Autodesk®
Scaleform®

Scaleform 集成应用指南

本文档通过 DirectX 9 实例介绍了基本的 Scaleform 使用方法和 3D 引擎的集成应用。

作者： Ben Mowery
版本： 3.04
最近更新： 2011 年 8 月 4 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Autodesk Scaleform 联系方式：

文档	Scaleform 4.1 集成应用指南
地址	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
网站	www.scaleform.com
邮箱	info@scaleform.com
电话	(301) 446-3200
传真	(301) 446-3199

目录

1 引言 1

2 文档概述 2

3 组建安装和关联编译 3

3.1 安装 3

3.2 编译演示文件 4

3.3 在 Scaleform Player 中 SWF 回放定位 4

3.4 编译基本实例 6

3.5 Scaleform 编译关联项 7

3.5.1 AS3_Global.h and Obj\AS3_Obj_Global.xxx 文件 8

4 游戏引擎集成 10

4.1 Flash 表现手法 10

4.2 缩放模式 18

4.3 输入事件处理 19

4.3.1 鼠标事件 19

4.3.2 键盘事件 20

4.3.3 点击测试 22

4.3.4 键盘焦点 23

5 文字本地化和字体介绍 24

5.1 字体概述和容量 24

6 C++、Flash 和动态脚本接口 27

6.1 动态脚本到 C++ 27

6.1.1 FSCommand 回收 27

6.1.2 ExternalInterface 30

6.2 C++ 到动态脚本 32

6.2.1 操作动态脚本变量 32

6.2.2 执行动态脚本子程序 35

6.3 多 Flash 文件之间通信 36

7 渲染到纹理 42

8 GFxExport 预处理 44
9 下一步 45

1 引言

Scaleform 作为一个可视化 UI 设计的中间件解决方案，其高性能已被充分证明，有了它，开发者通过 Adobe Flash Studio 可以高效率、低成本地创建富有现代感的 GPU 加速动画用户界面和矢量图形，而无需学习新工具的使用或者动画处理方法。Scaleform 为直接在 Flash Studio 创作游戏用户界面开辟了一条可视化开发途径，能够无缝衔接工作中的各个环节。

除了普通的 UI 界面显示，开发者还可以利用用 Scaleform 在 3D 场景中显示 Flash 动画，将 Flash 画面映射到 3D 表面。同样，3D 对象和视频也可以在 Flash UI 当中显示。由此，Scaleform 即可以作为单独的 UI 设计工具，也可以用来增强游戏前端界面设计功能。

本使用指南涵盖了 Scaleform 的安装和使用等相关内容。其中提供了一个基于 Flash 用户界面的增强型 DirectX ShadowVolume SDK 实例作为详细描述。

注释：Scaleform 已被大多数主流游戏引擎所集成。支持 Scaleform 的游戏引擎中只需少量的代码便可以使用。本指南主要目的是告知那些需要在各自的游戏引擎中集成 Scaleform 的工程师，如何去实现集成，以及让那些需要深入了解 Scaleform 性能的人员了解更多技术相关细节内容。

注释：请确保参考本指南时，使用最新版本的 Scaleform。本指南适用于 Scaleform 4.0 或者更高版本。

注释：本用用指南组件可能与一些较早的显卡不兼容。这是由于应用程序中所用的方法依赖于 DirectX SDK ShadowVolume 代码与 Scaleform 不兼容。当使用指南中的方法时是否会出现“cannot create renderer”错误提示，可以通过检查 Scaleform Player 应用程序是否能顺利执行来作判断。

2 文档概述

最新的多语种 Scaleform 文档可以在 Scaleform 开发者中心下载到，网址为：https://gameware.autodesk.com/scaleform/developer/。免费注册后即可登录站点。

当前文档包括：

- Web 格式 Scaleform SDK 参考文档。
- PDF 文档。
- 字体概述：描述字体和文本渲染系统，并详细介绍了艺术资源和 Scaleform C++ API 接口的配置，以实现适应国际化使用。
- XML 概述：描述了 Scaleform 中支持的 XML 元素。
- Scale9 Grid：解释了如何利用 Scale9Grid 功能函数创建可变视窗、面板和按钮。
- IME 配置：描述了如何在终端应用中集成 Scaleform IME，以及如何如何在 Flash 创建和设置 IME 文本输入窗口。
- 动态扩展脚本：覆盖了 Scaleform 动态脚本扩展 。

3 组建安装和关联编译

3.1 *安装*

下载最新 Windows 平台的 Scaleform 安装包。运行两个安装程序，保持其默认安装路径和选项。如有需要，Scaleform installer 将更新 DirectX。Scaleform 默认安装目录为 C:\Program Files\Scaleform\GFx SDK 4.1。

目录结构如下所示：

3rdParty

Scaleform 需要的扩展库文件如 libjpeg 和 zlib。

Projects

包含了建立上述应用的 Visual Studio 工程文件。

App\Samples

源代码实例和其他演示样板

Apps\Tutorial

此教程的源代码及项目。

Bin

包含了已编译的演示二进制文档和 Flash 实例：

FxPlayer:	简单的 Flash 文本 HUD。
IME:	自定义 IME 文字输入法界面实例的 Flash 文件。
Samples:	各种 Flash 实例的 FLA 原文件，包括用户界面元素如按钮、编辑框、键区、菜单和旋转计数等。
Video Demo：	Scaleform 视频示例和文件
Win32:	Scaleform Player 及其演示文件的预编译二进制文件。
AMP:	用于运行 AMP 工具的 Flash 文档。
GFxExport:	GFxExport 是一个预编译工具，具有加速导入 Flash 内容的功能，详细情况见第 7 小节

Include

Scaleform 便利头文。

Lib

Scaleform 库文件。

Resources

CLIK 组件和工具

Doc

第 2 节所描述的 PDF 文档。

3.2 编译演示文件

为了验证系统是否配置完成，能够编译 Scaleform。首先编译如下演示文件，在 C:\Program Files (x86)\Scaleform\GFx SDK 4.1\Projects\Win32\Msvc90\Demos\GFx 4.1 Demos.sln 目录中找到 Demos.sln 文件，在 Visual Studio 中打开.sln 文件，选择 D3D9_Debug_Static 设置属性并编译该 Scaleform Player 工程。

检查位于目录 GFx SDK 4.1\Bin\Win32\Msvc90\GFxPlayer\GFxPlayer_D3D9_Debug_Static.exe 文件的更新时间，确认已成功编译并运行该程序。

此程序和其他位于示例文件目录 Start → All Programs → Scaleform → GFx SDK 4.1 → Demo 的 GFx Player D3Dx 应用程序都为硬件加速 SWF 播放器。开发过程中，能用该工具来测试和定位 Scaleform Flash 回放功能。

3.3 在 Scaleform Player 中 SWF 回放定位

运行 Start → All Programs → Scaleform → GFx SDK 4.1 → GFx Players→ GFx Player D3D9 程序。打开 C:\Program Files\Scaleform\GFx SDK 4.1\Bin\Data\AS2\Samples\SWFToTexture 目录，拖放 3DWindow.swf 到应用程序中。

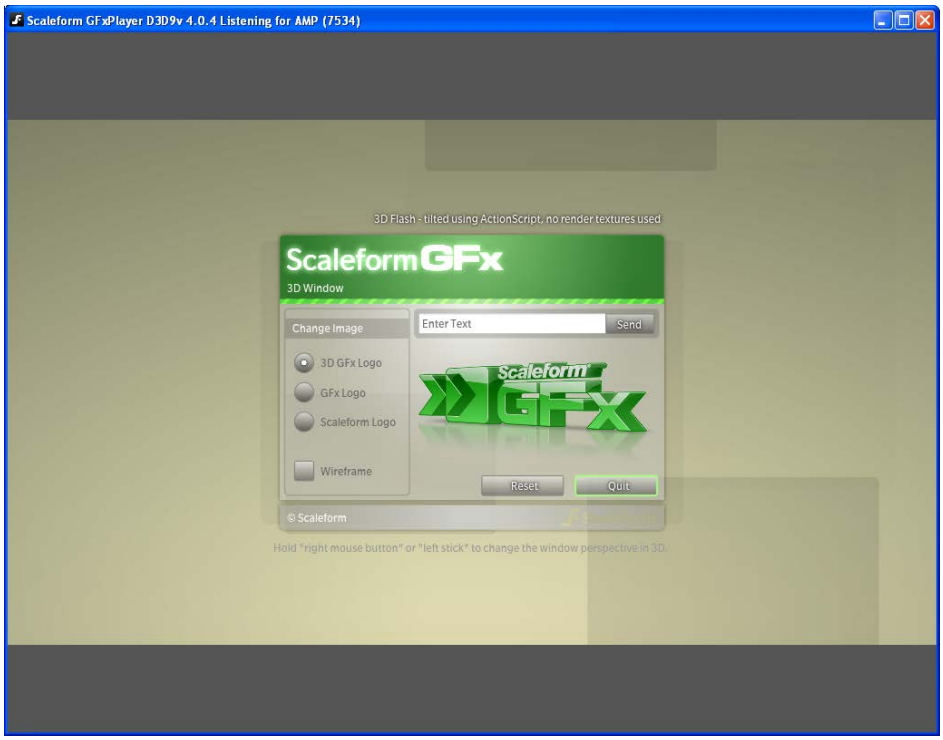

图 1： Flash 实例的硬件加速回放

按 F1 快捷键显示帮助窗口。尝试如下选项：

1. 选择 CTRL+F 组合键查看性能指标。当前 FPS 值将会出现在标题栏中。
2. 选择 CTRL+W 组合键进入线性帧模式。注意 Scaleform 中如何通过将 Flash 内容转换为三角元素来优化硬件回放功能。再一次选中 CTRL+W，则离开线性帧模式。
3. 放大图像操作，按下 CTRL 键并点击鼠标左键，然后将鼠标上下移动即可。
4. 图像变形操作，按下 CTRL 键并点击鼠标右键，然后移动鼠标即可。
5. 选择 CTRL+Z 组合键返回到普通视图。
6. 选择 CTRL+U 功能键进入全屏回放模式。
7. 选择 F2 快捷键分析动画参数，包括内存使用情况等。

注意到如何锐化按钮边角等边缘曲线部分，使得在近距离观看时仍显得曲线平整。当视窗放大或缩小时，Flash 也随着放大和缩小。矢量图的优势之一是可以放大缩小到任何比例以适应任何分辨率。而位图通常需要在 800x600 和 1600x1200 不同分辨率下准备两张不同尺寸的图像。

Scaleform 同时也支持传统的位图显示，如屏幕右侧中间的“Scaleform”图标。任何设计师能够在 Flash 中创作的绝大多数效果，在 Scaleform 中都能够描绘出来。

放大“3D Scaleform Logo”单选按钮，选择 CTRL+W 组合键切换到线性帧模式。注意到圆弧形被栅格化成三角形状。重复多次选择 CTRL+A 组合键进入反锯齿模式。在边缘反锯齿（EdgeAA）模式，三角

像素点阵填充原型边缘来实现反锯齿效果。这就是全屏反锯齿（FSAA）视频卡所带来的超强性能。该操作需要四倍的视频帧缓存区和四倍的像素渲染运算消耗来实现 AA 效果。Scaleform 的 EdgeAA 专利技术利用了矢量对象表示法作为反锯齿应用，该效果得益于那些绝大多数典型的弯曲边缘和大尺寸文本显示区域。尽管，三角计数将增加，性能影响是可控的，因为图形元（DP）的数量仍然是个常数。显然，使用视频卡的反锯齿功能和 EdgeAA 功能需要更多消耗，为了高效运作，可以将这些属性设置禁止使用或者适当调整。

Scaleform Player 工具用来调试 Flash 文件并测试性能指标。打开任务管理器查看 CPU 使用率。CPU 使用率随着 Scaleform 渲染帧的数量的增多而增加。选中 CTRL+Y 组合键查看显示刷新的帧速率，典型情况下为 60 帧每秒。需要注意降低 CPU 使用率至关重要。

注释：当测试您的应用工程时，务必运行发布（release ）编译模式而不要运行调试（debug）编译模式的 Scaleform，由于调试（debug）编译模式的 Scaleform 不提供性能优化功能。

3.4 编译基本实例

如果使用 Scaleform 4.1 的话，把 Scaleform\GFx SDK 4.1\Apps\Tutorial 中的 SLN 项目文件打开。可以在 Scaleform→ GFx SDK 4.1→ 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。确保工程文件配置设置为“Debug”模式并运行应用程序。

图 2：默认用户界面下的 ShadowVolume 应用画面

3.5 Scaleform 编译关联项

但新建一个 Scaleform 工程时，在编译之前 Visual Studio 中有些必须设置的选项。本 Tutorial 应用程序已经在适当的位置有相关的参考索引，只需按照其步骤执行。请牢记$(GFXSDK)是基本 SDK 安装目录的环境变量。默认路径为 C:\Program Files\Scaleform\GFx SDK 4.2\；如果你的库文件在不同的路径，需要改$(GFXSDK)环境变量指向系统中库文件所在的路径。在这些例子中我们将使用“Msvc90”；如果你使用 Visual Studio 2008 或者 Visual Studio 2010，则需要对应的使用 Msvc90 或 Msvc10 参数。

将 Scaleform 添加到工程所在的目录，包括调试模式和发布模式的目录下：

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

将下列文字粘贴到 Visual Studio 的“Additional Include Directories”区域

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。

将以下库文件所在的目录路径添加到调试编译配置中的路径搜索中：

$(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

将下列文本粘贴到“Additional Library Directories”区域：

"$(DXSDK_DIR)\Lib\x86";
"$(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"

注释：改变 Msvc90 参数以符合所用的 Visual Studio 版本。

相应的将发布版本的库文件添加到搜索路径中，以便发布和编译信息配置：

$(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

将下列文本粘贴到“Additional Library Directories”区域（发布和配置信息）：

"$(DXSDK_DIR)\Lib\x86";
"$(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”

注释：改变 Msvc90 参数以符合所用的 Visual Studio 版本。

最后，添加 Scaleform 库和相关文件：

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

开发者必须注意：AS3_Global.h 与 Obj\AS3_Obj_global.xxx 是**完全无关的**文件。

AS3_Obj_Global.xxx 文件包含对所谓“全局”ActionScript 3 对象的实现。每个 swf 文件均包含至少一个称为“脚本”(script) 的对象，这是一个全局对象。还有一个类 GlobalObjectCPP，它是用 C++ 实现的所有类的一个全局对象。这是 Scaleform VM 实现所特有的。

AS3_Global.h 有一个完全不同的用途。此文件包含 ClassRegistrationTable 阵列。此阵列的用途是引用实施相应 AS3 类的 C++ 类。没有此引用，代码就会被一个链接程序排除在外。因此，必须在可执行程序中定义 ClassRegistrationTable，否则就会收到一个链接程序错误。为此，我们的每个演示播放器均包含了 AS3_Global.h。

将 ClassRegistrationTable 置入一个 include 文件并要求开发者将其包含在内的全部目的就是允许自定义 ClassRegistrationTable（出于有可能减小代码大小的目的）。达此目的的最佳方法是制作 AS3_Global.h

的一个副本，注释掉不需要的类（然后，这些类就不会被链接进去），并把自定义的版本包含在您的应用程序中。

不过，有一个与此优化相关的陷阱。由于 AS3 VM 中的名称解析在运行时发生，因而就有可能找不到从表中解释掉的需要的类。因此，如果想要消除“不必要的”类，请确保您的应用程序在此操作之后仍然能够正常工作。

4 游戏引擎集成

Scaleform 提供了集成层，支持为大多数主流 3D 游戏引擎：包括 Unreal® Engine 3, Gamebryo™, Bigworld®, Hero Engine™, Touchdown Jupiter Engine™, CryENGINE™, 和 Trinigy Vision™ 引擎。这些游戏在开发中将 Scaleform 集成到其引擎只需少量或者无需编写代码。

本节描述了如何将 Scaleform 集成到自定义 DirectX 应用当中。DirectX ShadowVolume SDK 实例是一个标准的 DirectX 应用程序，拥有典型游戏所具有的应用过程和游戏环节。如前一节所述，该应用程序采用基于 DXUT 的 2D 界面覆盖来渲染 3D 场景。

本使用指南覆盖了 Scaleform 集成到应用程序中用基于 Flash 的 Scaleform 界面代替默认 DXUT 用户界面的整个过程。

DXUT 框架不显露应用程序的隐含渲染环，而显露了底层抽象层的细节调用过程。需理解这些与标准 Win32 DirectX 渲染环相关的步骤，则对比 Scaleform SDK 附带的 GFxPlayerTiny.cpp 实例源文件。

4.1 Flash 表现手法

集成过程的第一步为将 Flash 动画在 3D 背景中表现出来。这包含了实例中由一个 GFx::Loader 对象管理应用工程中 Flash 内容的全局调用，GFx::MovieDef 中包含了 Flash 内容，GFx::Movie 对象显示了动画的一个播放实例。另外，Render::Renderer2D 对象和 Render::HAL 对象将实例化，作为本实例 DirectX 中 Scaleform 和专用渲染 API 函数之间的接口。我们还讨论了如何根据释放的设备事件有效分配资源，如何处理全屏/视窗转换。

本节 ShadowVolume 实例版本修改步骤在 4.1 小节中有描述。文档中所示的以下相关代码仅供参考，并不全面。

步骤 #1: 添加头文件

为 ShadowVolume.cpp 添加必要的头文件。

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

某些 Scaleform 对象在视频渲染中需要用到，被封装为一个新的类添加到 GFxTutorial 应用程序中去。为使代码保持简介，保持 Scaleform 状态在统一在一个类中具有一定优势，只需一

个删除调用就可以释放所有的 Scaleform 对象。在以下步骤中将会对这些对象之间的相互作用做详细介绍。

```
//每个应用程序包含一个 GFx::Loader
Loader           gfxLoader;

//每个 SWF/GFx 文件包含一个 GFx::MovieDef
Ptr<MovieDef>   pUIMovieDef;

//每个视频播放实例包含一个 GFx::Movie
Ptr<Movie>  pUIMovie;

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

步骤#2：初始化 GFx::System

Scaleform 初始化的第一步为初始化 GFx::System 对象来分配 Scaleform 内存。在 WinMain 中我们添加如下代码行：

```
//每个应用程序包含一个 GFx::System 对象
GFx::System          gfxInit;
```

GFx::System 对象必须在第一个 Scaleform 调用时被获取，在 Scaleform 结束前不能被释放，这就是其位于 WinMain 函数头的原因。GFx::System 作为初始化，这里使用 Scaleform 的默认存储空间，但是可以被应用程序的自定义存储空间所覆盖。本 Tutorial 的目的为简化 GFx::System 对象并未对其深入操作。

GFx::System 在应用程序结束时必须被释放，意味着其不能作为一个全局变量。当 GFxTutorial 对象被释放时 GFx::System 对象也立即销毁。

根据特别应用所建立的体系结构，相对于创建一个 GFx::System 对象实例，调用 GFx::System::Init() 和 GFx::System::Destroy() 静态函数来的更加简便。

步骤#3: 导入和创建渲染器

其余的 Scaleform 初始化在应用工程的 WinMain 函数在其自身 InitApp() 初始化完成之后执行。在调用 InitApp()函数之后添加以下代码：

```
gfx = new GFxTutorial();
assert(gfx != NULL);
```

```
if(!gfx->InitGFx())
      assert(0);
```

GFxTutorial 包含了一个 GFx::Loader 对象。一个应用程序通常只有一个 GFx::Loader 对象，该对象用来导入 SWF/GFx 内容以及在资源库中存储 SWF/GFx 内容，以便资源的重用。相互独立的 SWF/GFx 文件可以共享图像和字体等资源以节省内存空间。GFx::Loader 还包含了状态配置集，如 GFx::Log，作为调试脚本。

GFxTutorial::InitGFx()函数的第一步为设置 GFx::Loader 对象状态。GFx::Loader 对象传递调试跟踪信息给 SetLog 中提供的句柄。调试输出信息包含了 Scaleform 功能函数执行出错信息，输出到脚本当中，这些信息起到很好的作用。本例中我们采用 Scaleform PlayerLog 句柄的默认方式，其将消息直接输出到控制台窗口，但是，GFx::Log 子集同时可完成与游戏引擎调试脚本信息的整合。

```
// 初始化脚本—Scaleform输出错误信息到脚本文件
// 数据流
gfxLoader->SetLog(Ptr<Log>(*new GFxPlayerLog()));
```

GFx::Loader 通过 GFx::FileOpener 类来读取文件内容。默认方式下从光盘文件中读取数据，同时GFx::FileOpener子集支持从内存或者其他资源文件读取数据的自定义方式。

```
// 导入默认文件
Ptr<FileOpener> pfileOpener = *new FileOpener;
gfxLoader->SetFileOpener(pfileOpener);
```

Render::HAL 是一个通用接口，使Scaleform输出图形到各种硬件设备。我们创建一个D3D9渲染器实例并与导入器相关联。渲染对象负责管理D3D设备、纹理和顶点缓存，这些Scaleform将会用到，在HAL::InitHAL模式下，我们需要为Render HAL对象传递一个IDirect3Device9指针，这个指针由游戏初始化，通过它Scaleform能够创建DX9资源并顺利渲染UI元素。

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

上述代码用到了 Scaleform 的 Render::HAL 子集中提供的 Render::D3D9::HAL 对象，更好的控制 Scaleform 渲染行为和紧密集成。

步骤#4：导入 Flash 动画

接下来 GFx::Loader 已经可以导入动画了。被导入的动画表示为 GFx::MovieDef 对象。这些 GFx::MovieDef 对象包含了动画的所有共享数据，如几何图形和纹理。但不包含单个实例的信息，如单个按钮的状态、动态脚本变量、当前动画帧等信息。

```
// 导入动画
pUIMovieDef = *gfxLoader.CreateMovie(UIMOVIE_FILENAME,
                                     Loader::LoadKeepBindData |
                                     Loader::LoadWaitFrame1, 0);
```

LoadKeepBindData变量指向了系统内存中的纹理图像内容副本，在应用中需要重新创建D3D设备时将会用到。该变量在游戏控制系统和纹理已知且不会丢失的情况下不是必须元素。

LoadWaitFrame1指令通知CreateMovie函数直到第一帧导入后才返回值。这在使用GFx::ThreadedTaskManager时非常重要。
最后一个参数是可选的，并说明了所用的内存区。有关创建和使用内存区的信息，请参阅 Memory System Overview 。

步骤#5：动画实例创建

在描绘动画前，必须从 GFx::MovieDef 对象创建 GFx::Movie 实例。GFx::Movie 包含了单个运行实例的相关状态信息，如当前帧、动画时间、按钮状态和动态脚本变量等。

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

传递给 CreateInstance 函数的参数为第一帧是否导入状态。如果该状态为 false，则能够 在位于第一帧的动态脚本执行前改变 Flash 和动态脚本的状态。最后一个参数是可选的，并说明了所用的内存区。有关创建和使用内存区的信息，请参阅 Memory System Overview 。

动画实例一旦创建，第一帧调用 Advance()函数初始化，只有当传递 false 给 CreateInstance 函数时才需要这个步骤。

```
// 指向动画第一帧
pUIMovie->Advance(0.0f, 0, true);

// 记录当前时间，测算当前帧到下一帧的时间消耗
MovieLastTime = timeGetTime();
```

传递给 Advance 函数的第一个参数为时差，第二个参数为当前帧到下一帧的时间间隔。记录当前系统时间用来计算当前帧和下一帧的时差。

为了在 3D 场景中表现动画透明效果：

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

没有上述函数调用，动画在渲染时将会用 Flash 文件中已定义的背景色覆盖 3D 场景。

步骤#6：设备初始化

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。当窗口大小改变或者全屏/窗口切换使 D3D 设备句柄发生变化时需要重新调用 InitHAL 函数。

ShadowVolume 中的 OnResetDevice 功能函数在设备创建并初始化或者设备复位时由 DXUT 框架调用。以下代码对应于 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 函数参数表示 Scaleform 不会调用 DirectX 中的 BeginScene() 和 EndScene()函数。这个设置是必不可少的，因为 ShadowVolume 实例应用程序中在 OnFrameRender 函数中已经调用了该两个函数。

步骤#7：设备丢失

当窗口大小改变或应用程序切换为全屏模式时，D3D 设备将丢失。所有 D3D 界面信息包或顶点缓存和纹理必须重新初始化。ShadowVolume 在 OnLostDevice 函数回收中释放界面信息。Render::HAL 获得设备丢失信息后将用 GFxTutorial 中的 OnLostDevice 方法释放其 D3D 资源。

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

本步骤和前面一个步骤解释了 DXUT 框架回收系统中初始化和设备丢失的相关操作。需要一个基本的 Win32/DirectX 渲染环例子，请查看 Scaleform SDK 中的 GFxPlayerTiny.cpp 实例源代码。

步骤#8：资源分配和回收

由于所有 Scaleform 对象都包含在 GFxTutorial 对象当中，这样资源回收操作变得很简单，只需在 WinMain 函数结尾处删除整个 GFxTutorial 对象即可：

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

另外一个需要考虑的问题是如顶点缓存等 DirectX 9 资源回收。这在 Scaleform 中得到很好的处理，但在游戏主循环中发生资源分配和回收需要了解 InitHAL() 和 ShutdownHAL()函数所起的作用。

在 DirectX 9 实现中，InitHAL 分配 D3DPOOL_DEFAULT 资源，包括顶点缓存。当与您自己的引擎集成时，设法把 InitHAL 放置到合适的位置来分配 D3DPOOL_DEFAULT 资源。

ShutdownHAL 将释放 D3DPOOL_DEFAULT 资源。应用程序使用 DXUT 框架，包括 ShadowVolume，应当在 DXUT 中的 OnResetDevice 调用中分配 D3DPOOL_DEFAULT 资源，而在 OnLostDevice 调用中释放资源。GFxTutorial::OnLostDevice 方法中的 ShutdownHAL 调用需与 GFxTutorial::OnResetDevice 中的 InitHAL 调用相配合。

当与您自己的引擎集成时，设法调用 InitHAL 和 ShutdownHAL 函数和其他相关函数来分配和释放 D3DPOOL_DEFAULT 资源。

步骤#9：视窗设置

必须在屏幕上确定一个特定的视窗以便描绘动画。在本实例中，动画占据了整个视窗。由于屏幕的分辨率可以改变，每次在 D3D 设备复位时我们也将复位视窗，可添加如下代码到 GFxTutorial::OnResetDevice 来完成本操作：

```
// 使用窗口客户端大小尺寸作为视窗大小。
RECT windowRect = DXUTGetWindowClientRect();
DWORD windowWidth = windowRect.right - windowRect.left;
DWORD windowHeight = windowRect.bottom - windowRect.top;
pUIMovie->SetViewport(windowWidth, windowHeight, 0, 0,
                      windowWidth, windowHeight);
```

SetViewport 函数中的前两个参数描述了使用的画面缓存区大小，通常为 PC 应用程序的窗口大小。其余四个参数描述了 Scaleform 可以描绘的窗口大小。

画面缓存区大小参数与 OpenGL 和其他平台兼容，不同平台间具有不同的坐标系数或画面缓存区的大小信息不可获取。

Scaleform 提供了一系列函数用于控制 Flash 画面在窗口范围内的缩放和移动。在 4.2 小节中应用程序准备好可以执行时我们可以检查这些功能选项。

步骤#10：描绘 DirectX 场景

描绘操作在 ShadowVolume 的 OnFrameRender()函数中实现。所有的 D3D 描绘调用都是在函数 BeginScene() 和 EndScene()之间执行。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()函数将动画向前推进 deltaTime 秒。动画的播放速度由应用程序所在的当前系统时间来控制。必须提供实时时钟给 GFx::Movie::Advance，确保动画能够在不同的的硬件配置中能够正确回放。

步骤#11：保留描绘状态

Render::Renderer2D::Display 调用 DirectX 在 D3D 设备上描绘动画帧。考虑到性能因素，D3D 设备状态信息，如混合模式和纹理存储设置等信息将不被保留，因此调用 Render::Renderer2D::Display 后 D3D 的设备状态信息将有所不同。有些应用程序可能受到负面影响。最直接的做法是在调用之前先保存设备状态信息，调用之后再恢复这些信息。游戏引擎在 Scaleform 渲染后重新初始化必要的状态信息，这样能达到优越的性能。本指南中用到了 DX9 状态块函数来简单的操作保存和恢复状态信息。

DX9 状态块在应用程序整个执行过程中被分配，并在 GFxTutorial::AdvanceAndRender()函数调用之前和之后被使用。

```
// 调用Scaleform前保存DirectX状态信息
g_pStateBlock->Capture();

// 描绘画面帧，推进时间计数
gfx->AdvanceAndRender();

// 恢复DirectX状态信息，避免干扰游戏描绘状态
g_pStateBlock->Apply();
```

步骤#12：禁止默认 UI

最后一步为禁止 DXUT 默认用户界面。这由注释相关部分代码的方法来完成，可参考 4.1 小节 ShadowVolume.cpp 文件。将该文件与之前小节的代码相对比查看改变部分。所有与 DXUT 相关的修改都有如下注释：

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

现在，我们在 DirectX 应用中已经拥有了一个硬件加速功能的 Flash 动画。

图 3：Scaleform Flash 用户界面的 ShadowVolum 应用程序

4.2 缩放模式

GFx::Movie::SetViewport函数调用使得视窗与屏幕分辨率保持一致。若屏幕的纵横比与Flash画面的纵横比例不协调，则Flash画面将扭曲。Scaleform提供如下函数：

- 保持画面纵横比例或者自由延伸. 。
- 移动画面至中间、角落或者视窗边沿。

这些函数在4:3或者宽屏显示中绘制同个图形非常有用。Scaleform的优势之一就是其可伸缩矢量图使得能够自由的与各种分辨率的显示尺度相适配。而传统的位图在不同分辨率下需要设计不同尺寸的位图分别显示。例如，一个设置成为低分辨率的800x600位图，另外一个设置成高分辨率的1600x1200位图。Scaleform可以把同一个矢量图缩放到任何比例的分辨率尺寸。除此之外，在某些游戏场景元素描绘中位图更加合适，Scaleform也完全支持。

GFx::Movie::SetViewScaleMode定义了缩放操作。为了保证画面能够与视窗相适应而不导致原始尺寸比列变形，可调用GFxTutorial::InitGFx()函数与其他调用相结合来创建GFx::Movie对象：

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

SetViewScaleMode函数的参数含义在在线文档中有阐述并说明如下：

SM_NoScale	将Flash画面恢复到原始尺寸
SM_ShowAll	维持原始比例，缩放到视窗大小
SM_ExactFit	不保持原始比例，缩放到视窗大小。填充整个视窗，但可能会发生变形。
SM_NoBorder	缩放到视窗大小，填充整个视窗，并保持原始比例，增加画面剪辑。

SetViewAlignment能够充分地控制画面在视窗中的位置。当需要维持原始纵横比例，使用SM_NoScale或SM_ShowAll参数时，视窗的某些部分将空白。在此情况下选择SetViewAlignment参数来确定画面在视窗中的对齐位置，如下代码表示按钮的界面垂直居中并右对齐：

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

试着改变SetViewScaleMode 和SetViewAlignment的参数，并调整窗口尺寸，查看应用程序的变化情况。

SetViewAlignment函数只在SetViewScaleMode函数设置为SM_NoScale默认参数时才起作用。更多复杂的关于对齐、缩放和移位需求，Scaleform支持动态脚本扩展定位动画位置和尺寸大小，这些实例动态脚本在文件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 输入事件处理

ShadowVolume 管线做了修改用来描绘 Scaleform 中的 Flash；我们现在需要与 Flash 播放动画交互。例如，鼠标移动到按钮上方使得按钮能突出显示，在文本框输入文本能显示新的字符。

GFx::Movie::HandleEvent 传递 GFx::Event 对象来描述事件类型和其他信息，如按键或者鼠标动作。应用程序简单得构造输入时间，并传递给对应视图的 GFx::Movie。

4.3.1 鼠标事件

ShadowVolume 在 MsgProc 调用中获得 Win32 输入事件。为 GFxTutorial::ProcessEvent 增加一个调用来运行相关代码使得 Scaleform 能够处理过程事件。以下代码表示的动作有 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 定位鼠标焦点位于视窗的左上方，而与画面的原始尺寸无关。下面例子就阐明了本含义：

实例#1: 视窗与屏幕尺寸匹配

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

本例中无需任何转换：由于画面被定位在坐标(0, 0)位置，鼠标在画面窗口中的位置也与左上方相关联。窗口的尺度与动画原始分辨率尺度的对应由 Scaleform 内部实现。

实例#2：视窗比屏幕小，定位于屏幕的左上角

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

同样，本例中也无需做任何转换。由于视窗被缩小，按钮的尺寸和位置也随之改变。但是不管是 HandleEvent 事件还是窗口屏幕的基准仍定位在视窗的左上角，无需任何转变。视窗尺寸缩放和动画原始尺寸由 Scaleform 内部处理。

实例#3：视窗比屏幕小居中显示

```
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);
```

在本例子中屏幕视窗的转换需要对应基准坐标。动画不在位置(0,0)，新的位置位于坐标为(screen_width / 2 – movie_width / 2, screen_height / 2 – movie_height / 2)处，必须根据屏幕坐标计算准确的坐标位置。

注意如果 Flash 画面通过 GFx::Movie::SetViewAlignment 居中或者用其它方式对齐，无需执行转换。只要鼠标动作与 GFx::Movie::SetViewport 相关联。SetViewAlignment 和 SetViewScaleMode 的对齐和缩放操作将由 Scaleform 内部来处理。

4.3.2 键盘事件

键盘事件也由 GFx::Movie:HandleEvent 来处理。有两类键盘事件：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 类似于原始的扫描码；而 GFx::CharEvent 类似于 ASCII 字符处理器。在 Windows 操作系统中，GFx::CharEvent 事件由 WM_CHAR 消息产生；GFx::KeyEvents 事件由 WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN 和 WM_KEYUP 消息产生。以下为相关的几个例子：

- 同时按住 'c' 键和 SHIFT 键盘:：一个 GFx::KeyEvent 事件即产生，对应的 WM_KEYDOWN 消息描述的过程为：
  - 'c' 键已点击，获得了键值
  - 键盘已被按下
  - SHIFT 键有效

- 同时 GFx::CharEvent 应该触发响应 WM_CHAR 消息并传递 "触发" 的 ASCII 码字符'C' 给 GFx。

- 一点'c'键被释放，将产生 GFx::KeyEvent 事件对应消息 WM_KEYUP。当键释放时无需传递 GFx::CharEvent 事件。

- 按下 F5 键盘： 当键盘按下时随即产生 GFx::KeyEvent 事件，当键弹起产生第二个事件。由于 F5 不代表任何可打印的 ASCII 码，所以整个过程不需要产生 GFx::CharEvent 事件。

在键盘按下和弹起时产生独立的 GFx::KeyEvent 事件。为了保持平台无关性，各个键值在 GFx_Event.h 头文件中有统一定义以配合 Flash 内部使用。GFxPlayerTiny.cpp 例子中和 Scaleform Player 程序中均包含了键值，将 Windows 的按键码转换成 Flash 内部使用的键值。本节所描述的最终代码还包含 ProcessKeyEvent 函数，可在与自定义 3D 引擎集成中重用：

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

从 Windows 的 WndProc 函数调用相关消息，包括 WM_CHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_KEYDOWN 和 WM_KEYUP 消息。相应地触发 Scaleform 事件并传递给 pMovie。

发送 GFx::KeyEvent 和 GFx::CharEvent 事件是非常重要的。例如，大多数文本框需响应 GFx::CharEvents 事件，因为文本框需要获取可打印 ASCII 字符。同样的，文本框也需获取 Unicode 字（比如用中文输入框 IME），. 在 IME 输入情况下，原始键值是没有用的，只有最终的字符（通常由 IME 中的若干个键值组合而成）才输入到文本框。于此相反，下拉框需要通过 GFx::KeyEvent 事件截取 Page Up 和 Page Down 键，由于该建不是用来显示字符，而是用来做控制用。

该功能的相关代码在 4.3 小节中有详细描述。运行程序并移动鼠标至按钮上方。按钮将突出显示，那些无需与 3D 引擎集成的实例将能够很好运行。点击“Settings”出现 DX9 配置界面，无需任何 C++代码，因为 d3d9guide.fla 用动态脚本来实现此简单的逻辑功能。

同时命令键也可以传递给 Flash，空格键控制 UI 是否可见。固定 UI 观测应用程序的帧速率并测算 Scaleform 在应用程序中的性能。与 DXUT 界面对比帧速率查看在 DXUT 和 Scaleform 中的性能比较。务必用发布编译方式的程序来做对比，不要用调试编译模式的程序。通常 Scaleform 不会对应用程序产生多大影响，也不会比 DXUT 速度要慢很多。请记住 Scaleform 界面支持全部功能的 Flash 动画（如文本输入框的隐现）、动画窗口转换，同时能够任何比例地缩放按钮、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 点击测试

运行应用程序并移动鼠标，同时按下鼠标左键改变指针方向进入 3D 的世界。然后移动鼠标至其中一个用户界面元素的上方并重复此操作。尽管鼠标点击在界面中产生了预期的响应，仍然使镜头在移动。

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 键盘焦点

运行应用程序点击“Change Mesh”按钮打开文本输入框。输入文字到输入框，可以看到键盘的输入字符，代码如 4.3.2 中所示。但是，输入 W,S,A,D 和 Q 到输入文本框，同时也移动了 3D 视镜。键盘事件被 Scaleform 处理，但也同时传递给了 3D 视镜。

要解决这个问题需要判断是否输入文本框已获得焦点。6.1.2 小节将描述动态脚本如何用来发送事件给 C++以确定事件处理焦点。

5 文字本地化和字体介绍

5.1 字体概述和容量

Scaleform 提供了一套高效、易扩展的字体和本地化系统。多样化的字体、点阵大小和风格能同时有效地显示出来，且占用较低的内存空间。字体数据可以从内嵌字体数据的 Flash 文件、共享字体库、操作系统和直接 TTF 字体库中获取。基与矢量图的字体压缩数据在大尺寸亚洲文字应用中减少了内存空间。字体的支持具有跨平台功能，在控制台系统、Windows 和 Linux 中都能够支持。Scaleform 关于字体和国际化覆盖范围的完整文档可在开发中心文档页面中找到：
https://gameware.autodesk.com/scaleform/developer/?action=doc 。

通常字体的描绘手段为为每个文字的不同的字体准备不同的纹理外观，映射到不同的字体使用中去。附加的字体纹理需要不同的尺寸和风格。如拉丁文文字总的数据存储量还可以接受，但是亚洲文字有 5000 个不同的文字，实现起来就不大方便。需要大量的内存和处理时间。同时描绘不同文字大小和风格更加不可能。

Scaleform 用动态字体缓存来解决该问题。字符根据显示需要放入到缓存，缓存中文字数据随时可更新和替换。不同的文字尺寸和字体风格可以共享公共缓存，Scaleform 用到了智能的轮廓压缩算法来实现共享。Scaleform 使用矢量字体，这意味着只需单个 TTF 字体存储在内存，可以用来表示各种大小的字体。另外，Scaleform 还支持“仿斜体”和“仿粗体”功能，使得单个字体矢量能够按照要求表示为斜体和粗体，节省额外内存开销。

超大字体可以通过栅格镶嵌直接描绘出来。这解决了在游戏开始屏幕标题中超大文字显示的问题。在 Bin\Data\AS2\Samples\FontConfig 目录中找到字体配置实例，将 sample.swf 文件拖放到打开的 Scaleform Player D3D9 窗口。

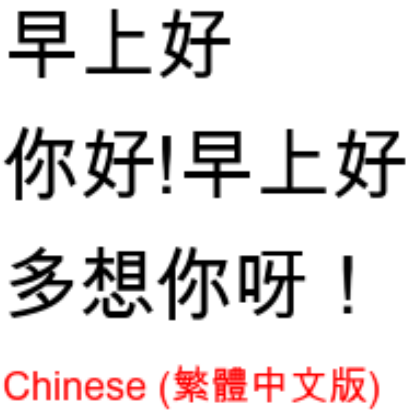

图 3a：中文小字体

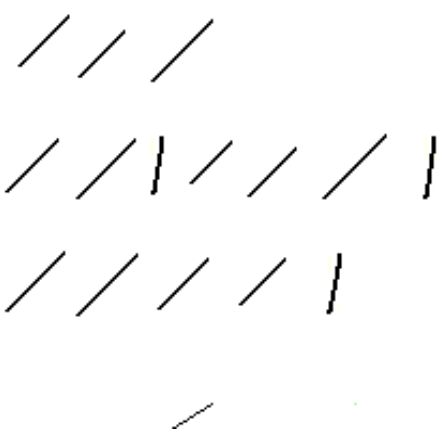

图 4b：线状描绘

选择 CTRL+W 组合键显示这些文字的线状描绘，可以看到每个文字用两种纹理映射而成。这里字体比较小，所以用位图来描绘比较有效。

当增加窗口尺寸而文字保持在线状模式。文字将从纹理映射转换成纯色覆盖模式：

图 4c：大号中文字

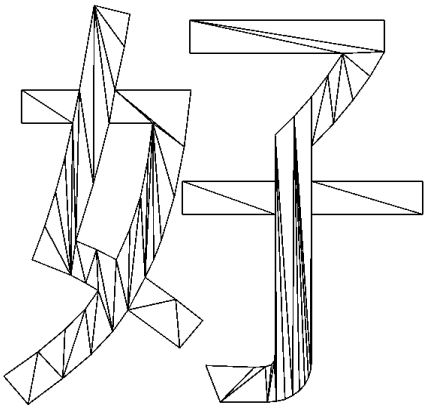

图 4d：线状描绘

大号字中，使用纯色覆盖更加有效。对大的位图操作需要消耗大量内存。只有像素点需要着色，在字符的空白处避免浪费处理器资源。图 4c 和图 4d 中，Scaleform 将字符通过一个固定的阈值，检测其尺寸大小，并将其栅格化，当作几何图形来处理，而不作为位图来处理。

字体柔性阴影、模糊和其他效果都被支持。只要简单的将对应的滤镜作用于 Flash 文本域来产生预期的效果。更多细节请参考字体和文本设置概述，前面有对其的链接。例子可以从开发中心下载，文件为 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

图 5：文本效果

6 C++、Flash 和动态脚本接口

Flash 的动态脚本语言支持交互动画的创建。按钮点击、到达特定帧或者导入画面等事件都能执行代码来动态改变动画内容、控制动画流程，甚至启动附加的动画。动态脚本功能强大，以至于能够在 Flash 中创建完整的小游戏。与大多数编程语言类似，动态脚本支持变量和子函数。Scaleform 提供 C++接口直接操作动态脚本变量、数组以及直接调用动态脚本子程序。Scaleform 还提供了调用回收机制，使得动态脚本能够传递事件和数据给 C++程序。

6.1 *动态脚本到 C++*

Scaleform 提供了两种机制，使得 C++应用程序能够从动态脚本接收事件：FSCommand 和 ExternalInterface 函数。FSCommand 和 ExternalInterface 都在 GFx::Loader 中注册一个 C++事件句柄来接收事件信息。FSCommand 事件由动态脚本的 fscommand 函数发出，接收两个字符串参数，无返回值。ExternalInterface 事件当动态脚本调用 flash.external.ExternalInterface.call 函数时被触发，接收一个 GFx::Value 参数列表（在 6.1.2 中有描述），返回值给调用者。

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

动态脚本 fscommand 函数传递命令和数据给主应用程序。该函数在动态脚本的典型应用如下所示：

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

任何非字符串参数传递给 fscommand 函数，如布尔型或者整型，将被转换成字符串。而 ExternalInterface 可直接接收整型参数。

此处传递两个字符串给 GFx FSCommand 句柄。一个应用程序通过 GFx::FSCommandHandler 子类注册 FSCommand 句柄同时注册 GFx::Loader 或者单个 GFx::Movie 对象相关的类。若设置一个命令句柄到 GFx::Movie，则能够接收到 fscommand 函数在动画实例中的调用返回值。Scaleform PlayerTiny 实例展现了这个过程(搜索“FxPlayerFSCommandHandler”)，我们将添加类似的代码到 ShadowVolume。本节详细代码步骤位于 6.1 小节。

首先，GFx::FSCommandHandler 子类：

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

Callback 方法接收传递给动态脚本中的 fscommand 的两个字符串参数，同时接收了指向调用 fscommand 的特殊动画实例的指针。

接下来，在 GFxTutorial::InitGFx()中的 GFx::Loader 对象创建后注册句柄：

```
// 注册 FSCommand 句柄
Ptr<FSCommandHandler> pcommandHandler = *new OurFSCommandHandler;
gfxLoader->SetFSCommandHandler(pcommandHandler);
```

通过 GFx::Loader 注册句柄使得每个 GFx::Movie 继承该句柄。SetFSCommandHandler 可以在每个实例中被调用来跳过这些默认设置。

我们的自定义句柄简单的将每个 fscommand 事件打印到调试控制台窗口。运行 ShadowVolume 并点击“Toggle Fullscreen”按钮。注意到只要用户界面事件发生就会打印出相关信息：

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

在 Flash Studio 中打开 d3d9guide.swf 文件，打开动态脚本面板(F9)。将屏幕上打印出的事件信息与位于 ActionScript block for Symbol Definition(s) → hud → var : Frame 1
中的 fscommand 调用做对比：

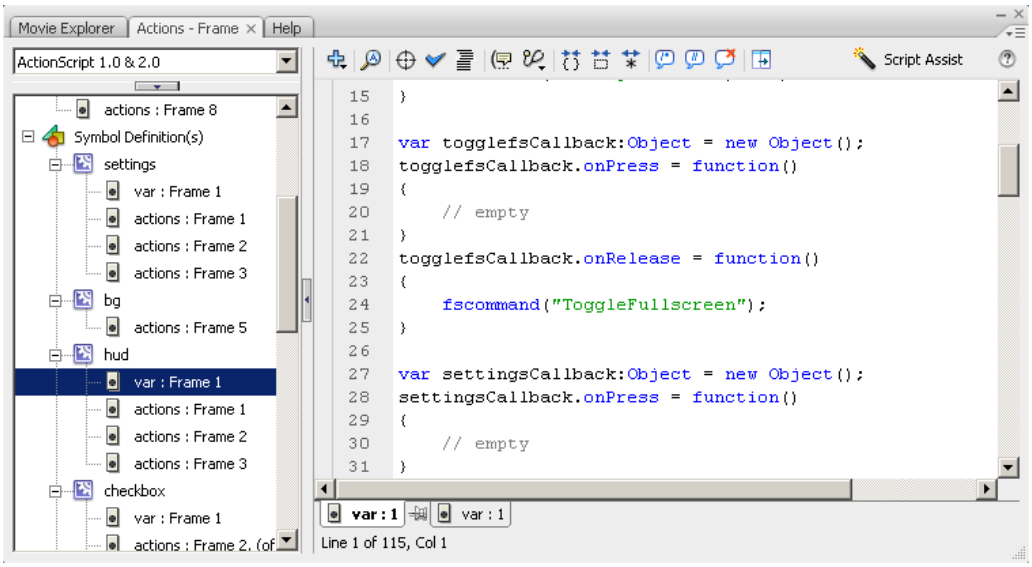

图 6：动态脚本 fscommand()调用

作为演示，将 fscommand(“ToggleFullscreen”) 改变为 fscommand(“ToggleFullscreen”, this.UI._visible) 来打印 C++可视的 this.UI._visible 值。导出 flash 动画(CTRL+ALT+Shift+S)并替换 d3d9guide.swf 文件。

当 FSCommand 事件发生时启动相关代码，用户界面上的按钮可集成到 ShadowVolume 实例中来。例如：增加以下代码行，使 fscommand 句柄进入全屏模式：

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

DXUT 函数 OnFrameMove 在一帧被描绘时调用。在 OnFrameMove 函数返回后增加如下相应代码：

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

通常，事件句柄不应该被阻塞，需要尽快地返回给调用函数。事件句柄可在任何时候被调用，甚至在描绘一个帧的中间。这就是为何 fscommand 句柄设置 doToggleFullscreen 参数并让 DXUT 在 OnFrameMove 适当的时间执行操作。

6.1.2 ExternalInterface

Flash ExternalInterface 调用方法与 fscommand 类似，但更具有优越性，因为它提供了更多灵活的参数并可返回值。

注册一个 ExternalInterface 类与注册一个 fscommand 句柄类似：

```
class OurExternalInterfaceHandler : public ExternalInterface
{
  public:
      virtual void Callback(Movie* pmovieView,
                            const char* methodName,
                            const Value* args,
                            unsigned argCount)
      {
            GFxPrintf("ExternalInterface: %s, %d args: ",
                                          methodName, argCount);
            for(unsigned i = 0; i < argCount; i++)
            {
                  switch(args[i].GetType())
                  {
                    case Value::VT_Null:
                        GFxPrintf("NULL");
                        break;
                    case Value::VT_Boolean:
                        GFxPrintf("%s", args[i].GetBool()? "true" : "false");
                        break;
                    case Value::VT_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 注册句柄：

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

当文本输入框获得或者失去焦点时，可以从动态脚本创建一个外部接口调用。选择 F9 键查看 ActionScript for Symbol Definitions() → hud → var : Frame 1：

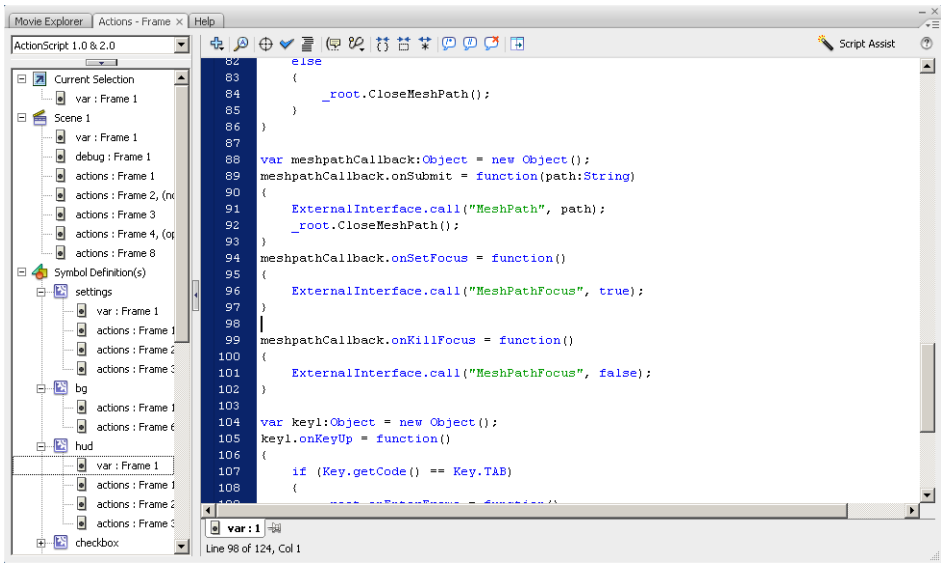

图 7：MeshPathFocus 当文本输入框获得或者失去焦点时被调用。

外部接口调用将触发外部接口句柄并传递输入文本框(true 或者 false)和专用命令字符串“MeshPathFocus”的焦点状态信息。打开输入文本框，点击文本区，然后点击屏幕空白处以从文本区移开焦点。控制台将输出如下信息：

```
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
```

事件句柄将检测何时获得焦点或者何时失去焦点并传递该信息到 GFxTutorial 对象：

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

若文本框获得焦点，GFxTutorial::ProcessEvent 将只传递键盘事件给动画。当一个键盘事件传递给文本框时，将产生一个标记，防止传递到 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 码为 32)和 tab(ASCII 码为 9)对应于“Toggle UI” 和 “Settings” 按钮，总是被传递。

为了使用户能够改变描绘的网格，点击“Change Mesh”按钮来打开文本框，输入新网格名，按下回车键。当回车键被按下时，文本框将调用动作脚本事件句柄，其调用了用新名字命名的外部接口。调用 OurExternalInterfaceHandler::Callback 中的代码如下所示：

```
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，实际工作在 DXUT 的 OnFrameMove 调用函数中完成。代码建立在 DXUT 接口的事件句柄基础之上。

6.2 *C++ 到动态脚本*

6.1 小节介绍了动态脚本如何被 C++调用。本节描述了使用 Scaleform 函数如何与其他单元通信，使得 C++程序初始化与播放动画的通信。Scaleform 支持 C++函数读取和设置动态脚本变量并调用动态脚本子程序。

6.2.1 操作动态脚本变量

Scaleform 支持 GetVariable 和 SetVariable 函数，通过这两个函数可以直接操作动态脚本变量。6.2 小节中的对应代码修改用来导入黄色 HUD 显示(fxplayer.swf)，该显示是 Scaleform Player 程序使用并在按下 F5 键时增加计数功能：

```
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 返回变量_root.counter 的值。起初该变量不存在，GetVariableDouble 返回零。计数器增加计数，新的值通过 SetVariable 保存到_root.counter 中。当前文档关于 GFx::Movie 列表中包含了 GetVariable 和 SetVariable 参数的不同变量值。

fxplayer Flash 文件具有两个动态文本区，能够通过应用程序设置专用文本。MessageText 文本区位于屏幕中间，HUDText 变量位于屏幕的左上角。根据_root.counter 变量的值来产生一个字符串，SetVariable 用来更新消息文本。

性能注释：改变 Flash 动态文本区的首先方法为设置 TextField.text 变量或者调用 GFx::Movie::Invoke 运行动态脚本程序来改变文本（更多信息参考 6.2.2）。禁止将动态文本区与专用变量绑定后通过改变该变量来改变文本。尽管这样做是可行的，但是有一个缺陷，因为 Scaleform 在每个帧都要检查变量的值。

图 8：SetVariable 改变 HUD 文本值。

上面用法中将变量直接当成字符串或者数字来处理。在线文档中描述了使用新型 GFx::Value 对象句法有效地处理变量，直接将变量作为整数处理，取消了字符串到整数的转换过程。

SetVariable 函数拥有一个可选使用的第三个参数，参数表示 GFx::Movie::SetVarType 类型，用来定义“粘贴”分配。这个参数在变量被分配但在时间轴上还未被创建时起到作用。例如：假设文本域 _root.mytextfield 直到动画的第 3 帧才被创建。如果 SetVariable(“_root.mytextfield.text”, “testing”, SV_Normal)在动画刚被创建的第 1 帧就被调用，则分配不起作用。如果调用由 SV_Sticky 产生（默认值），则请求排入队列，直到第 3 帧中_root.mytextfield.text 的值有效时执行。通过 C++来初始化动画将变得更加容易。

SetVariableArray 通过单个操作传递整个变量的数组到 Flash。该功能可以用来动态移动下拉列表控制操作。以下代码处于 GFxTutorial::InitGFx()函数中用来设置_root.SceneData 下拉框的值。

```
// 初始化场景
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 小节代码包括了附加的外部接口句柄，对应于亮度控制、光源数量、场景类型和其他呈现原始 ShadowVolume 接口的控制单元。

注释：本例子中通过 C++来组装下拉菜单作为展示。通常，包含静态选择列表的下拉菜单应该通过动态脚本初始化而不是 C++代码。

6.2.2 执行动态脚本子程序

除了改变动态脚本变量，动态脚本代码也可以通过 GFx::Movie::Invoke 方法调用。这在一些复杂过程处理中非常有用，如触发动画、改变当前帧、可编程改变 UI 控制状态和动态创建新按钮或文本等用户界面等。

本节使用 GFx::Movie::Invoke 来编程实现“Change Mesh”文本输入框的打开，快捷键 F6 和 F7 分别用来打开和关闭该文本输入框。由于当单选按钮被选中时动画才产生，所以使用 SetVariable 不能改变该状态。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 被调用来执行 WM_CHAR 键盘句柄中的 openMeshPath 程序：

```
...

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 的动态脚本代码位于第 1 帧，确保动态脚本程序在第 1 帧画面播放时被执行。使用钩子时常会出现的错误为调用动态脚本程序时无法获得脚本，这种情况下错误信息将被输出到 Scaleform 脚本文件中。直到动态脚本程序关联帧开始播放或者其关联嵌套对象被导入，动态脚本程序才能改变其变量的值。当 GFx::Movie::Advance 初次调用时或者 GFx::MovieDef::CreateInstance 被调用且其参数 initFirstFrame 为 true 时，位于第 1 帧中的所有动态脚本代码都可以被获取。

本例子用了钩子的 printf 类型。其他版本的功能函数使用 GFx::Value 高效地处理非字符串类型的参数。InvokeArgs 调用方法类似，除非其参数 va_list 使应用程序提供指向参数列表的指针。Invoke 和 InvokeArgs 之间的关系与 printf 和 vprintf 之间的关系类似。

6.3 *多 Flash 文件之间通信*

到现在为止所讨论的通信方法都与 C++有关。在一个大的应用程序中，用户接口被分割成多个 SWF 文件，则需要编写大量 C++代码使得界面的不同组件之间互相通信。

例如，一个 MMOG 游戏有一个总清单 HUD、分清单 HUD 和交易室。宝剑和金钱等条目可以从游戏者的总清单中移交到交易室并交给另外一个游戏者。但玩家穿上一件衣服，该条目需要从总清单 HUD 中移交到分清单 HUD。

这三个界面必须分为三个独立的 SWF 文件存储并独立导入以节省内存。但是，C++代码在这三者的 GFx::Movie 对象间通信很快就会消耗衰竭。

还有更好的方法来解决这个问题。动态脚本支持 loadMovie 和 unloadMovie 方法，这类方法使得多个 SWF 文件在单个 GFx::Movie 中导入和导出交换。由于 SWF 动画在相同的 GFx::Movie 中，它们可以共享变量空间，这样无需 C++代码在每次导入时初始化动画。

在 MMOG 例子中，总清单可以描述成名为_global.inventory 的动态脚本数组。当其中一个 SWF 界面导入时，则根据该数据中数据来绘制条目。当其中一个 SWF 界面用动态脚本 unloadMovie 方法释放时，_global.inventory 数组仍然可被其他接口获取。

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\section6.2 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 = pContainerMovie->Invoke("_root.UnloadFlashLevel",
                                              "%d", 5);
        GFxPrintf("_root.UnloadFlash returns '%d'\n", (int) retval);
    }
    else if (wParam == VK_F3)
    {
        bool retval = pContainerMovie->Invoke("_root.UnloadFlashLevel",
                                              "%d", 6);
        GFxPrintf("_root.UnloadFlash returns '%d'\n", (int) retval);
    }
}
```

以上代码将 HUD 导入到图层 5，将主界面导入到图层 6。Flash 动态脚本所在的图层与场景的 z 轴相关联。在相同 GFx::Movie 中，上层图层中的画面将覆盖下层图层中的画面。动画最初导入 Container.swf 文件时默认放在图层 0。

在不同图层中的变量可以用_level 关键字来访问。比如，位于图层_level6 中的主界面可以直接访问 HUD 的文本域，只需改变_level5.MessageText.text 的参数即可。同样的这两个位于不同图层的画面也可以通过关键字_global 变量来访问 Flash 文件中的全部变量空间。例如，每个画面都可以访问_global.counter 全局变量。这样有一个好处就是当该两个画面都释放后，在_global 全局命名空间中的变量不会丢失。不管画面的导入还是导出均可以访问固定不变的_global 全局命名空间。更多关于_level, _global 变量和动态脚本变量的命名空间相关信息，请参考 Adobe Flash 相关文档。

运行应用程序按 F8 导入 HUD。然后按 F9 导入主界面。注意到当导入主界面时会有一定的延时。这是因为导入中文字体需要耗费一定的时间。可以用一定的方法来加速该过程，将 SWF 文件预先编译为 Scaleform 文件（参考第 8 章），设置 Scaleform 多线程导入功能，使得在背景画面中就可以导入共享字体文件（参阅开发中心网站的字体和文本相关文档）。

按 F8 提出 HUD，然后按 F5 使能计数器计数，在 6.2 节中添加了计数器并停止了工作。SetVariable 方法应用了_root.counter，但是现在 HUD 导入到了_level5 命名空间，所以计数部分代码需要做如下修改：

```
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 文件中位于图层 6 中的动态脚本直接访问，使得接口之间不通过 C++代码就可以直接通信。

运行应用程序，按 F8 键导入 HUD，按 F5 开启计数器。按 F2 释放 HUD，然后按 F8 重新导入 HUD 继续用 F5 来启动计数。当 HUD 被释放时候，计数值丢失，继续从零开始计数。这是因为计数变量位于

_level5 当中。将变量从_level5.counter 改变为_global.counter 然后再作前面的操作。将变量保存到_global 中后，无论是动画导入还是导出变量值都将保存。

7 渲染到纹理

渲染到纹理是 3D 渲染中一项通用的先进技术。此技术使用户可以渲染到纹理表面，而非后台缓冲区。然后，就可以把结果产生的纹理用作常规纹理，并可根据需要将常规纹理应用到场景几何。例如，此技术可用来创建“游戏内广告牌”。首先，在 Flash Studio 中将您的广告牌撰写为 SWF 文件，将该 SWF 文件渲染到一个纹理，然后将该纹理应用到适当场景几何。
在此教程中，我们只需将 UI 从上一个教程渲染到后墙。如果您按鼠标中键到处移动光源，您就会看到相应的 UI 变化情况 。

现在我们将循序渐进地演练此示例的内部运作情况。

1. 在所有以前的教程中，我们加载了 cell.x 场景文件。此文件包含显示房间地板、墙壁和屋顶所需的顶点坐标、三角指数和纹理信息。如果您看看此文件，就会注意到同一墙壁纹理 (cellwall.jpg) 用于全部四面墙壁。我们想要只在后墙上渲染我们的 UI。因此，我们创建一个单独的纹理，用于称为 cellwallRT.jpg 的后墙。接下来，修改 MeshMaterialList 阵列以引用后墙的纹理。

```
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. 以前的教程中已经提到，cell.x 文件的解析是由 DXUT 框架在内部进行的。在此解析过程中，DXUT 初始化顶点和索引缓冲区，并创建在资料列表中引用的纹理。这些纹理是使用 CreateTexture 调用创建的；不过，传递到此函数的用法参数不适用于渲染纹理 (Render Texture)。有关此话题的更多信息，请参阅 DXSDK 文档。要创建一个可用作渲染目标的纹理，请使用恰当的用法参数重新创建渲染纹理，并将其附加到背景网络。同时释放原来由 DXUT 创建的纹理，以避免内存泄漏。

```
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. 接下来修改 AdvanceAndRender 函数以便将 GFx 渲染到我们的纹理，而不是渲染到后台缓冲区上。此过程的关键步骤如下：
   a. 首先保存原始后台缓冲区表面，这样一旦我们完成渲染就可以恢复。
      ```
      pd3dDevice->GetRenderTarget(0, &poldSurface);
      pd3dDevice->GetDepthStencilSurface(&poldDepthSurface);
      ```
   b. 接下来，从我们的纹理获取渲染表面，并将其设置为渲染目标。
      ```
      ptex->GetSurfaceLevel(0, &psurface);
      HRESULThr = pd3dDevice->SetRenderTarget(0, psurface );
      ```
   c. 将电影视口调整到我们的纹理的尺寸，调用 Display，然后恢复到步骤 a 中保存的原始渲染表面。

8 GFxExport 预处理

直到现在我们才开始直接导入 SWF 文件。这个开发流程是为了使设计者在开发 SWF 文件的过程中能够交换新 SWF 文件内容，无需用到游戏执行代码便可看到游戏的运行结果。但是，在发布版本中包含 SWF 文件不被推荐，因为导入 SWF 文件需要预先处理影响导入时间。

GFxExport 是处理 SWF 文件转换为导入流优化格式的有用工具。在预处理的过程中，图像被萃取并放置到不同的文件中以便由游戏资源引擎统一管理。图像能够转换成 DDS 格式，采用 DXT 纹理压缩优化导入过程、减少实时运行时的内存消耗。内嵌字体被多重压缩，字体纹理可以任意选择以适应位图文字的使用。GFxExport 输出可以有多种压缩方法。

使用 Scaleform 文件开发是件非常容易的事情。GFxExport 工具支持广泛的配置项，在线帮助文档中就有相关描述。将 d3d9guide.swf 和 fxplayer.swf 文件转换成 Scaleform 格式：

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

gfxexport.exe 可执行文件位于目录 C:\Program Files\Scaleform\$(GFXSDK) 。这些命令同时包含在 convert.bat 批处理文件中，具体步骤查看第 7 节。

-i 选项表示图像格式，这里参数为 DDS。DDS 参数使在 DirectX 平台上发挥优势，因为它支持 DXT 纹理压缩，DXT 纹理压缩通常在实时运行时能节约四分子三的内存。

-c 选项使能压缩。只有在 Scaleform 文件中的矢量和动态脚本计数器才被压缩。图像的压缩格式取决于图像的输出格式和 DXT 压缩选项。

-share_images 选项通过特征码识别不同文件中的相同图像，且导入时只需导入一份共享副本。

ShadowVolume 应用程序的版本在第 8 节步骤中做相关了修改，使得导入 Scaleform 文件变得简单而容易，只需要修改 GFx::Loader::CreateMovie 的文件名参数即可。

9 下一步

本指南对 Scaleform 的功能做了基本的介绍。需要探讨的其它主题还包括以下方面：

- Flash 游戏中的渲染与纹理。参考 Scaleform Player SWF 与纹理 SDK 的例子，Gamebryo 集成演示和虚拟引擎 3 集成演示。

- 开发中心的 FAQs 中涉及的其他因素

- Scale9 视窗支持文档位于 Scale9Grid Overview 概述相关文档页中：

- 自定义导入：除了从 Scaleform 或者 SWF 导入文件，Flash 内容还可以直接从内存或者游戏资源管理器导入，只需要使用 GFx::FileOpener 的子集。

- 通过 GFx::ImageCreator 可自定义图像和纹理。

- 使用 GFx::Translator 进行语言文字的本地化