InitGFx()函數中用來設置_root.SceneData 下拉清單的值。

```
// 初始化場景
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);
```

6.1 小節代碼包括了附加的外部介面控制碼，對應於亮度控制、光源數量、場景類型和其他呈現原始 ShadowVolume 介面的控制單元。

注釋：本例子中通過 C++來組裝下拉功能表作爲展示。通常，包含靜態選擇列表的下拉功能表應該通過動態腳本初始化而不是 C++代碼。

6.2.2 執行動態腳本副程式

除了改變動態腳本變數，動態腳本代碼也可以通過 GFx::Movie::Invoke 方法調用。這在一些複雜過程處理中非常有用，如觸發動畫、改變當前幀、可編程改變 UI 控制狀態和動態創建新按鈕或文本等用戶介面等。

本節使用 GFx::Movie::Invoke 來編程實現“Change Mesh”文本輸入框的打開，快捷鍵 F6 和 F7 分別用來打開和關閉該文本輸入框。由於當單選按鈕被選中時動畫才產生，所以使用 SetVariable 不能改變該狀態。SetVariable 也不能启动动画，但是通过 Invoke 调用 ActionScript 程序可以实现此功能。

GFx::Movie::Invoke 被調用來執行 WM_CHAR 鍵盤控制碼中的 openMeshPath 程式：

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

...
```

openMeshPath 的動態腳本代碼位於第 1 幀，確保動態腳本程式在第 1 幀畫面播放時被執行。使用鉤子時常會出現的錯誤爲調用動態腳本程式時無法獲得腳本，這種情況下錯誤資訊將被輸出到 Scaleform 腳本文件中。直到動態腳本程式關聯幀開始播放或者其關聯嵌套物件被導入，動態腳本程式才能改變其變數的值。當 GFx::Movie::Advance 初次調用時或者 GFx::MovieDef::CreateInstance 被調用且其參數 initFirstFrame 爲 true 時，位於第 1 幀中的所有動態腳本代碼都可以被獲取。

本例子用了鈎子的 printf 類型。其他版本的功能函數使用 GFx::Value 高效地處理非字串類型的參數。InvokeArgs 調用方法類似，除非其參數 va_list 使應用程式提供指向參數列表的指標。Invoke 和 InvokeArgs 之間的關係與 printf 和 vprintf 之間的關係類似。

6.3 多 Flash 文件之間通信

到現在爲止所討論的通信方法都與 C++有關。在一個大的應用程式中，用戶介面被分割成多個 SWF 文件，則需要編寫大量 C++代碼使得介面的不同元件之間互相通信。

例如，一個 MMOG 遊戲有一個總清單 HUD、分清單 HUD 和交易室。寶劍和金錢等條目可以從遊戲者的總清單中移交到交易室並交給另外一個遊戲者。但玩家穿上一件衣服，該條目需要從總清單 HUD 中移交到分清單 HUD。

這三個介面必須分爲三個獨立的 SWF 文件存儲並獨立導入以節省記憶體。但是，C++代碼在這三者的 GFx::Movie 物件間通信很快就會消耗衰竭。

還有更好的方法來解決這個問題。動態腳本支援 loadMovie 和 unloadMovie 方法，這類方法使得多個 SWF 文件在單個 GFx::Movie 中導入和導出交換。由於 SWF 動畫在相同的 GFx::Movie 中，它們可以共用變數空間，這樣無需 C++代碼在每次導入時初始化動畫。

在 MMOG 例子中，總清單可以描述成名爲_global.inventory 的動態腳本陣列。當其中一個 SWF 介面導入時，則根據該資料中資料來繪製條目。當其中一個 SWF 介面用動態腳本 unloadMovie 方法釋放時，_global.inventory 陣列仍然可被其他介面獲取。

下面举一例子，我们用 ActionScript 函数创建一个 container.swf 文件，导入和导出动画到共享变量空间。container.swf 文件不含美术资源，只包括几个 ActionScripts 脚本函数用来管理动画的导入和导出。首先创建一个 MovieClipLoader 对象如下所示：

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

这里用到的 ActionScript 函数以及其他更相信的情形，请参考 Flash 文档。动画既可以导入到一个命名空间对象，也可以导入到一个特定编号层。

```
////
// 导入文件到特定图层

function LoadFlashLevel(url, level)
{
    trace("LoadFlashLevel(" + url + ", " + level + ")\n");
    mclLoader.loadClip(url, level);
```

```
}

//
// 导入文件到一个名称对象

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

每个标识数字的“level”层可以包括单个动画。在不同图层中的动画可以共享数据和互相访问变量。图层按照动画剪辑的 Z 轴分布，使 UI 界面设计师可以选择哪个剪辑显示在前，哪个在后。不同图层中的动画能共享数据和互相访问变量。

通过 LoadMovieLevel 将动画导入到特定图层的好处为 Z 轴上根据图层数字隐含标注。动画中的变量可以通过_levelN.variableName 访问（例如，level6.counter）。

使用 LoadMovie 导入动画到特定名称的剪辑使复杂界面有更多的组织结构。动画可以为树形结构分布，变量可以根据地址排列（例如，root.inventoryWindow.widget1.counter）。

很多 Flash 动画剪辑基于 root 索引变量。如果动画导入到一个特定图层，root 指向该图层的起点。例如，将动画导入到图层 level 6，root.counter 和 level6.counter 指向相同变量。如果动画通过 LoadMovieLevel 导入到特定图层的起点，则该动画剪辑可以用 root 索引自身内部变量。

而用 LoadMovie 导入相同的动画到 root.myMovie 则不能正常工作，因为 root.counter 为应用树形根位置的计数变量。被组织成树形结构的动画应该设置为：lockroot = true。Lockroot 为一项 ActionScript 树形参量，可以将指向 root 的索引都指到子动画的 root 位置，而不是图层的 root 位置。更多关于 ActionScript 变量、图层和动画剪辑的信息请参考 Adobe Flash 文档。

无需考虑子动画是如何组织的，运行在相同的 GFx::Movie 中的动画剪辑可以互相范围变量并操作共享状态，大大简化了复杂界面的创建。

Container.fla 也包含了相应函数用来卸载不需要的动画剪辑以减少内存消耗（例如，一旦用户关闭一个窗口，该窗口资源就可以被释放）。

当 LoadMovie 或 LoadMovieLevel 函数已经返回，但动画尚为完成必要的导入。则 loadClip 函数只对导入动画部分进行初始化，其余工作由后台程序进行。如果你的应用程序必须知道何时动画能够完全导入

（例如，程序初始化状态）。可以使用 MovieClipLoader 的监听器函数。Container.fla 中包含了一个函数执行的符号，可以扩展为处理 ActionScript 中的事件或者作为 ExternalInterface 调用使能 C++应用程序来执行动作。

```
// 定义回调函数报告事件：
//   1.开始导入动画
//   2.导入动画失败
//   3.导入动画结束
//   4.正在导入
//
// 这些回调函数必不可少，
// 因为动画在 LoadMovie()函数返回时未必能导入完毕。
//
// 当前回调函数只打印调试信息。
// 一个应用程序需要执行这些事件
// 应该使用 ExternalInterface 调用告知 C++应用程序
// 或者在 ActionScript 中处理事件。

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);
```

Tutorial\section7.3 中的代码在初始化时只导入 container.swf 文件替代 d3d9guide.swf 和 fxplayer.swf 文件。按 F8 根据需求导入 HUD，按 F9 导入用户主 UI 界面：

```
void GFxTutorial::ProcessEvent(HWND hWnd, unsigned uMsg,
                               WPARAM wParam, LPARAM lParam,
                               bool *pbNoFurtherProcessing)
{
        ...

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

以上代碼將 HUD 導入到圖層 5，將主介面導入到圖層 6。Flash 動態腳本所在的圖層與場景的 z 軸相關聯。在相同 GFx::Movie 中，上層圖層中的畫面將覆蓋下層圖層中的畫面。動畫最初導入 Container.swf 文件時默認放在圖層 0。

在不同圖層中的變數可以用_level 關鍵字來訪問。比如，位於圖層_level6 中的主介面可以直接訪問 HUD 的文本域，只需改變_level5.MessageText.text 的參數即可。同樣的這兩個位於不同圖層的畫面也可以通過關鍵字_global 變數來訪問 Flash 文件中的全部變數空間。例如，每個畫面都可以訪問_global.counter

總體變數。這樣有一個好處就是當該兩個畫面都釋放後，在_global 全局命名空間中的變數不會丟失。不管畫面的導入還是導出均可以訪問固定不變的_global 全局命名空間。更多關於_level, _global 變數和動態腳本變數的命名空間相關資訊，請參考 Adobe Flash 相關文檔。

運行應用程式按 F8 導入 HUD。然後按 F9 導入主介面。注意到當導入主介面時會有一定的延時。這是因爲導入中文字體需要耗費一定的時間。可以用一定的方法來加速該過程，將 SWF 文件預先編譯爲 Scaleform 文件（參考第 7 章），設置 Scaleform 多線程導入功能，使得在背景畫面中就可以導入共用字體文件（參閱開發中心網站的字體和文本相關文檔）。

按 F8 提出 HUD，然後按 F5 使能計數器計數，在 6.2 節中添加了計數器並停止了工作。SetVariable 方法應用了_root.counter，但是現在 HUD 導入到了_level5 命名空間，所以計數部分代碼需要做如下修改：

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

      ...
```

_level5.counter 也可以被 d3d9guide.swf 文件中位於圖層 6 中的動態腳本直接訪問，使得介面之間不通過 C++代碼就可以直接通信。

運行應用程式，按 F8 鍵導入 HUD，按 F5 開啓計數器。按 F2 釋放 HUD，然後按 F8 重新導入 HUD 繼續用 F5 來啓動計數。當 HUD 被釋放時候，計數值丟失，繼續從零開始計數。這是因爲計數變數位於_level5 當中。將變數從_level5.counter 改變爲_global.counter 然後再作前面的操作。將變數保存到_global 中後，無論是動畫導入還是導出變數值都將保存。

7 渲染到紋理

渲染到紋理是 3D 渲染中一項通用的先進技術。此技術使使用者可以渲染到紋理表面,而非背景緩衝區。然後,就可以把結果產生的紋理用作常規紋理,並可根據需要將常規紋理應用到場景幾何。例如,此技術可用來創建“遊戲內看板”。首先,在 Flash Studio 中將您的看板撰寫為 SWF 檔,將該 SWF 檔渲染到一個紋理,然後將該紋理應用到適當場景幾何。

在此教程中,我們只需將 UI 從上一個教程渲染到後牆。如果您按滑鼠中鍵到處移動光源,您就會看到相應的 UI 變化情況。

現在我們將循序漸進地演練此示例的內部運作情況。

1. 在所有以前的教程中,我們載入了 cell.x 場景檔。此檔包含顯示房間地板、牆壁和屋頂所需的頂點座標、三角指數和紋理資訊。如果您看看此檔,就會注意到同一牆壁紋理 (cellwall.jpg) 用於全部四面牆壁。我們想要只在後牆上渲染我們的 UI。因此,我們創建一個單獨的紋理,用於稱為 cellwallRT.jpg 的後牆。接下來,修改 MeshMaterialList 陣列以引用後牆的紋理。

```
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. 以前的教程中已經提到,cell.x 檔的解析是由 DXUT 框架在內部進行的。在此解析過程中,DXUT 初始化頂點和索引緩衝區,並創建在資料清單中引用的紋理。這些紋理是使用 CreateTexture 調用創建的;不過,傳遞到此函數的用法參數不適用於渲染紋理 (Render Texture)。有關此話題的更多資訊,請參閱 DXSDK 文檔。要創建一個可用作渲染目標的紋理,請使用恰當的用法參數重新創建渲染紋理,並將其附加到背景網路。同時釋放原來由 DXUT 創建的紋理,以避免記憶體洩漏。

```
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. 接下來修改 AdvanceAndRender 函數以便將 GFx 渲染到我們的紋理,而不是渲染到背景緩衝區上。此過程的關鍵步驟如下：
    a. 首先保存原始背景緩衝區表面,這樣一旦我們完成渲染就可以恢復。
    ```
    pd3dDevice->GetRenderTarget(0, &poldSurface);
    pd3dDevice->GetDepthStencilSurface(&poldDepthSurface);
    ```
    b. 接下來,從我們的紋理獲取渲染表面,並將其設置為渲染目標。
    ```
    ptex->GetSurfaceLevel(0, &psurface);
    HRESULThr = pd3dDevice->SetRenderTarget(0, psurface );
    ```
    c. 將電影視口調整到我們的紋理的尺寸,調用 Display,然後恢復到步驟 a 中保存的原始渲染表面。

8 GFxExport 預處理

直到現在我們才開始直接導入 SWF 文件。這個開發流程是爲了使設計者在開發 SWF 文件的過程中能夠交換新 SWF 文件內容，無需用到遊戲執行代碼便可看到遊戲的運行結果。但是，在發佈版本中包含 SWF 文件不被推薦，因爲導入 SWF 文件需要預先處理影響導入時間。

GFxExport 是處理 SWF 文件轉換爲導入流優化格式的有用工具。在預處理的過程中，圖像被萃取並放置到不同的文件中以便由遊戲資源引擎統一管理。圖像能夠轉換成 DDS 格式，採用 DXT 紋理壓縮優化導入過程、減少即時運行時的記憶體消耗。內嵌字體被多重壓縮，字體紋理可以任意選擇以適應點陣圖文字的使用。GFxExport 輸出可以有多種壓縮方法。

使用 Scaleform 文件開發是件非常容易的事情。GFxExport 工具支援廣泛的配置項，在線幫助文檔中就有相關描述。將 d3d9guide.swf 和 fxplayer.swf 文件轉換成 Scaleform 格式：

```
gfxexport -i DDS -c d3d9guide.swf
gfxexport -i DDS -c fxplayer.swf
```

gfxexport.exe 可執行文件位於目錄 C:\Program Files\Scaleform\$(GFXSDK) 。這些命令同時包含在 convert.bat 批次檔案中，具體步驟查看第 7 節。

-i 選項表示圖像格式，這裏參數爲 DDS。DDS 參數使在 DirectX 平臺上發揮優勢，因爲它支援 DXT 紋理壓縮，DXT 紋理壓縮通常在即時運行時能節約四分子三的記憶體。

-c 選項使能壓縮。只有在 Scaleform 文件中的向量和動態腳本計數器才被壓縮。圖像的壓縮格式取決於圖像的輸出格式和 DXT 壓縮選項。

-share_images 選項通過特徵碼識別不同文件中的相同圖像，且導入時只需導入一份共用副本。

ShadowVolume 應用程式的版本在第 8 節步驟中做相關了修改，使得導入 Scaleform 文件變得簡單而容易，只需要修改 GFx::Loader::CreateMovie 的檔案名參數即可。

9 下一步

本指南對 Scaleform 的功能做了基本的介紹。需要探討的其他主題還包括以下方面：

- Flash 遊戲中的渲染與紋理。參考 Scaleform Player SWF 與紋理 SDK 的例子，Gamebryo 集成演示和虛擬引擎 3 集成演示。
- 開發中心的 FAQs 中涉及的其他因素
- Scale9 視窗支援文檔位於 Scale9Grid Overview 概述相關文檔頁中
- 自定義導入：除了從 Scaleform 或者 SWF 導入文件，Flash 內容還可以直接從記憶體或者遊戲資源管理器導入，只需要使用 GFx::FileOpener 的子集。
- 通過 GFx::ImageCreator 可自定義圖像和紋理。
- 使用 GFx::Translator 進行語言文字的本地化