Autodesk®
Scaleform®

Scaleform Integration Tutorial

本書では、DirectX 9 のサンプルを使って Windows 上での基本的な Scaleform の使用法と 3D エンジンの統合について紹介しています。家庭用ゲームアプリケーション開発者の方々も必読です。

著者: Ben Mowery

バージョン: 3.09
最終更新日: 2013 年 10 月 7 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Autodesk Scaleform の連絡先:

ドキュメント	Scaleform 4.2 統合チュートリアル (Scaleform 4.2 Integration Tutorial)
住所	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
ホームページ	www.scaleform.com
電子メール	info@scaleform.com
電話	(301) 446-3200
FAX	(301) 446-3199

目次

1 はじめに

Scaleform は、ビジュアル ユーザー インターフェイス(UI)デザインのためのソリューションで、すでに定評のある高性能なミドルウェアです。ディベロッパーは、Adobe Flash® Studio を利用して迅速かつ安価に、新しいツールやプロセスを学習する必要もなく、最新の GPU 加速されたアニメーション UI とベクター グラフィックを作成することができます。Scaleform は、Flash Studio からゲーム UI へと、直接つながるシームレスなビジュアル開発の道を拓きました。

UI だけではなく、ディベロッパーは Scaleform を使用して、3D サーフェスにマッピングされたテクスチャをアニメーション化するなど、3D 環境内部で Flash コンテンツを表示することができます。同様に、3D オブジェクトと動画を Flash UI 内に表示することもできます。結果的に Scaleform は、スタンドアロン UI ソリューションとしても、既存のフロントエンド ゲーム フレームワークを強化する手段としても十分に機能します。

このチュートリアルは、Scaleform をインストールして使って行くプロセスを順を追って紹介しています。DirectX SDK の Shadow Volume サンプルを用いて、Flash ベースの UI を用いて拡張して行きます。

注意: Scaleform はすでに多くの主要なゲーム エンジンに組み込まれています。Scaleform は、すでに対応済みのエンジンでは、最小限のコーディングで直接使用することができます。本書は、カスタム ゲーム エンジンに Scaleform を組み込もうと考えているエンジニア、あるいは Scaleform 機能の技術的に詳細な概要を探している人を主に対象としています。

注意: このチュートリアルを参考にするときは、最新版の Scaleform を使用するようにしてください。このチュートリアルは、Scaleform のバージョン 4 以上を対象としています。

注意: このチュートリアルは一部の古いビデオ カードとは互換性がない場合があります。これは、このチュートリアルがベースにしている DirectX SDK ShadowVolume コードのためであり、Scaleform との互換性の問題ではありません。このチュートリアルの実行中に、“cannot create renderer” (レンダラを作成できません) というエラー メッセージが表示された場合、問題の原因は Scaleform Player アプリケーションが問題なく起動しているかどうかをチェックすることで、明らかにすることができます。

2 ドキュメントの概要

最新の Scaleform ドキュメントは、Scaleform Developer Center からダウンロードすることができます (https://developer.scaleform.com/gfx)。このサイトにアクセスするには、登録 (無料) が必要です。

今のところ、以下のドキュメントがご利用いただけます:

- Web ベースの Scaleform SDK 参考資料
- PDF ドキュメント
- Font Overview: フォントとテキスト レンダリング システムを説明し、海外版製作のためのアート アセットと Scaleform C++ API 両方の設定方法について、詳しく解説しています。
- XML Overview: Scaleform で利用可能な XML のサポートについて説明しています。
- Scale9Grid: Scale9Grid 機能を使用してサイズ変更可能なウィンドウ、パネル、ボタンの作成方法を説明しています。
- IME Configuration: Scaleform の IME サポートをアプリケーションに組み込む方法と、Flash でカスタムの IME テキスト入力ウィンドウ スタイルを作成する方法を説明しています。
- ActionScript Extensions: Scaleform ActionScript 拡張機能について説明しています。

3 インストールとビルドの依存関係

3.1 インストール

Windows 対応の最新の Scaleform インストーラインストーラをダウンロードします。起動させてデフォルトのインストールパスとオプションを保持させてください。DirectX はもし必要なら GF x インストーラによってアップデートされなければなりません。Scaleform は、C:\Program Files\Scaleform\GFx SDK 4.3 ディレクトリにインストールされます。

以下のフォルダで構成されています:

3rdParty

libjpeg や zlib などの Scaleform で必要となる外部ライブラリです。

Projects

上記のアプリケーションを構築する Visual Studio プロジェクトです。

Apps\Samples

Scaleform Player とほかのデモ用サンプルソースコード。

Apps\Tutorial

このチュートリアル用ソースコードとプロジェクト。

Bin

プレビルトサンプルバイナリとサンプル Flash コンテンツを含みます。

FxPlayer:	簡単な Flash テキスト HUD です。
Samples:	ボタン、編集ボックス、キーパッド、メニュー、スピン カウンタなどの UI エレメントを含む各種 Flash FLA ソース サンプルです。
Video Demo:	サンプルの Scaleform Video のデモとファイルです。
Win32:	Scaleform Player とデモ アプリケーションのための、あらかじめコンパイル さ れたバイナリ ファイルです。
AMP:	AMP ツール起動用 Flash ファイル。
GFxExport:	GFxExport は、Flash コンテンツのロードを加速する前処理ツールです。詳細は、第 7 章を参照してください

注意: デモの.sln Visual Studio プロジェクトを再構築することによって、この Win32 ディレクトリのバイナリ ファイルが置き換えられます。

Include

Scaleform コンベニエンスヘッダー。

Lib

Scaleform のライブラリです。

Resources

CLIK のコンポーネントとツールです。

Doc

第 2 章で述べた PDF ドキュメントが含まれています。

3.2 デモのコンパイル

システムが Scaleform を構築できるように適切に構成されていることを検証するため、まず C:\Program Files (x86)\Scaleform\GFx SDK 4.3\Projects\Win32\Msvc90\Demos\GFx 4.1 Demos.sln のデモ ソリューションをビルドします。Visual Studio でこの.sln ファイルを開いたら、D3D9_Debug_Static 構成を選択し、Scaleform Player プロジェクトを作成します。

C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Win32\Msvc90\GFxPlayer\GFxPlayer_D3D9_Debug_Static.exe ファイルの更新日時を確認して、正常に再ビルドされ、プログラムが実行されていることを確認してください。

このプログラムと、[スタート] → [すべてのプログラム] → [Scaleform] → [GFx SDK 4.3] → [Demo Examples] フォルダにあるそれ以外の GFx Player D3Dx アプリケーションは、ハードウェア加速の SWF プレーヤーです。開発中に、Scaleform Flash の再生は、これらのツールを使用してテストし、ベンチマークすることができます。

3.3 Scaleform Player における SWF 再生のベンチマーク

[スタート] → [すべてのプログラム] → [Scaleform] → [GFx SDK 4.3] → [Demo Examples] → [GFx Player D3D9] プログラムを実行します。C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Data\AS2\Samples\SWFToTexture フォルダを開き、3DWindow.swf をこのアプリケーションにドラッグします。

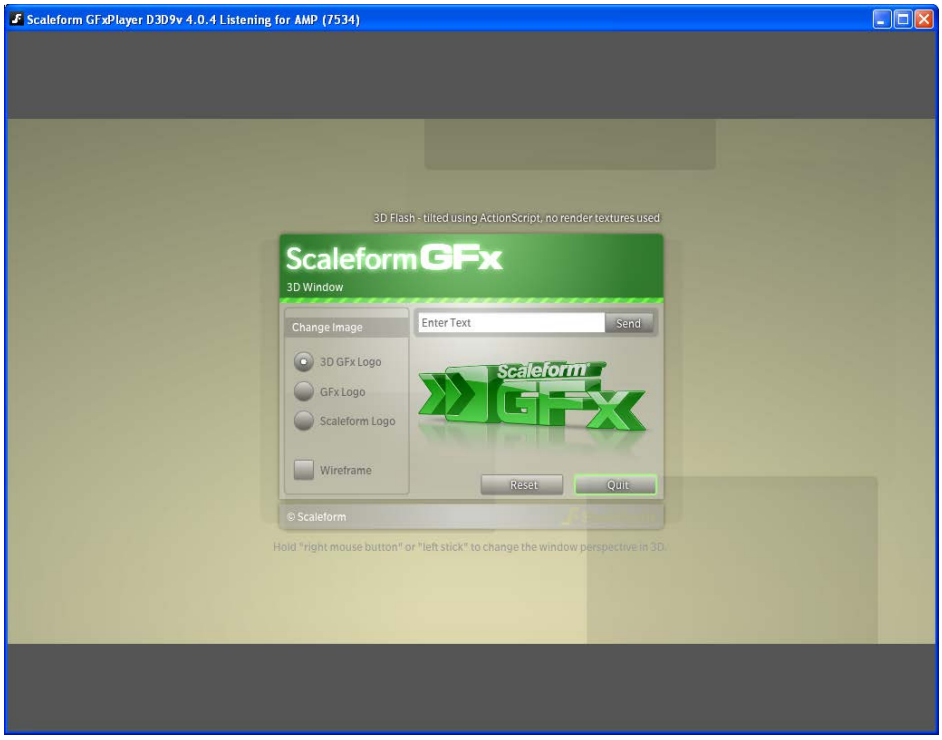

図 1: サンプル Flash コンテンツのハードウェア加速された再生

F1 キーを押すと、ヘルプ画面が表示されます。以下のオプションを試してみてください:

1. Ctrl+F キーでパフォーマンスを測定します。タイトル バーに現在の FPS が表示されます。
2. Ctrl+W キーでワイヤーフレーム モードを切り替えます。Flash コンテンツが Scaleform によってトライアングルに変換され、最適なハードウェア再生が行われていることに注目してください。再度 Ctrl+W キーを押し、ワイヤーフレーム モードを終了します。
3. 拡大するには、Ctrl キーを押したまま左マウス ボタンをクリックして、マウスを上下させます。
4. このイメージをパンするには、Ctrl キーを押したまま右マウス ボタンをクリックして、マウスを移動します。
5. Ctrl+Z キーで通常の表示に戻ります。
6. Ctrl+U キーでフルスクリーン再生に切り替えます。
7. F2 キーでメモリ使用量を含むムービーの統計値を見ることができます。

拡大表示しても、ボタンの四隅のように、曲線のエッジがいかに鮮明に表示されているかに注意してください。ウィンドウのサイズを拡大/縮小すると、Flash コンテンツもそれに応じて変化します。ベクター グラフィックの利点の 1 つは、どのような解像度にも合わせてコンテンツがスケーリングする点にあります。従来のビットマップ グラフィックでは、800x600 の画面と 1600x1200 の画面に、1 つずつビットマップ セットが必要です。

画面の右中央にある “Scaleform” のロゴからも分かるように、Scaleform は、従来のビットマップ グラフィックにも対応しています。デザイナーが Flash で作成するどのようなコンテンツも Scaleform でレンダリングすることができます。

[3D Scaleform Logo] (3D Scaleform ロゴ) ラジオ ボタンを拡大し、Ctrl+W キーでワイヤーフレーム モードに切り替えます。ボタンの円がトライアングルにテッセレートされていることに注意してく

ださい。Ctrl+A キーを何度か押し、アンチエイリアス モードを切り替えます。Edge Anti-Aliasing (エッジのアンチエイリアス) モード (EdgeAA) では、円のエッジ周囲にサブピクセル トライアングルが追加され、アンチエイリアス効果を作成します。これは通常、ビデオ カードを使用したフルスクリーンのアンチエイリアス (FSAA) よりも効率的です。FSAA では、アンチエイリアスの実行に 4 倍のフレームバッファ ビデオ メモリと、4 倍のピクセル レンダリングを必要とします。Scaleform 独自の EdgeAA 技術は、オブジェクトのベクター表現を利用して、曲線のエッジや大きな文字などその効果が最も得られやすい画面領域にのみアンチエイリアスを適用します。トライアングルの数は増えますが、描画プリミティブ (DP) の数は一定に保たれるので、パフォーマンスへの影響は管理できる範囲におさまります。これは通常、ビデオ カードのアンチエイリアス機能よりも効率的であり、EdgeAA は、他の品質設定値と同様にオフにしたり、調整することが可能です。

Scaleform Player ツールは、Flash コンテンツのデバッグやパフォーマンスのベンチマーキングに便利です。タスク マネージャを開いて、CPU 使用率を確認してください。Scaleform はベンチマークするために、可能な限り高い fps 値でレンダリングしているため、CPU 使用率が高くなっている可能性があります。Ctrl+Y キーを押して、フレーム レートを画面のリフレッシュ レート (通常、60 fps) に固定します。CPU 使用率が著しく低下することに注意してください。

注意: お使いのアプリケーションのベンチマークを行うときは、必ずリリース ビルドを実行してください。デバッグ Scaleform ビルドでは、最適なパフォーマンスを得ることができません。

3.4 サンプル ベースのコンパイル

Scaleform 4.3 以降のバージョンをお使いの場合には、
Program Files の Scaleform\GFx SDK 4.3\Apps\ Tutorial の Tutorial ソリューション ファイルを開いてください。Windows の [スタート] メニューから [Scaleform] → [GFx SDK 4.3] → [Tutorial] には Visual Studio 2005 と Visual Studio 2008 のソリューションが含まれています。プロジェクトの構成が "Debug" に設定されていることを確認し、アプリケーションを実行します。

図 2: ShadowVolume アプリケーションとデフォルトの UI

3.5 Scaleform ビルドの依存関係

新規に Scaleform プロジェクトを作成するときは、コンパイルする前に、構成が必要な Visual Studio の設定がいくつかあります。以下の手順を行うときは、このチュートリアルには参照用として相対パスがすでに用意されています。$(GFXSDK) は、ベースの SDK インストール ディレクトリに定義された環境変数であることに注意してください。デフォルトの場所は、C:\Program Files\Scaleform\GFx SDK 4.3\; です。lib が別の場所にあるときは、$(GFXSDK) を含むパスをシステムの lib の場所に合わせて変更する必要があります。ここでは例として、“Msvc90” を使用しているため、Visual Studio 2005, Visual Studio 2008 または Visual Studio 2010 を使用するのであれば、それぞれ Msvc80, Msvc90 または Msvc10 を使用する必要があります。

デバッグ構成とリリース ビルド構成の両方に対して、プロジェクトのインクルード パスに Scaleform を追加します。

$(GFXSDK)\Src
$(GFXSDK)\Include
Visual Studio の [追加のインクルード ディレクトリ] フィールドに以下の文字列をペーストします.

コンビニエンス ヘッダ（GFx.h）がインクルードされていない場合、AS3 を使っているのであれば、必須となる AS3 クラス レジストレーション ファイル「GFx/AS3/AS3_Global.h」をアプリケーションに直接インクルードしてください。コンビニエンス ヘッダは不必要な AS3 クラスを除外するためにカスタマイズしてもかまいません。詳細は Scaleform LITE Customization を参照してください。

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

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

注意: Msvc90 を、使用している Visual Studio のバージョンに対応する文字列に変更してください。

最後に Scaleform ライブラリとその依存関係にあるものを追加します。

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

このサンプル アプリケーションが、デバッグ構成とリリース構成の両方でコンパイルとリンクを続行していることを確認します。参考データとして、Scaleform のインクルード設定とリンカ設定を持つ変更後の.vcproj ファイルは、Tutorial\Section3.5 フォルダにあります。

3.5.1AS3_Global.h と Obj¥AS3_Obj_Global.xxx ファイル

AS3_Global.h と Obj¥AS3_Obj_global.xxx は**全く関係の無い**ファイルですので、デベロッパーの方はご注意ください。
AS3_Obj_Global.xxx ファイルにはいわゆる「グローバル」な ActionScript 3 オブジェクトの実装が含まれています。各 swf ファイルには「script」と呼ばれるオブジェクトが少なくとも 1 つあり、これはグローバルオブジェクトです。各ファイルには GlobalObjectCPP というクラスもあり、これは C++で実装された全てのクラスのグローバルオブジェクトです。これは Scaleform VM の実装に限ります。

AS3_Global.h の目的は全く異なります。このファイルには ClassRegistrationTable アレイがあります。このアレイの目的は対応する AS3 クラスを実装している C++クラスを参照することです。この参照が無ければ、リンカーがコードを除外します。したがって ClassRegistrationTable は実行ファイルの中で定義しておく必要があり、定義されていなければリンカーエラーが出ます。このため、当社のデモプレイヤーには夫々AS3_Global.h が入っています。

インクルードファイルに ClassRegistrationTable を入れ、またデベロッパーが入れる必要のある理由は、ClassRegistrationTable のカスタマイズができるようにすることです（コードサイズを縮小できる可能性があるため）。最も良いカスタマイズの方法は AS3_Global.h のコピーを作成し、不必要なクラスをコメントアウトし（こうするとリンクされない）、カスタマイズしたバージョンをアプリにインクルードすることです。

しかし、この最適化には気を付けなければならない点もあります。AS3 VM での名前の解決はランタイムに行われるため、必要とするクラスがテーブル内でコメントアウトされていればこれが見つからない可能性もあります。そのため、「不必要な」クラスを除外する場合は、その後でもアプリが機能するように注意してください。

4 ゲーム エンジンの統合

Scaleform には、Unreal® Engine 3、Gamebryo™、Bigworld®、Hero Engine™、Touchdown Jupiter Engine™、CryENGINE™、Trinigy Vision Engine™など、ほとんどの主要 3D ゲーム エンジンに統合レイヤーを提供しています。
これらのエンジンで開発されたゲームで Scaleform を利用するに際して、コーディングはほとんど、あるいはまったく必要ありません。

この章では、カスタムの DirectX アプリケーションに Scaleform を統合する方法を説明します。DirectX ShadowVolume SDK サンプルは、標準的な DirectX アプリケーションであり、典型的なゲームに類似したアプリケーションとゲーム ループを備えています。前章で説明したように、このアプリケーションは、3D 環境と DXUT ベースの 2D UI オーバーレイをレンダリングします。

このチュートリアルでは、Scaleform をアプリケーションに統合して、デフォルトの DXUT ユーザーインターフェイスを、Flash ベースの Scaleform インターフェイスに置き換えるプロセスを説明します。

DXUT フレームワークは、基礎的な描画ループをアプリケーションに公開する代わりに、コールバックを公開して、下位レベルの詳細に抽出します。これらのステップが、標準的な Win32 DirectX 描画ループとどのように関係しているかを理解するために、Scaleform SDK に含まれている GFxPlayerTiny.cpp サンプルと比較してみてください。

4.1 Flash のレンダリング

統合処理の最初のステップは、3D の背景の上に Flash アニメーションをレンダリングすることです。これには、アプリケーションに対して、グローバルにすべての Flash コンテンツのロードの管理を行う GFx::Loader オブジェクトのインスタンス化、Flash コンテンツを含めるための GFx::MovieDef のインスタンス化、ムービーの単一の再生インスタンスを表現する GFx::Movie のインスタンス化が関わってきます。さらに Render::Renderer2D オブジェクトと Render::HAL オブジェクトがインスタンス化され、Scaleform と実装専用レンダリング API（ここでは DirectX です）との間のインターフェイスとして機能します。また、クリーンにリソースを解放する方法、喪失したデバイス イベントへの対処方法、そしてフルスクリーン/ウィンドウ モードの移行方法についても説明します。

この章の変更を反映した ShadowVolume のバージョンは、Tutorial\Section4.1 フォルダにあります。ここで使用しているコードは、あくまで説明のためであり、完全なものではありません。

ステップ#1: ヘッダー ファイルの追加

必要なヘッダー ファイルを ShadowVolume.cpp に追加します。

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

ビデオのレンダリングを行うには、複数の Scaleform オブジェクトが必要になり、アプリケーションである GFxTutorial に追加された新規のクラスに、一括してまとめられます。コードをよりクリーンにするだけではなく、Scaleform の状態をクラスにまとめておくことによって、1 回の削除呼び出しでだけで Scaleform オブジェクトをすべて解放できる利点もあります。これらのオブジェクトのやりとりを、以下のステップで詳細に説明します:

```
// アプリケーション毎に 1 個の GFx::Loader
Loader            gfxLoader;

// SWF/GFx ファイル毎に 1 個の GFx::MovieDef
Ptr<MovieDef>   pUIMovieDef;

// ムービーインスタンスの再生毎に 1 個の GFx::Movie
Ptr<Movie>  pUIMovie;

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

ステップ #2:GFx::System の初期化

Scaleform の初期化の第一段階は、GFx::System オブジェクトをインスタンス化して、Scaleform のメモリ割り当てを管理することです。WinMain で次の行を追加します:

```
// アプリケーション毎に 1 個のGFx::System
GFx::System           gfxInit;
```

この GFx::System オブジェクトは、最初の Scaleform 呼び出しの前にスコープに入る必要があります。また、アプリケーションが Scaleform の使用を終えるまでスコープから離れることはできません。これが WinMain に置かれる理由です。ここでインスタンス化された GFx::System は、Scaleform のデフォルトのメモリ アロケータを使用しますが、アプリケーションのカスタムのメモリ アロケータでオーバーライドすることもできます。このチュートリアルの場合、単に GFx::System をインスタンス化するだけで十分です。それ以上の作業は必要ありません。

GFx::System はアプリケーションが終了する前にスコープから離れる必要があります。つまり、これはグローバル変数にするべきではないということです。この場合、GFxTutorial オブジェクトが解放されるとスコープから離れます。

お使いのアプリケーションの構造によっては、GFx::System オブジェクト インスタンスを作成する代わりに、GFx::System::Init() と GFx::System::Destroy() の静的関数を呼び出す方が簡単な場合があります。

ステップ#3: ローダーとレンダラの作成

Scaleform の初期化の残りの部分は、アプリケーションの WinMain が自らの初期化を InitApp()で行った直後に行われます。以下のコードを InitApp() の呼び出し直後に追加します:

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

GFxTutorial には、GFx::Loader オブジェクトが含まれています。アプリケーションには通常、GFx::Loader オブジェクトは 1 つしかありません。このオブジェクトは SWF/GFx コンテンツをロードし、そのコンテンツをリソース ライブラリに保存して、将来の参照に備えて、リソースの再利用を可能にする役目があります。個々の SWF/GFx ファイルは、イメージやフォントなどのリソースを共有してメモリを節約することができます。また、GFx::Loader は、デバッグ ログとして使用される GFx::Log のような構成状態のセットも保持しています。

GFxTutorial::InitGFx()における最初のステップは、GFx::Loader へのステートの設定です。GFx::Loader は、デバッグ トレースを SetLog が提供するハンドラに渡します。多くの GFx 関数は、障害の理由をログに出力するため、デバッグ出力はデバッグを行うときに非常に便利です。この場合、デフォルトの Scaleform PlayerLog ハンドラを使用してメッセージをコンソール ウィンドウに出力しますが、ゲーム エンジンのデバッグ ログ システムとの統合は、GFx::Log をサブクラス化することで達成することができます。

```
// ログシステムを初期化する – GF x がエラーをログ ストリームに
// 出力します
   gfxLoader->SetLog(Ptr<Log>(*new Log()));
```

GFx::Loader は、GFx::FileOpener クラスを通してコンテンツを読み取ります。デフォルトのインプリメンテーションは、ディスク上のファイルから読み取りますが、メモリまたはリソース ファイルからのカスタム ローティングは、GFx::FileOpener をサブクラス化することで達成することができます。

```
// ローダーにデフォルトのファイル オープナーを与える
Ptr<FileOpener> pfileOpener = *new FileOpener;
            gFxLoader->SetFileOpener(pfileOpener);
```

Render::HAL は、Scaleform で様々なハードウェアにグラフィックを出力するための汎用インターフェイスです。本書ではこの D3D9 レンダラのインスタンスを作成し、それをローダーに関連付けます。そのレンダラ オブジェクトには、Scaleform で使用する D3D デバイス、テクスチャ、頂点バッファを管理する役割があります。その後、HAL::InitHAL で、

ゲームが初期化した IDirect3Device9 ポインタを Render HAL に渡し、Scaleform が DX9 リソースを作成して、UI コンテンツをレンダリングできるようにします。

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

上記のコードは、Scaleform が提供しているデフォルトの Render::D3D9::HAL オブジェクトを使用しています。Render::HAL をサブクラス化すると、Scaleform のレンダリング動作の制御を向上することができ、結果的に統合がより強固になります。

ステップ #4: Flash ムービーのロード

次に GFx::Loader でムービーをロードします。ロードされたムービーは、GFx::MovieDef オブジェクトとして表示されます。GFx::MovieDef は、ジオメトリやテクスチャを始め、ムービーの共有データのすべてをカバーします。GFx::MovieDef には、個々のボタンの状態、ActionScript 変数、現在のムービー フレームなどのインスタンスごとの情報はありません。

```
// ムービーをロードする
pUIMovieDef = *gFxLoader.CreateMovie(UIMOVIE_FILENAME
                                Loader::LoadKeepBindData |
                                Loader::LoadWaitFrame1, 0);
```

LoadKeepBindData フラグは、システム メモリにテクスチャ イメージのコピーを維持します。これはアプリケーションが D3D デバイスを再生成する際に便利です。このフラグは、家庭用ゲーム機や、テクスチャが失われることがないと分かっている状況では、不要です。

LoadWaitFrame1 は、CreateMovie に対して、ムービーの最初のフレームがロードされるまで戻らないように指示します。これは、GFx::ThreadedTaskManager を使用する場合には重要です。

最後の引数は、オプションで、使用されるメモリーアリーナを指定します。「メモリーシステムの概要（Memory System Overview）」資料を参照してメモリー アリーナの作成方法と使用法について学んでください。

ステップ #5: ムービー インスタンスの作成

ムービーをレンダリングする前に、GFx::MovieDef オブジェクトから GFx::Movie インスタンスを作成しておく必要があります。GFx::Movie には、現在のフレーム、ムービーの時間、ボタンの状態、ActionScript 変数など、ムービーの実行中の単一インスタンスに関連するステートが保存されています。

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

CreateInstance の引数で、最初のフレームを初期化するかどうかが決まります。その引数が false の場合、ActionScript の最初のフレームの初期化コードが実行される前に、Flash と ActionScript のステートを変更するチャンスができます。最後の引数は、オプションで、使用されるメモリーアリーナを指定します。「メモリーシステムの概要（Memory System Overview）」資料を参照してメモリー アリーナの作成方法と使用法について学んでください。

いったんムービー インスタンスが作成されると、Advance()を呼び出して最初のフレームを初期化します。これは、false が CreateInstance に渡されたときに限って必要です。

```
// ムービーを最初のフレームに Advance します
pUIMovie->Advance(0.0f, 0, true);

// time は、このフレームから次のフレームまでの時間経過を
// 決定するものです
MovieLastTime = timeGetTime();
```

Advance の最初の引数は、ムービーの最後のフレームとこのフレーム間の**時間差**（単位は、秒）です。現在のシステム時間が記録され、このフレームと次のフレーム間の時間差の計算を可能にします。

3D シーンの上にムービーをアルファ ブレンドするには、コードを以下のようにします:

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

上記の呼び出しがないと、ムービーはレンダリングを行いますが、Flash ファイルで指定した背景ステージカラーでその 3D 環境を覆ってしまいます。

ステップ #6: デバイスの初期化

レンダリングを行うには、DirectX デバイスと、プレゼンテーション パラメータのハンドルを Scaleform に与える必要があります。これらの値は、
Render::D3D9::HALInitParams
ストラクチャー内に抱合される、Render::D3D9::HAL::InitHAL 関数を渡される、D3D デバイス作成後と Scaleform がレンダリングを命令される前に呼び出す、などの値に一致します。
ウィンドウのサイズを変更したり、フルスクリーン/ウィンドウ モードの移行があると、D3D デバイス ハンドルの変更が生じるため、そのときは、あらためて InitHAL を呼び出す必要があります。

最初のデバイスの作成後やデバイス リセット後は、DXUT フレームワークによって ShadowVolume の OnResetDevice 関数が呼び出されます。以下のコードが、GFxTutorial の該当する OnResetDevice メソッドに追加されます:

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

InitHAL() の呼び出しで、D3D デバイス情報が Scaleform に渡されます。HAL_NoSceneCalls フラグは、DirectX BeginScene() end EndScene() 呼び出しが、Scaleform によって行われないことを指定しています。この ShadowVolume サンプルは、すでにこれらの呼び出しを、OnFrameRender コールバックでアプリケーションのために行っているため、この指定が必要になります。

ステップ #7: デバイスの喪失

ウィンドウのサイズが変更されたり、アプリケーションがフルスクリーンに切り替えられると、D3D デバイスは失われます。頂点バッファやテクスチャを含む D3D サーフェスはすべて、再初期化する必要があります。ShadowVolume は、OnLostDevice コールバックでサーフェスを解除します。Render::HAL には、喪失したデバイスに関する情報が通知され、GFxTutorial の該当する OnLostDevice メソッドで、自らの D3D リソースを解放するチャンスが与えられます。

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

このステップと前のステップで、DXUT フレームワークのコールバック システムに基づいた初期化と喪失したデバイスについて説明を行いました。基本的な Win32/DirectX 描画ループの例については、Scaleform SDK にある Scaleform PlayerTiny.cpp の例を参照してください。

ステップ #8:リソースの割り当てとクリーンアップ

Scaleform オブジェクトはすべて ScaleformTutorial オブジェクトに含まれているため、クリーンアップは、WinMain の最後で ScaleformTutorial オブジェクトを削除するという簡単なことです。

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

それ以外の考慮事項としては、頂点バッファのような DirectX 9 リソースのクリーンアップがあります。これは Scaleform によって行われますが、メインのゲーム ループ中に発生する割り当てとクリーンアップに関しては、InitHAL() と ShutdownHAL() が果たす役割を理解することが重要になります。

DirectX 9 のインプリメンテーションでは、InitHAL が頂点バッファを含む D3DPOOL_DEFAULT リソースの割り当てを行います。独自のエンジンと統合するときは、D3DPOOL_DEFAULT リソースの割り当てに最適の場所で、InitHAL を呼び出すようにしてください。

ShutdownHAL は D3DPOOL_DEFAULT リソースを解放します。ShadowVolume を含め、DXUT フレームワークを使用するアプリケーションは、D3DPOOL_DEFAULT リソースを DXUT OnResetDevice コールバックで割り当て、リソースを OnLostDevice コールバックで解放する必要があります。GFxTutorial::OnLostDevice の ShutdownHAL 呼び出しは、GFxTutorial::OnResetDevice の InitHAL 呼び出しに一致します。

独自のエンジンと統合するときは、他の呼び出しと一緒に InitHAL と ShutdownHAL を呼び出すようにして、エンジン リソースである D3DPOOL_DEFAULT を作成し解放します。

ステップ #9: ビューポートの設定

ムービーには、画面内でレンダリング先となる一定のビューポートを与えてやらなければなりません。この場合、ビューポートはウィンドウ全体を占有します。画面の解像度は変化する可能性があるため、D3D デバイスがリセットされるたびに、以下のコードを GFxTutorial::OnResetDevice に追加してビューポートをリセットします:

```
// クライアント ウィンドウの矩形サイズをビューポートとして使います
RECT  windowRect   = DXUTGetWindowClientRect();
DWORD windowWidth  = windowRect.right - windowRect.left;
DWORD windowHeight = windowRect.bottom - windowRect.top;
pUIMovie->SetViewport(windowWidth, windowHeight, 0, 0,
                                windowWidth, windowHeight);
```

SetViewport の最初の 2 つのパラメータで、使用するフレームバッファのサイズを指定します。これは通常、PC アプリケーションのウィンドウ サイズになります。さらに次の 4 つのパラメータで、フレームバッファ内の GFx がレンダリングされるビューポートのサイズを指定します。

このフレームバッファ サイズの引数は、OpenGL やその他のプラットフォームとの互換性のために提供されています。座標系の向きが異なっていたり、フレームバッファ サイズの照会手段がない場合が考えられるからです。

Scaleform には、ビューポート内の Flash コンテンツのスケーリングや、配置方法を制御する関数が用意されています。これらのオプションについては、アプリケーションが実行可能になった後、第 4.2 章で検証します。

ステップ #10: DirectX シーンへのレンダリング

レンダリングは、ShadowVolume の OnFrameRender() 関数で実行されます。D3D レンダリング呼び出しはすべて、BeginScene() 呼び出しと EndScene() 呼び出しの間で行われます。EndScene()を 呼び出す前に、GFxTutorial::AdvanceAndRender() を呼び出します。

```
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 で deltaTime (単位は秒) だけムービーを進めます。ムービーの再生速度は、現在のシステム時間をベースに、アプリケーションによってコントロールされます。GFx::Movie::Advance には、実際のシステム時間を提供することが重要です。それによって、ハードウェア構成が変わっても正確にムービーが再生されます。

ステップ #11: レンダリング状態の保存

Render::Renderer2D::Display は DirectX 呼び出しを実行して、D3D デバイス上でムービー フレームのレンダリングを行います。パフォーマンス上の理由で、ブレンド モード、テクスチャ保存設定などの各種 D3D デバイス ステートは保存されず、Render::Renderer2D::Display への呼び出し後は、D3D デバイスのステートは変わってしまいます。アプリケーションによっては、これによって不都合が生じます。最も直接的な解決方法は、Display への呼び出し前にデバイス ステートを保存しておいて、その後復元することです。Scaleform レンダリング後、同じゲーム エンジンに、必要なステートを再初期化させれば、パフォーマンスをさらに向上できます。このチュートリアルでは、DX9 のステート ブロック関数を使用して、単純にステートを保存し復元します。

DX9 のステート ブロックは、そのアプリケーションを終了するまで割り当てられており、GFxTutorial::AdvanceAndRender() への呼び出し前も呼び出し後も使用できます。

```
// GF x 呼び出しの前に DirectX ステートを保存する
g_pStateBlock->Capture();

// フレームをレンダリングしてタイム カウンターを進める
gFx->AdvanceAndRender();

// DirectX ステートを復元してゲームのレンダリング ステートに影響を与えないようにする
g_pStateBlock->Apply();
```

ステップ #12: デフォルト UI の無効化

最後にオリジナルの DXUT ベースの UI をオフにします。これは、コードの関連ブロックをコメント アウトすることで行います。最終結果は、Section4.1\ShadowVolume.cpp にあります。diff を使って前章のコードとの差分をとると、その違いが分かります。DXUT に関連する変更にはすべてコメントがつきます。

```
// デフォルト UI を無効化する
...
```

これで、DirectX アプリケーション上で、ハードウェア加速された Flash ムービーがレンダリングされるようになりました。

図 3: GFx Flash ベースの UI を使用した ShadowVolume アプリケーション

4.2 スケーリング モード

GFx::Movie::SetViewport を呼び出すことによって、ビューポートの寸法と画面の解像度が同じになります。画面のアスペクト比が Flash コンテンツの元のアスペクト比と異なる場合は、インターフェイスに歪みが生じる場合があります。Scaleform には、以下の機能があります:

- コンテンツのアスペクト比を保ち、自由に拡大できる。
- ビューポートの中心、コーナーあるいは側面を基準にしてコンテンツを配置する。

これらの機能は、4:3 画面とワイド画面の両方で同じコンテンツをレンダリングするのに非常に便利です。Scaleform の利点の 1 つは、スケーラブル ベクター グラフィックによって、画面の解像度に合わせてコンテンツのサイズを自由に変更できる点にあります。従来のビットマップ グラフィックでは、さまざまな画面の解像度に合わせて、大きなビットマップと小さなビットマップの両方 (800x600 の低解像度用と、1600x1200 の高解像度用など) を作成しなければなりません。Scaleform では、同じコンテンツをどのような解像度にでもスケーリングすることができます。さらに従来のビットマップ グラフィックについても、ビットマップがより適しているようなゲームの構成要素に対して、完全にサポートされています。

GFx::Movie::SetViewScaleMode は、スケーリングの方法を定義します。元のアスペクト比に影響を与えずにムービーをビューポート内に収めるために、GFxTutorial::InitGFx() の最後に、他の呼び出しとともに下記のコードを追加して、GFx::Movie オブジェクトを設定することができます。

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

SetViewScaleMode の 4 つの可能な引数は以下のとおりで、オンライン ドキュメントにも説明があります:

SM_NoScale	コンテンツのサイズは、Flash ステージの本来の解像度に固定されます。
SM_ShowAll	元のアスペクト比を維持しつつ、ビューポートに合うようにコンテンツをスケーリングします。
SM_ExactFit	元のアスペクト比に関係なく、ビューポート全体を満たすようにコンテンツをスケーリングします。ビューポート全体に表示されますが、歪みが生じる可能性があります。
SM_NoBorder	元のアスペクト比を維持しつつ、ビューポート全体を満たすようにコンテンツをスケーリングします。ビューポート全体に表示されますが、クリッピングが生じる可能性があります。

補完的な SetViewAlignment でビューポートを基準にコンテンツの位置を制御します。アスペクト比が維持されると、SM_NoScale または SM_ShowAll の選択時に、ビューポートの一部が空白になることがあります。SetViewAlignment でビューポート内のどこにコンテンツを配置するかが決まります。ここでは、インターフェイス ボタン類は、画面の右端に縦方向にセンタリングさせてください。

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

SetViewScaleMode と SetViewAlignment の引数を変えて、ウィンドウのサイズ変更時にアプリケーションの動作がどのように変化するかを確認してみてください。

SetViewAlignment 関数は、SetViewScaleMode を SM_NoScale のデフォルト値に設定したとき以外は何の効果もありません。より複雑なアライメント、スケーリング、配置の必要条件に関しては、Scaleform は ActionScript 拡張機能をサポートしており、それによって、ムービーは自分のサイズと位置を選択することができます。サンプルの ActionScript コードについては、d3d9guide.fla を参照してください。

スケールとアライメントのパラメータは、C++の代わりに ActionScript から設定することも可能です。SetViewScaleMode と SetViewAlignment は、ActionScript Stage クラス (State.scaleMode、Stage.align) によって表される同じプロパティを修正します。

4.3 入力イベントの処理

ShadowVolume のレンダリング パイプラインは、Scaleform で Flash をレンダリングできるように修正されたので、次は再生する Flash とのやりとりです。たとえば、ボタンの上をマウスが移動することで、ボタンがハイライト表示されたり、テキスト ボックスに入力すると、新しい文字が表示されるようにしなければならないとします。

GFx::Movie::HandleEvent は、入力したキーやマウスの座標などのイベント タイプやその他の情報を表す GFx::Event オブジェクトを渡します。アプリケーションは、入力に基づいたイベントを構築し、それを適切な GFx::Movie に渡すだけです。

4.3.1マウス イベント

ShadowVolume は、MsgProc コールバックで Win32 入力イベントを受け取ります。呼び出しが GFxTutorial::ProcessEvent に追加され、Scaleform でイベントを処理するコードを実行します。下記のコードによって、WM_MOUSEMOVE、WM_LBUTTONDOWN、WM_LBUTTONUP が処理されます。

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

Scaleform は、マウス座標が指定されたビューポートの左上隅を基準にすると想定します。ムービーの本来の解像度ではありません。このことは以下の例で明らかになります:

例 #1: ビューポートと画面のサイズの一致

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

この場合、変換は不要です。ムービーは (0, 0) に配置されているため、Windows のマウス座標はすでにムービーの左上隅が基準になっています。この座標は、内部処理のために、Scaleform によってビューポート サイズから本来のムービー解像度へ内部でスケーリングされます。

例 #2: ビューポートは画面より小さいが、画面の左上隅に配置されている

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

この場合も変換は不要です。ビューポートはスケール ダウンされているため、ボタンのサイズと位置は変わります。ただし、HandleEvent で使用される座標も、Windows 画面の座標もいまだにウィンドウの左上隅が基準になっており、変換は不要です。ビューポートのサイズから本来のムービー解像度への座標のスケーリングは、Scaleform によって内部処理されます。

例 #3: ビューポートが画面より小さく、画面センターにある

```
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);
```

この場合、Windows 画面の座標の変換が必要です。ムービーはすでに (0, 0) の位置にはないため、(screen_width / 2 – movie_width / 2, screen_height / 2 – movie_height / 2) の新しい位置を、Windows によって渡された画面の座標から差し引く必要があります。

Flash コンテンツがセンタリングされているか、他の何らかの方法で GFx::Movie::SetViewAlignment によってアラインされている場合、これらの変換は不要です。マウス座標が GFx::Movie::SetViewport に与えられた座標を基準としている限り、SetViewAlignment および SetViewScaleMode によるアライメントとスケーリングは、Scaleform によって内部処理されます。

4.3.2キーボード イベント

キーボード イベントも GFx::Movie::HandleEvent によって処理されます。キー イベントには、GFx::KeyEvent と GFx::CharEvent の 2 種類があります。

```
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 は、未加工のスキャン コードに似ており、GFx::CharEvent は、加工された ASCII 文字に似ています。Windows では、GFx::CharEvent は、WM_CHAR メッセージに応じる形で生成

されます。それに対して、GFx::KeyEvents は、WM_SYSKEYDOWN、WM_SYSKEYUP、WM_KEYDOWN、WM_KEYUP メッセージに応えて生成されます。次にこれらの例をいくつか紹介します:

- Shift キーを押したままの状態で、c キーを押します。GFx::KeyEvent が WM_KEYDOWN メッセージに応じる形で生成されますが、それは以下のことを示す必要があります:
  - c キーが押されたので、そのキーのスキャン コード
  - そのキーは押されたままになっている
  - Shift キーがアクティブな状態である

- 同時に WM_CHAR メッセージに対応する形で GFx::CharEvent が起動し、「処理された」ASCII 値の 'C' を Scaleform に渡す必要があります。

- いったん c キーが解放されると、WM_KEYUP メッセージに対応して、GFx::KeyEvent を送出する必要があります。キーが解放されたときには、GFx::CharEvent を送出する必要はありません。

- F5 キーが押されます。キーが押されたとき、GFx::KeyEvent が送出され、キーが元に戻ると、2 つめのイベントが送出されます。F5 は、出力可能な ASCII コードには対応していないため、GFx::CharEvent を送出する必要はありません。

キーを押すイベントとキーを離すイベントに対して、個別の GFx::KeyEvent が送出されます。プラットフォームに左右されないように、キー コードは GFx_Event.h に定義されており、Flash で内部的に使用されているキー コードと一致するようになっています。GFxPlayerTiny.cpp サンプルと Scaleform Player プログラムにはいずれも、Windows のスキャン コードを対応する Flash コードに変換するコードが含まれています。この章の最後のコードには、ProcessKeyEvent 関数が含まれており、カスタム 3D エンジンとの統合を行う場合は、それを再利用することができます。

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

WM_CHAR、WM_SYSKEYDOWN、WM_SYSKEYUP、WM_KEYDOWN、WM_KEYUP のメッセージに対して、Windows WndProc 関数からこの関数を呼び出すだけです。最適な Scaleform イベントが生成され、pMovie に送出されます。

GFx::KeyEvents と GFx::CharEvents の両方を送出することが重要です。たとえば、多くのテキスト ボックスは、出力可能な ASCII 文字を対象としているため、GFx::CharEvents にしか応答しません。また、テキスト ボックスは、Unicode 文字 (漢字の入力方式エディタ [IME] など) にも対応できなければなりません。IME 入力に際しては、本来のキー コードは役に立たず、最終的な文字イベント (通常、IME で処理された複数のキー ストロークの結果) のみがテキスト ボックスで使用可能になります。それに対して、リスト ボックスでは、Page Up および Page Down キーの使用は、GFx::KeyEvent を通して受付けないように制御する必要があります。これらのキーは、出力可能な文字に対応していないためです。

この関数はこの章の最後のコードとともに、Tutorial\Section4.3 フォルダに収められています。このプログラムを実行し、マウスをボタン上に移動してみてください。ボタンが正常にハイライト表示さ

れ、3D エンジンとの統合を必要としないボタンは正常に動作します。[Settings] (設定) を押すと、C++コードなしで DX9 構成画面に移行します。これは、d3d9guide.fla が ActionScript を使用してこの簡単なロジックを実装しているためです。

動作中のキーボード処理コードを確認するには、[Change Mesh] (メッシュの変更) をクリックし、テキスト入力ボックスに文字を入力します。いくつかの小さな問題点はありますが、それはこのチュートリアルの後半で修正します。[Change Mesh] (メッシュの変更) ボタンを押してテキスト入力ボックスを開いたとき、表示されるアニメーションに注意してください。このアニメーションは、ベクター グラフィックを使用して Flash で作成するのは容易ですが、従来のビットマップをベースにしたインターフェイスでは、実用的ではありません。このアニメーションには、ビットマップを追加するだけではなく、カスタム コードが必要になるため、読み込みに時間がかかる可能性があり、レンダリングにもコストがかかり、なによりもコーディングが大変です。

4.3.3ヒット テスト

アプリケーションを実行し、左のマウス ボタンを押したままマウスを動かして、3D の世界でカメラの方向を変えてみます。ここでマウスを UI エレメントの 1 つに移動して、同じことをしてみます。マウスのクリックによって、インターフェイス上で希望どおりに反応しますが、カメラが相変わらず動いてしまいます。

UI と 3D の世界との間のフォーカス制御が問題を起こしているのです。これは GFx::Movie::HitTest が解決します。この関数は、ビューポート座標が、Flash コンテンツでレンダリングされたエレメントにヒットするかどうかを判断します。マウス イベントの処理後、GFxTutorial::ProcessEvent を修正して、HitTest を呼び出します。UI エレメント上でイベントが発生したら、DXUT フレームワークに信号が通知され、そのイベントを、カメラに渡さないようにします。

```
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キーボード フォーカス

アプリケーションを実行して、[Change Mesh] (メッシュの変更) ボタンをクリックし、テキスト入力ボックスを開きます。テキスト ボックスに入力すると、4.3.2 章で追加されたコードにより、キーボード入力が機能します。ただし、W、S、A、D、Q、E のキーを押すと文字は入力されますが、同時に 3D の世界でカメラも移動してしまいます。このキーボード イベントは Scaleform で処理されますが、同時に 3D カメラにもそれらの情報が渡ってしまいます。

この問題を解決するには、テキスト入力ボックスにフォーカスが適用されているかどうかを判断する必要があります。6.1.2 章では、Flash ActionScript を使用してイベントを C++に送出し、イベント ハンドラにフォーカスを追跡させる方法について説明します。

4.3.5タッチイベント

タッチイベントはマウスイベントと似ていますが、ユーザーが画面を触っているときにだけ出される点が異なります。タッチイベントを利用するには、生のタッチデータをWindows APIから直接Scaleformにルーティングします。プラットフォームの最低要件はタッチ対応のウィンドウズ7です。これなしではこのチュートリアルを行えませんので、スキップして次のセクションに進んでください。

コンパイル前に、プリプロセッサの定義に**WINVER=0x601**を追加してください。これが**Windows API**に **Windows 7**に特有の機能（マルチタッチ）をコンパイルさせます。ShadowVolume.cppで、サポートされていないコードがコンパイルされないように保護するマクロである#define ENABLE_MULTITOUCHをコメント解除します。

特別なマルチタッチライブラリを使用するために、#include <windows.h>が追加されました。ウィンドウズの初期化中、**RegisterTouchWindow**で、タッチイベントをリッスンする意図を登録します。これは**MsgProc**コールバックで**WM_TOUCH**を受信して、処理できるようにします。

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

ProcessTouchEvent は、**WM_TOUCH** メッセージをパーシングし、リレーする関数です。タッチの座標をピクセル座標に変換するにはウィンドウズの機能を使用します。変換後、**GFx::TouchEvent** が生成されて、**HandleEvent** 経由でムービーに送信されます。

GFx::TouchEvent クラスは、ユニークな ID(OS が管理）、x/y 座標、接触エリア、圧力（0～1）、プライマリポイントかどうかで構成されています。

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

これで **SWF** ムービーがタッチイベントを受け取りますが、**GFx** は既定では最大 0 のタッチポイントをサポートしています。最後のステップは継承されたマルチタッチ環境、具体的にはサポートされる最大タッチポイント数と、**GFx::GestureEvent** をリレーするかどうかを記述する **MultitouchInterface** を定義することです（このチュートリアルではネイティブなジェスチャーイベントは取り扱いませんが、このセクションで取り上げるものと同じ原則で **Scaleform** がこれらを処理できるようにできます）。

```
class FxPlayerMultitouchInterface :public MultitouchInterface
{
public:
   // ハードウェアがサポートする最大数のタッチポイントを返す
   virtual unsigned GetMaxTouchPoints() const { return 2; }

   // サポートされるジェスチャーのビットマスクを返す（現在のところは無し）
   virtual UInt32   GetSupportedGesturesMask() const { return 0; }

   // マルチタッチがサポートされているか？
   virtual bool     SetMultitouchInputMode(MultitouchInputMode) { return true; }
};
```

このインターフェイスクラスの参照を **SetMultitouchInterface** 経由でムービーに送ると、タッチイベントがムービーに伝達されます。

5 ローカライズとフォントについて

この章について詳しくは、 Font and Text Configuration Overview(日本語版) を参照して下さい。

5.1 フォントの概要と機能

Scaleform は、効率的で自在なフォントとローカライズのシステムを備えています。複数のフォント、ポイント サイズ、スタイルを、メモリの使用量を抑えて、同時に効率良くレンダリングすることができます。フォント データは、Flash の埋め込みフォント、共有フォント ライブラリ、オペレーティング システム、および TTF フォント ライブラリから直接得ることができます。ベクターベースのフォントの圧縮によって、大型のアジア フォントの大きいメモリ使用量を削減します。フォントのサポートは完全にクロス プラットフォームであり、家庭用ゲーム システム、Windows、Linux で同様に動作します。Scaleform のフォントと多言語化機能について詳しくは、次の Developer Center の Documentation ページを参照してください:
http://gameware.autodesk.com/scaleform/developer/?action=doc&lang=jp

一般的なフォント ソリューションには、フォント全体のそれぞれの文字をテクスチャにレンダリングすることが含まれます。次に、文字は必要に応じてそのフォント テクスチャから画面にマッピングされたテクスチャとなります。フォント サイズやスタイルが変わると、別のテクスチャが必要になります。ラテンの文字については、メモリの使用量は許容範囲ですが、5000 個のグリフを持つアジア フォントをテクスチャにレンダリングするのは、実用的ではありません。大量の貴重なテクスチャ メモリと、各グリフをレンダリングする処理時間が必要になります。さまざまなフォント サイズとスタイルを同時にレンダリングすることは、問題外です。

Scaleform はこの問題を、ダイナミック フォント キャッシュで解決します。文字は必要に応じてキャッシュにレンダリングされ、キャッシュのスロットは必要なときに置き換えられます。さまざまなフォント サイズとスタイルが、Scaleform のインテリジェント グリフパッキング アルゴリズムを使って、1 つのパブリック キャッシュを共有することができます。Scaleform はベクター フォントで動作します。つまり、メモリに TTF フォントを一書体だけ保管して、どのサイズでも文字を鮮明に表示します。最後に、Scaleform は「擬似イタリック」や「擬似ボールド」の機能をサポートし、レギュラー フォント一書体のベクター表示を、必要に応じてイタリックやボールドに変形し、メモリを節約することができます。

大量のテキストも、テッセレートしたトライアングルとして、直接レンダリングすることができます。これは、ゲームのタイトル画面に見られるような、少ない文字数の大きなテキストにも便利です。Bin\Data\AS2\Samples\FontConfig フォルダのフォントの構成サンプルを探して、sample.swf ファイルを開いている GFxPlayer D3D9 ウィンドウにドラッグします。

おはようございます。
お元気ですか？
さみしかったです。

図 4a:文字　　**図 4b:** ワイヤーフレーム表示

Ctrl＋W を押して、これらの文字のワイヤーフレーム表示を見ると、それぞれの文字がテクスチャにマップされた 2 個のトライアングルとして表現されているのがわかります。これらの文字は量が少ないので、ダイナミック フォント キャッシュで、ビットマップとして表示する方が効率が良くなります。

ワイヤーフレーム モードのときに、ウィンドウを最大化してみます。文字はテクスチャ マップの表示から、無色のトライアングルとしての表示に切り替わります。

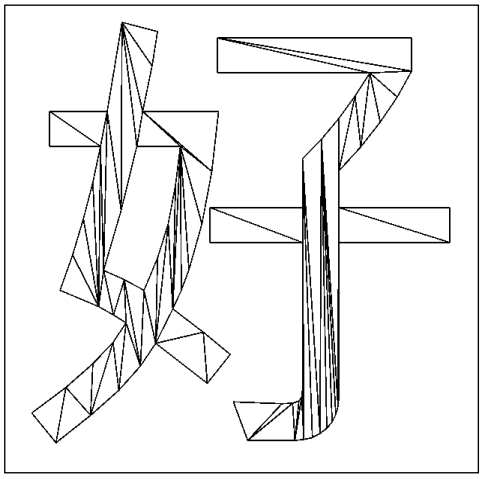

図 4c: 大きな漢字　　**図 4d:** ワイヤーフレーム表示

大きな文字については、無色のトライアングルとして表示する方が効率的です。大きなビットマップで作業することは、過度のメモリ容量を使用するため不経済です。カラーを必要とするピクセルだけを設定して、その文字の空白領域上の処理能力の浪費を防ぎます。図 4c と 4d では、Scaleform は文字サイズがある特定のしきい値を超えたことを検知して、その文字をトライアングルにテッセレートし、ビットマップとしてではなく、幾何学的に文字を表現しています。

フォントのソフト シャドウ、ぼかし、その他の効果もサポートされています。Flash のテキスト フィールドで該当するフィルタを設定するだけで、必要な効果が生成されます。詳細は、上記のリンクの「Font Overview(Japanese)」を参照してください。Developer Center の GFx_2.1_texteffects_sample.zip をダウンロードすると、サンプルを得ることができます。

図 5: テキストの視覚効果

6 C++、Flash、ActionScript の相互作用

Flash の ActionScript スクリプト言語は、インタラクティブなムービー コンテンツの作成を可能にします。ボタンをクリックする、特定のフレームに到達する、ムービーをロードするなどのイベントは、ダイナミックなムービーのコンテンツの変更、ムービーの流れの制御、さらに追加ムービーの起動などのコードを実行することができます。ActionScript は、ミニゲームの完全版を Flash だけで作るには十分です。ほとんどのプログラミング言語のように、ActionScript は変数とサブルーチンをサポートしています。Scaleform は C++インターフェイスを提供して、ActionScript 変数と配列を直接操作し、ActionScript サブルーチンを直接呼び出します。また、Scaleform は ActionScript が C++プログラムにイベントを渡し、データを戻すことができるコールバック メカニズムも提供します。

6.1 ActionScript から C++へ

Scaleform は C++アプリケーションに FSCommand と ExternalInterface の 2 つのメカニズムを提供して、ActionScript からイベントを受け取ります。FSCommand と ExternalInterface は両方とも、C++のイベント ハンドラを GFx::Loader に登録して、イベント通知を受け取ります。FSCommand のイベントは、ActionScript の fscommand 関数でトリガされ、2 つの文字列引数を受け取りますが、値を戻すことはできません。ExternalInterface のイベントは、ActionScript が flash.external.ExternalInterface.call 関数を呼び出したときにトリガされ、入力されたさまざまな GFx::Value 引数 (6.1.2 章で説明します) のリストを受け取り、呼び出し元に値を戻すことができます。

柔軟性が限られているため、FSCommand は ExternalInterface に取って代わられ、使用はあまりお勧めできません。完全を期すためと、レガシー コードには FSCommand が含まれている可能性があるため、本書では FSCommand について説明しています。さらに、GFxExport は–fstree、–fslist、–fsparams オプションを使って、SWF ファイルで使用されるすべての FSCommand のレポートを生成することができます。これは ExternalInterface では不可能であり、場合によっては、このことが FSCommand を使用する十分な理由になりえます。

6.1.1 FSCommand のコールバック

ActionScript の fscommand 関数は、コマンドとデータ引数をホスト アプリケーションに渡します。ActionScript における通常の使用は、次のようなものです:

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

ブーリアンや整数などの fscommand への非文字列引数は、文字列に変換されます。ExternalInterface は直接、整数引数を受け取ることができます。

この関数は 2 つの文字列を GFx FSCommand ハンドラに渡します。アプリケーションは GFx::FSCommandHandler をサブクラス化し、そのクラスを GFx::Loader か、または個別の GFx::Movie オブジェクトのいずれかに登録して、fscommand ハンドラを登録します。コマンド ハンドラが GFx::Movie に設定されている場合、そのムービー インスタンスで行われる fscommand 呼び出しのためだけにコールバックを受け取ります。GFxPlayerTiny のサンプルはこのプロセス (“FxPlayerFSCommandHandler” の検索) を表しており、ここでは ShadowVolume に類似コードを追加します。この章の最後のコードは Tutorial\Section6.1 フォルダにあります。

まず、GFx FSCommandHandler をサブクラス化します。

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

このコールバック メソッドは、ActionScript の fscommand に渡される 2 つの文字列引数と、fscommand を呼び出した特定のムービー インスタンスのポインタを受け取ります。

次に、GFxTutorial::InitGFx()で GFx::Loader オブジェクトを作成した後で、このハンドラを登録します。

```
// FSCommand ハンドラを登録する
Ptr<FSCommandHandler> pcommandHandler = *new OurFSCommandHandler;
gFxLoader->SetFSCommandHandler(pcommandHandler);
```

このハンドラを GFx::Loader に登録すると、すべての GFx::Movie がこのハンドラを継承することになります。SetFSCommandHandler を個々のムービー インスタンスで呼び出して、このデフォルト設定をオーバーライドすることができます。

このカスタム ハンドラは、各 fscommand イベントを単純にデバッグ コンソールに出力します。ShadowVolume を起動して、[Toggle Fullscreen] (フルスクリーンに切り替え) ボタンをクリックします。UI イベントが発生するときは常に、イベントが出力されています。

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

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

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

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

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

練習として、fscommand(“ToggleFullscreen”) を fscommand(“ToggleFullscreen”, hud.visible) に変更して、hud.visible.value 値を C++に出力します。この Flash ムービーを書き出して (Ctrl+Alt+Shift+S キー) d3d9guide.swf を置き換えます。

UI のボタンは、FSCommand イベントが発生するときに、該当するコードをトリガすることで、ShadowVolume サンプルに統合することができます。例として、以下のラインを fscommand ハンドラに追加して、フルスクリーン モードを切り替えます:

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

DXUT 関数 OnFrameMove が、フレームがレンダリングされる直前に呼び出されます。OnFrameMove コールバックの最後に、以下の対応するコードを追加します:

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

一般的に、イベント ハンドラはブロック不可であり、可能な限り早く呼び出し元に戻る必要があります。イベント ハンドラは通常、Advance または Invoke 呼び出し中に限って、呼び出されます。

6.1.2ExternalInterface

Flash の ExternalInterface.call メソッドは、fscommand に似ていますが、より自在に引数処理を行い、値を戻すことができるため、好まれています。

ExternalInterface ハンドラの登録は、fscommand ハンドラの登録に似ています。

```
class OurExternalInterfaceHandler : public ExternalInterface
{
  public:
      virtual void Callback(Movie* pmovieView,
```

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

このハンドラを GFx::Loader に登録します。

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

外部のインターフェイス呼び出しは、テキスト入力ボックスにフォーカスが適用されたとき、またはそこからフォーカスが移動したときに、ActionScript から行われます。Open d3d9HUD.as used by d3d9guideAS3.fla and look at the ActionScript for the following functions:

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

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

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

MeshPathFocus コールバックは、テキスト入力ボックスにフォーカスが適用されたとき、

またはそこからフォーカスが移動したときは常に呼び出されます。

ExternalInterface 呼び出しは、この ExternalInterface ハンドラをトリガして、テキスト入力ボックスのフォーカス状態 (true または false) と任意のコマンド文字列 “MeshPathFocus” を渡します。テキスト入力を開き、テキスト領域をクリックし、さらに画面の空白の領域をクリックして、テキスト入力からフォーカスを移動します。コンソールには以下のように出力されるはずです:

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

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

このイベント ハンドラは、フォーカスが適用されたとき、またはフォーカスが移動したときを検出し、その情報を GFxTutorial オブジェクトに渡すように修正することができます。

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

テキスト ボックスにフォーカスが適用されている場合、GFxTutorial::ProcessEvent はキーボード イベントをムービーに渡すだけです。キーボード イベントがテキスト ボックスに渡ると、そのイベントが 3D エンジンに渡るのを防ぐために、フラグが設定されます。

```
            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;
               }
            }
```

スペース (ASCII code 32) とタブ (ASCII code 9) は、[Toggle UI] (UI の切り替え) ボタンと [Settings] (設定) ボタンに対応しているので、常に渡されます。

ユーザーがレンダリングされているメッシュを変更できるように、[Change Mesh] (メッシュの変更) ボタンをクリックして、テキスト ボックスを開き、新規のメッシュ名を入力して Enter キーを押します。Enter キーを押すと、そのテキスト ボックスは ActionScript イベント ハンドラを呼び出します。このハンドラは、この新規のメッシュ名で ExternalInterface を呼び出します。OurExternalInterfaceHandler::Callback の追加コードは以下のとおりです:

```
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));
}
```

フルスクリーン切り替えと同様に、実際の作業は DXUT OnFrameMove コールバックで行われます。このコードは、デフォルトの DXUT インターフェイスのイベント ハンドラに基づいています。

6.2 C++から ActionScript へ

第 6.1 章では ActionScript を C++に呼び出す方法を説明しました。この章では、C++プログラムに、再生しているムービーとの通信を開始させる Scaleform 関数を使った、逆方向の通信方法を説明します。Scaleform は C++関数をサポートして、ActionScript サブルーチンを呼び出す他に、ActionScript 変数を取得し設定します。

6.2.1ActionScript 変数の操作

Scaleform は GetVariable と SetVariable をサポートします。この 2 つは ActionScript 変数を直接操作することができます。Tutorial\Section6.2 フォルダのコードは修正され、Scaleform Player プログラムが使用する黄色の HUD ディスプレイ (fxplayer.swf) を読み込み、F5 キーを押すたびにカウンタを増やすようになりました。

```
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 は_root.counter 変数の値を戻します。最初はこの変数は存在せず、GetVariableDouble はゼロを戻します。カウンタは増え、SetVariable を使ってこの新規の値が_root.counter に保存されます。GFx::Movie のオンライン ドキュメントは、GetVariable と SetVariable のさまざまなバリエーションを掲載しています。

この fxplayer Flash ファイルには、アプリケーションが任意のテキストに設定できる、2 つの動的テキスト フィールドが含まれています。MessageText テキスト フィールドは、画面の中央に位置して、HUDText 変数は、画面の左上隅に配置されます。文字列は_root.counter 変数の値に基づいて生成され、SetVariable を使ってこのメッセージ テキストが更新されます。

パフォーマンスの注意点: この Flash の動的なテキスト フィールドの値を変更する好ましいメソッドは、TextField.text 変数を設定する、または GFx::Movie::Invoke を呼び出して、ActionScript ルーチンを実行し、そのテキストを変更することです (詳細は 6.2.2 章を参照してください)。動的なテキスト フィールドを任意の変数にバインドし、次にその変数を変えてテキストを変更**しないでください**。これは有効ですが、Scaleform はすべてのフレームでその変数の値をチェックしなければならないので、パフォーマンス ペナルティを受けます。

図 8: HUD テキストの値を変更する SetVariable

上記の使用法は、変数を直接、文字列または数字として処理します。オンライン ドキュメントは、GFx::Value オブジェクトを使って、変数を直接、整数として効果的に処理し、文字列から整数への変換の必要性を解消する別のシンタックスについて説明しています。

SetVariable は、割り当てを「sticky (固定)」と宣言する GFx::Movie::SetVarType タイプの 3 つ目の引数をオプションで備えています。これは、割り当てられている変数が、その Flash タイムラインにまだ作成されていないときに便利です。たとえば、テキスト フィールド_root.mytextfield が、ムービーのフレーム 3 まで作成されていないと仮定します。ムービーの作成直後、SetVariable("_root.mytextfield.text", "testing", SV_Normal) がフレーム 1 で呼び出される場合、この割り当ては何の効果もありません。この呼び出しが SV_Sticky (デフォルト値) で行われる場合、このリクエストは列に追加され、 _root.mytextfield.text 値がフレーム 3 で有効になったときに、適用されます。これは C++からのムービーの初期化をさらに容易にします。一般的に、SV_Normal は SV_Sticky よりも効率が良いので、可能な場合は SV_Normal を使用する必要があります。

SetVariableArray は変数の配列全体を一回の操作で Flash に渡します。この関数は、ダイナミックにドロップダウン リスト制御を設定するなどの操作に利用することができます。以下のコードは GFxTutorial::InitGFx()に置かれ、_root.SceneData ドロップダウンの値を設定します。

```
// シーン ドロップダウンを初期化する
Value sceneData[3];
sceneData[0].SetString("Scene with shadow");
sceneData[1].SetString("Show shadow volume");
sceneData[2].SetString("Shadow volume complexity");
pUIMovie->SetVariableArraySize("_root.SceneData", 3);
pUIMovie->SetVariableArray(Movie::SA_Value,
                           "_root.SceneData", 0, sceneData, 3);
```

Tutorial\Section6.1 フォルダのコードには、追加の ExternalInterface ハンドラが含まれ、輝度の制御、照明の数、シーンのタイプ、オリジナルの ShadowVolume インターフェイスに存在する他のコントロールに対応します。

注意: このサンプルでは、図示するために C++を使ってドロップダウン メニューを設定しました。一般的に、選択肢の静的なリストを持つドロップダウン メニューは、C++ではなく ActionScript を通じて初期化する必要があります。第 6.2 章からは、d3d9guide.swf/fla ファイルは ActionScript を使って、このドロップダウン リストを初期化します。

6.2.2ActionScript サブルーチンの実行

ActionScript 変数を変更する他に、ActionScript コードは、GFx::Movie::Invoke メソッドを使ってトリガすることができます。これは、より複雑な処理を行ったり、アニメーションをトリガしたり、現在のフレームを変更したり、プログラム的に UI コントロールのステートを変更したり、新規のボタンやテキストなどダイナミックに UI コンテンツを作成したりするような場合に便利です。

この章では GFx::Movie::Invoke を使って、F6 キーを押したときに、プログラム的に [Change Mesh] (メッシュの変更) テキスト入力ボックスを開き、F7 キーを押したときにこのボックスを閉じることとします。これは、SetVariable を使ってステートを変えても不可能です。アニメーションはこのラジオ ボタンをクリックすると発生するからです。SetVariable はアニメーションをトリガできませんが、Invoke で呼び出された ActionScript ルーチンでは可能です。

GFx::Movie::Invoke を呼び出して、WM_CHAR キーボード ハンドラで openMeshPath ルーチンを実行することができます。

```
            ...

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


            ...
```

Invoke を使用するときの一般的なエラーの 1 つとして、まだ利用できない ActionScript ルーチンを呼び出してしまうことがあります。このような場合エラーは Scaleform ログに書き出されます。ActionScript ルーチンは、そのルーチンが関係するフレームが再生されるか、または関係するネスト化されたオブジェクトが読み込まれるまで、利用できません。フレーム 1 のすべての ActionScript コ

ードは、GFx::Movie::Advance への最初の呼び出しが行われると直ちに、または GFx::MovieDef::CreateInstance を true に設定された initFirstFrame と共に呼び出す場合は、利用できるようになります。

このサンプルでは、Invoke の printf スタイルを使用しました。この関数の別のバージョンは GFx::Value を使って、効率的に非文字列引数を処理します。InvokeArgs は、アプリケーションが変数の引数リストにポインタを提供できるように、va_list 引数を取ること以外、Invoke と同じです。Invoke と InvokeArgs の関係は、printf と vprintf の関係に似ています。

6.3 複数の Flash ファイル間の通信

これまで説明してきた通信方法は、すべて C++が関わっていました。UI が複数の SWF ファイルに分割されているような大型のアプリケーションでは、インターフェイスのさまざまなコンポーネントがお互いに通信できるようにするには、大量の C++コードを書く必要があります。

たとえば、多人数参加型オンライン ゲーム (MMOG) は、インベントリーHUD、アバターHUD、さらにトレーディング ルームも備えている場合があります。剣やお金などのアイテムを、プレーヤーのインベントリーからトレーディング ルームに移し、別のプレーヤーに渡すことができます。プレーヤーが衣装を着ると、それはインベントリーHUD からアバターHUD に移動します。

これらの 3 つのインターフェイスは、個別の SWF ファイルに分けて、メモリを節約するため、別々に読み込む必要があります。ただし、C++コードを書いて、この 3 つのインターフェイスの 3 つの GFx::Movie オブジェクト間の通信を維持することは、すぐに煩わしくなるでしょう。

もっと良い方法があります。ActionScript は loadMovie と unloadMovie メソッドをサポートしています。この 2 つのメソッドは複数の SWF ファイルを単独の GFx::Movie で切り替えることができます。SWF ムービーが同じ GFx::Movie にある限り、その同じ変数の空間を共有し、ムービーを読み込むたびに毎回そのムービーを初期化する C++コードが必要なくなります。

例として、ActionScript 関数で container.swf ファイルを作成し、共有された変数の空間でムービーのロード/アンロードを行います。この container.swf ファイルはアート アセットを含まず、ムービーのロード/アンロードを管理する複数の ActionScript にすぎません。最初のステップは、MovieClipLoader オブジェクトの作成です。

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

この関数とここで使用される他の ActionScript 関数の詳細については、Flash の説明書を参照してください。

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

`loadMovie` を使って、ムービーを特定の名前の付いたクリップにロードすると、複雑なインターフェイスをより構造的に組織することができます。ムービーをツリー構造に整理し、それに応じて変数にアクセスすることもできます (例 _root.inventoryWindow.widget1.counter)。

多くの Flash ムービー クリップは、_root に基づく変数を参照します。ムービーが特定のレベルにロードされると、_root はそのレベルのベースを参照します。たとえば、レベル 6 にロードされたムービーについては、_root.counter と_level6.counter は、同じ変数を参照します。
`loadMovie` で_root.myMovie に同じムービーをロードしても正常に動作しません。_root.counter はこのツリーのベースのカウンタ変数を参照するからです。ツリー構造に組織されたムービーは、_lockroot = true を設定する必要があります。_lockroot は、_root を参照するすべてのファイルに、そのレベルのルートではなく、サブムービーのルートをポイントさせる ActionScript のプロパティです。ActionScript 変数、レベル、ムービー クリップの詳細については、Adobe Flash の説明書を参照してください。

サブムービーが組織されている方法に関係なく、同じ GFx::Movie 内で動作するムービークリップは、お互いの変数にアクセスして、共有ステートから操作でき、複雑なインターフェイスの作成を大幅に簡素化します。

また、container.fla は対応する関数も含み、不要なムービー クリップをアンロードして、消費するメモリを削減します (たとえば、ユーザーがウィンドウを閉じると、そのコンテンツを解放することができます)。

`loadMovie` が戻っても、ムービーは必ずしもロードを終える必要はありません。この loadClip 関数は、ムービーのロードを開始するだけです。その後はバックグラウンドで続行されます。使用しているアプリケーションが、ムービーのロードが終了したことを知る必要がある場合 (プログラム的に状態を初期化するためなど)、Loader のリスナー関数を使用することができます。container.fla は、ActionScript でイベントを処理する、または C++アプリケーションが行動できるように ExternalInterface 呼び出しを行うかのいずれかに拡張できる、これらの関数のプレースホルダを実装しています。

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

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

```
function handleLoadProgress( e:Event ):void {
    trace("Event.PROGRESS - Load Progress!");
}
function handleLoadComplete( e:Event ):void {
    trace("Event.COMPLETE - Load Complete!");
}
```

7 テクスチャーへのレンダリング

3D レンダリングではテクスチャーへのレンダリングは良く使用される高度なテクニックです。このテクニックを使用すると、バックバッファにレンダーする代わりにテクスチャーへのレンダリングができるようになります。こうしてできたテクスチャーは通常のテクスチャーとして、使いたい時にシーンのジオメトリーに適用できるようになります。例えば、このテクニックは「ゲーム内でのビルボード」の生成に使用できます。先ず Flash Studio 内でビルボードを SWF ファイルとして作成し、この SWF ファイルをテクスチャーにレンダーし、それからこのテクスチャーを適切なシーンジオメトリーに適用してください。
このチュートリアルでは、この前のチュートリアルからの UI を単にバックウォールにレンダーします。マウスの真ん中のボタンを押して光源を動かしてみると、これに応じて UI の見え方が変化するのがわかります。

それではこのサンプルがどのような仕組になっているかを見て行きましょう。

1. これまでのチュートリアルでは全て cell.x ファイルをロードしてきました。このファイルには部屋の床、壁、天井を表示するのに必要な頂点の座標、三角形のインデックス、テクスチャーに関する情報が入っています。このファイルを見ると、4 つ全ての壁に同じ壁テクスチャー（cellwall.jpg）が使われているのが分かります。ここでは後ろの壁にだけ UI をレンダーします。そのため、後ろの壁には別のテクスチャーを作成し、これを cellwallRT.jpg と呼びます。次に、この後ろの壁用のテクスチャーを参照するように MeshMaterialList アレイを次のように変更します。

```
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. これまでのチュートリアルで述べたように、cell.x ファイルのパーシングは DXUT フレームワークの内部で行われます。このパーシング中に DXUT は頂点とインデックスのバッファーを初期化し、材質リストで参照されているテクスチャーを生成します。このテクスチャーは呼び出し、CreateTexture で生成しましたが、この関数に引き渡した使用パラメーターは Render Textures には適切なものではありません。この件に関して詳しくは DXSDK のマニュアルを参照してください。レンダーターゲットとして使用できるテクスチャーを生成するには、正しい使用パラメーターを使用してレンダーテクスチャーを再度生成して、これをバックグランドのメッシュに添付します。また、元々DXUT が生成したテクスチャーを解除してメモリーの漏れを避けてください。

```
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. 次に AdvanceAndRender 関数を変更して、GFx をバックバッファ―上ではなく作成するテクスチャー上にレンダーします。ここでの主な手順は次の通りです。
   a. 先ず元々のバックバッファ―面を保存してレンダリング終了後にこれに戻れるようにしておきます。

      ```
      pd3dDevice->GetRenderTarget(0, &poldSurface);
      pd3dDevice->GetDepthStencilSurface(&poldDepthSurface);
      ```

   b. 次に作成しているテクスチャーからレンダー面を得てこれをレンダーターゲットに設定します。

      ```
      ptex->GetSurfaceLevel(0, &psurface);
      HRESULT hr = pd3dDevice->SetRenderTarget(0, psurface );
      ```

   c. 作成しているテクスチャーのディメンションへのムービービューポートを調節し、Display を呼び出し、手順 a で保存した元のレンダー面に戻ります。

8 OpenGL サンプル

Nvidia のモザイク化に基づいた、小さな OpenGL/GLUT インテグレーションサンプル (Tutorial¥OpenGL) (http://developer.download.nvidia.com/SDK/10/opengl/samples.html#tessellation) もあります。これは最小限度のサンプルで、D3D9 チュートリアルに比べて少し異なるアプローチを使用しています。tessellation.cpp と tessellation_original.cpp を比較すればわかるように、元の NVIDIA サンプルに数行のコードを加えただけのものです。メインプログラムから GFx への通信は、Init()、ShutDown、AdvanceAndDisplay()などの高レベル呼び出しで行われ、GFx 特定のインプリメンテーションコードは GFxPlayerImpl クラス (GFxPlayerGL.cpp) にカプセル化されています。

9 GFxExport を使った前処理

これまでは、Flash の SWF ファイルを直接ロードしていました。これは開発を合理化します。アーティストは開発を進めながら、新規の SWF コンテンツと切り替えて、改訂したゲームの実行ファイルを必要とせずに、ゲームでその結果を確認することができるからです。ただし、リリース版に SWF コンテンツを含めることはお勧めしません。SWF ファイルの読み込みには、処理オーバーヘッドが必要であり、読み込み時間に影響するからです。

GFxExport は、合理的に読み込むために最適化されたフォーマットに、SWF ファイルを処理するユーティリティです。前処理の間、イメージはゲームのリソース エンジンが管理するため、別のファイルに抽出されます。イメージは、最適化された読み込みとランタイム メモリの節約のために、DXT テクスチャ圧縮で DDS ファイルに変換することができます。埋め込まれたフォントが再圧縮されるか、またはビットマップ フォントが使用される場合に備えて、フォント テクスチャをオプションで事前に計算することができます。GFxExport のアウトプットは、オプションで圧縮できます。

GFX ファイルを使った展開は単純です。GFxExport ユーティリティは、ヘルプ画面で説明しているような、多種多様なオプションをサポートしています。d3d9guide.swf ファイルと fxplayer.swf ファイルを Scaleform フォーマットに変換する:

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

GFxexport.exe は C:\Program Files\Scaleform\ $(GFXSDK) ディレクトリにあります。これらのコマンドも Tutorial\Section7 フォルダの convert.bat ファイルに含まれています。

-i オプションは、イメージ フォーマット、この場合は DDS を指定します。DDS は DirectX プラットフォームに最も有効です。このフォーマットは DXT テクスチャ圧縮を有効にするからです。この圧縮は通常、ランタイム テクスチャ メモリを 4 倍節約します。

–c オプションは圧縮を有効にします。Scaleform ファイルのベクターと ActionScript コンテンツだけが圧縮されます。イメージの圧縮は、選択されたイメージの出力フォーマットと DXT 圧縮オプションによって変わります。

–share_images オプションは、異なる SWF ファイルの同じイメージを識別して、共有コピーを 1 つだけロードすることで、メモリ使用量を削減します。

Tutorial\Section8 フォルダの ShadowVolume のバージョンは、ファイル名引数を GFx::Loader::CreateMovie に変更するだけで、Scaleform ファイルを読み込むように修正されました。

10　次の段階へ

このチュートリアルでは、Scaleform 機能の基本的な概要を説明してきました。他にも以下のようなトピックを参照することをお勧めします:

- ゲーム内における Flash のテクスチャへのレンダリング。
- Scaleform Player SWF to Texture SDK サンプル、Gamebryo™ インテグレーション デモ、Unreal® Engine 3 インテグレーション デモを参照してください。
- パフォーマンスやその他の問題は、次の Developer Center の FAQs を参照してください
- Scale9 ウィンドウのサポートは、次の Documentation ページの Scale9Grid Overview で説明しています:
- カスタムの読み込み: Scaleform や SWF ファイルからの読み込みの他に、Flash コンテンツをメモリやゲームのリソース マネージャから、GFx::FileOpener をサブクラス化することで直接読み込むことができます。
- GFx::ImageCreator を使ったカスタム イメージとテクスチャ
- GFx::Translator を使ったローカライズ