Autodesk®
Scaleform®

Unity Scaleform Integration Overview

This document describes the main features of the Unity-Scaleform integration.

Author: Ankur Mohan, Andy Domin, JP Ratliff
Version: 1.05
Last Edited: December 3, 2013

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® Plugin for Unity

How to Contact Autodesk Scaleform:

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

Table of Contents

1 Introduction

This document describes the main features of the Unity-Scaleform integration. We assume a basic level of understanding of UI design in Adobe flash (Movieclips, Events, etc.) as well as some familiarity with using Scaleform GFx (Advance/Display/ExternalInterface etc). If you are not familiar with Flash and/or Scaleform, please refer to our Getting Started with Scaleform document included in our eval kit.

The rest of this document is structured as follows.

- “Installation” describes what you need to do after installing the package
- “Distribution” talks about what is included in this package.
- "Demos" talks about the ScaleformTutorial demo that is distributed with this package. Note that detailed instructions for building this demo on PC/iOS/Android are provided in the appendix.
- “Limitations” outlines the limitations of the current version of the integration.
- “Architecture” provides an overview of how the integration works and some general information about interfacing across managed-unmanaged code.
- “Using the Integration” details the steps you need to perform to use the integration in your own application.
- “Key Considerations” outlines some important considerations to be aware of while using the integration.
- “Special Considerations on iOS” outlines the differences between Windows, iOS and Android pertaining to this product. This is particularly important if you are an Android/iOS developer, as most of this document is written from the perspective of a Windows user.
- “Render To Texture” outlines the steps for rendering to a texture.

Detailed build instructions for Windows, iOS and Android platforms are provided in the appendix.

2 Installation

The Scaleform/Unity package is now integrated in to one Unity Package. To install, just create a new Unity project and import the .unitypackage located in the Unity folder of your install, or, if you purchased the integration from the Unity Asset store, click on the Import button after you download it from the asset store.

For detailed steps on importing the Unity level, please see the readme located in "Assets/Scaleform/Integrations/Unity/Doc/".

3 Distribution

This distribution includes a proof-of-concept demo project for Unity 4+ highlighting some of the strongest features of the Unity integration. This project can be used as a starting point, and to quickly understand the necessary resources for using the Scaleform plugin effectively. This project further contains the following:

Assets/Plugins: Plugin binaries for each platform.

Assets/Plugins/SF: C# scripts required by all projects to interface with the binaries.

Assets/Scaleform/Doc: Contains an extensive of documentation that covers all aspects of our SDK. If you are not familiar with Scaleform, the "Getting Started" guides located in Doc/GFx are a great place to start. You should also familiarize yourself with our profiling tool called AMP (Analyzer for Memory and Performance).

Assets/Scaleform/Bin/{PLATFORM}_Tools:

- Win32 or MacOS: Release player executable for PC/Mac (GFxMediaPlayer.exe). You can use this executable to run flash files on your development platform (PC/Mac), before using them with Unity. Typically, flash files can simply be dragged and dropped onto the player window. Please note that this distribution does not contain player executable that can be launched on the mobile device.
- Data/AS3: Some sample SWF/FLA files that can be used for testing purposes
- AmpClient.exe is our profiler tool that can be used to obtain detailed frame level rendering and memory statistics for Scaleform content running in the Unity editor or on a mobile device. Exporter.exe is the exporting application used to create compressed gfx files from swf files. More information about these applications can be obtained in the documentation

Assets/Scaleform/Lib: Contains Scaleform libraries necessary to build your application on the iOS. Not needed for PC/Mac/Android

Assets/Scaleform/Resources: Contains our CLIK (Common Lightweight Interface Kit) library, which contains prepackaged implementations for commonly used UI components such as buttons, scrolling lists etc. More information about getting started with CLIK can be obtained in our documentation. The demos included with this distribution make extensive use of CLIK and we encourage you to get familiar with using CLIK as doing so will make UI development faster and easier

Assets/Scaleform/Integrations/Unity/Doc: Host folder for this document

Assets/Scaleform/Integrations/Unity/Bin: Contains Debug/Release/Shipping version of Scaleform binaries. The binaries are dynamic link library (dll) on PC and shared object (.so) on Android. For iOS, see “Integrations/Unity/Lib” below. The release versions of these binaries are copied into project specific folders (for example, Assets/Plugins/ for Mac and PC, and Assets/Plugins/Android for Android) as well. The dynamic libraries are loaded by Unity automatically when you push play in the editor or run your application in the standalone mode.

Assets/Scaleform/Integrations/Unity/Src: Contains SFExports and a few other files needed to build your application on iOS, but is not needed on PC/Android.

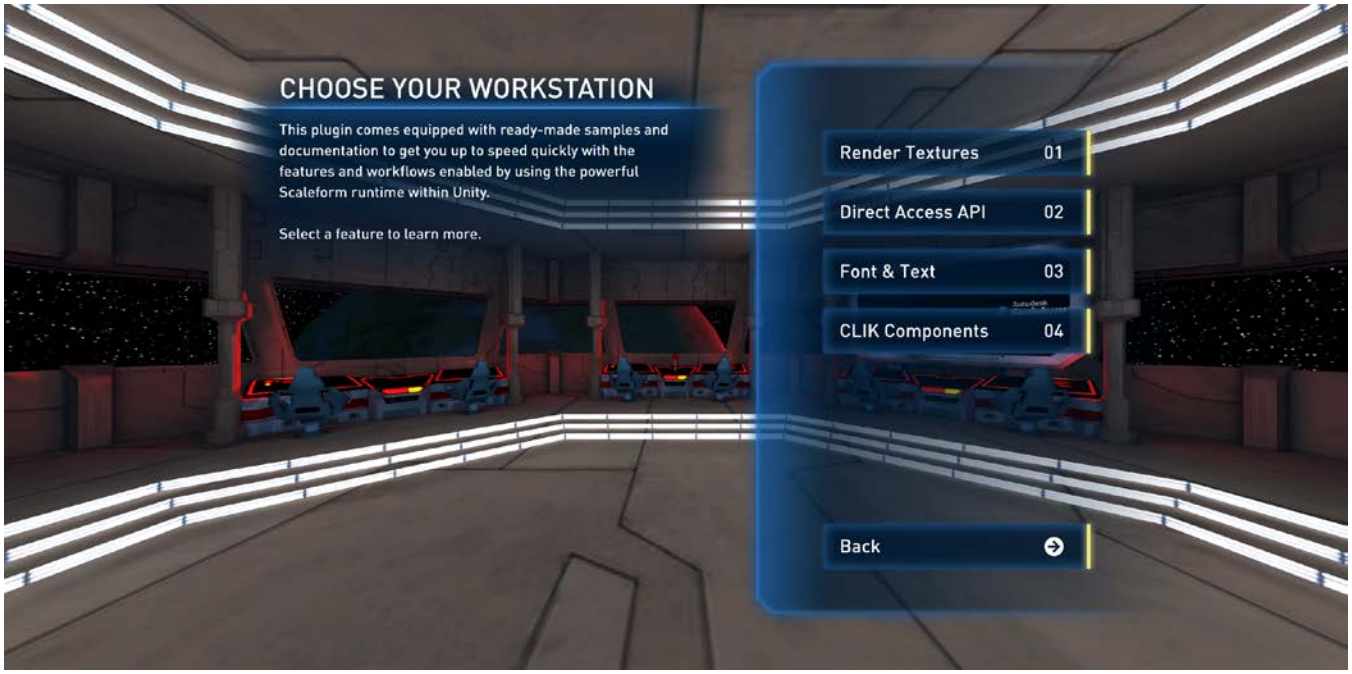
Note: The Scaleform binary files are located in Assets\Plugins\Android (for Android) and Assets\Plugins\ (for Mac and PC). On iOS, you will use the generated XCode to build your application. This project links in the Scaleform libs along with all other Unity and system libs needed to build your iOS application.

Assets/Scripts/Scaleform: Project-specific scripts that demonstrate instantiating the core C# scripts

Assets/StreamingAssets: Unity folder that holds project-specific Flash assets loaded at runtime.

4 ScaleformTutorial Demo

This demo is a proof-of-concept menu interface for a cross-platform game environment. The key features of Scaleform for Unity are outlined, and the source files provided. Feel free to use any of the resources within your own project.



As shown in the image above, this project demonstrates key features available in the plugin, including:

- **Render Textures** – Render 2d Flash content onto a texture usable on any mesh
- **Direct Access API** – Communicate easily between Flash and Unity scripts
- **Font & Text** – Vector-based font system complete with sub-pixel anti-aliasing
- **CLIK Components** – Drag and drop components frequently required in games

The flash assets (SWF/FLA files) used by the project are located in Assets\StreamingAssets. The files in this folder are copied on to the target platform automatically by Unity at deploy time. The only files needed at runtime are as follows:

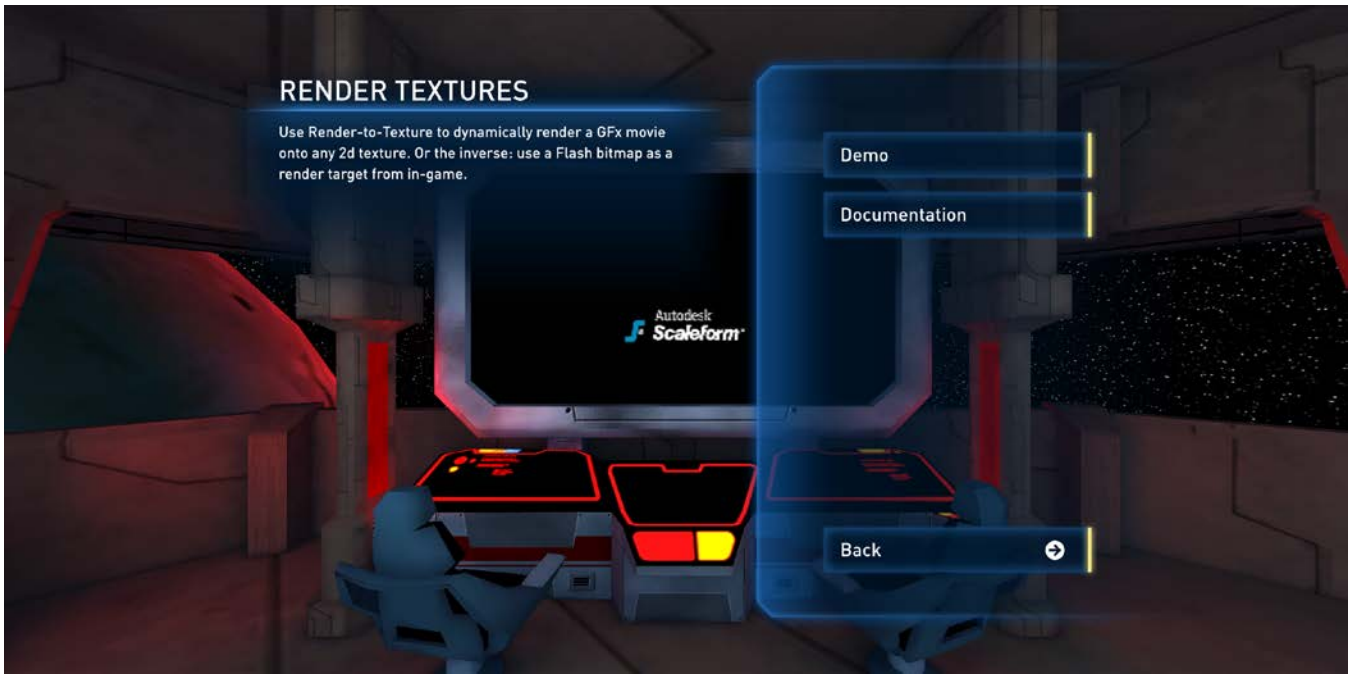
- **MainMenu.swf** – Controls navigation of demo environment
- **RenderTexture.swf** – Simple SWF rendered to texture

The FLA source files are also provided, along with additional AS files that contain the AS3 code. Each SWF file has a corresponding C# class that subclasses the Movie class (declared in SFMovie.cs). The Movie class implements functionality needed by all flash Movies such as calling Advance, passing

input events and so on. The subclass is used to implement movie specific functionality by defining methods that the user wants to call when a certain UI event takes place. For example, when the user clicks on a button in the UI or moves a slider bar.
The C# scripts required to load and display these flash assets are located in Assets\Scripts\Scaleform\swfs:

- **MainMenu.cs** – Extends core Movie class, wraps MainMenu.swf
- **RenderTexture.cs** – Extends core Movie class, wraps RenderTexture.swf

These scripts utilize the core scripts located in Assets\Plugins\SF, which define the C# interfaces used when programming against the native binaries. If you create a new Unity application that uses Scaleform, you can just copy these scripts into a corresponding location.



4.1 Putting it all together

The main entry point to this project is located in Assets/Scripts/Scaleform. **MyCamera.cs** is a class that extends SFCamera. This class instantiates Scaleform runtime, listens for OnGUI events (MouseMove, KeyUp/Down etc) and creates corresponding SFEvent objects, and puts the Scaleform Render event on the Unity Render queue. Recall that Unity is a multithreaded system with game update and rendering occurring on different threads. The multithreaded architecture requires that Scaleform rendering is called from the Unity render thread. This is why we insert the Scaleform rendering event into the Unity rendering queue instead of calling Scaleform rendering directly.

In our project, MyCamera is also responsible for state transitions and project-specific interactivity, along with providing a bridge between the project's Flash and Unity assets. We viewed this class as the main point of interaction, so analyzing this class as the main starting point is a great way to visualize the working pieces required to setup the plugin in your own project.

5 Limitations in the Current Implementation

This section details ongoing known issues or bugs in the current version.

5.1 JavaScript & Boo Not Currently Supported

While Unity supports multiple scripting languages, we have implemented our integration in C# so far. We believe that C# is the most widely used scripting language for Unity. If you would like to use JS/Boo in your scripting, it should be straightforward to convert the C# logic to other languages.

5.2 Unable to Initialize FMOD on iOS (Updated for Unity 4.2)

With Unity versions 3.5 to 4.1, when running on iOS, Scaleform is unable to initialize FMOD alongside Unity's own implementation of FMOD, and thus cannot playback sounds directly on its own.

Unity 4.2 has added the ability to disable audio from the editor. With this option disabled, Scaleform is able to initialize FMOD and playback sounds directly. Please note that this disables the ability to use the native Unity sound system.

If you are unable to upgrade to Unity 4.2 or do not want to disable Unity's native sound system, our suggestion is to embed sounds within Unity and trigger them via an ExternalInterface.call (the specifics of which are detailed later in this document).

5.3 Y-inverted Render Textures

Unity render-textures are Y-inverted relative to other textures. For this reason, materials with textures that are rendered to at runtime will appear inverted when running in Editor mode. If you would like to use Render-to-Texture using a normal mesh without changing any UVs, an easy solution is to set the y Tiling of the texture's material to -1, and the y Offset to 1.

5.4 Non-functioning Masks on Android Devices (Updated for Unity 4.2)

For Unity versions 3.x to 4.1 (on iOS and Android) stencil buffers are not allocated by default, which means that masks won't render correctly as they are implemented using a Stencil buffer. We detail a

workaround for this problem on iOS in the appendix. However, Unity doesn't expose the corresponding source files for Android.

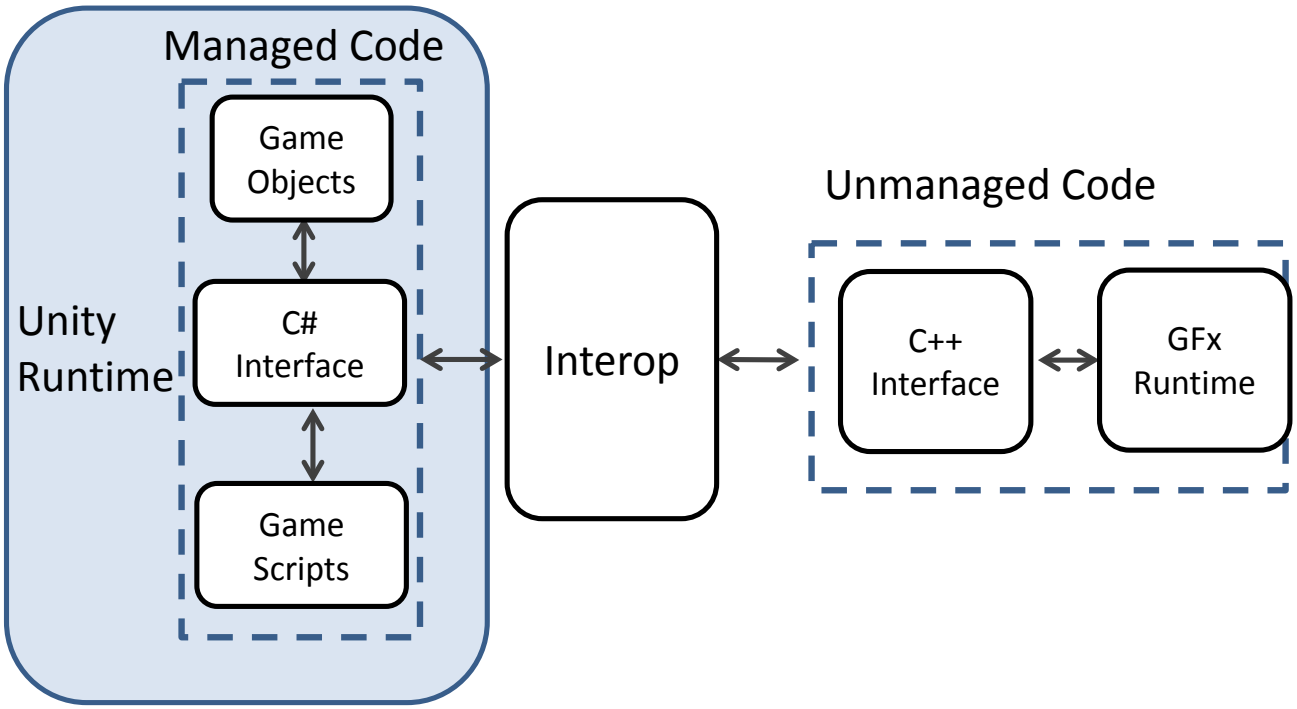
In Unity 4.2, this option has been exposed in the Editor's "Player Settings" under "Per-Platform" settings. Checking the option to use "24-bit Depth Buffer" will enable the use of masks not only in Android, but in iOS as well.

6 Architecture

This integration has two major components:

- C# Layer (Managed Code): This layer is responsible for instantiating the Scaleform runtime objects in the C++ layer, calling Advance/Display, relaying mouse/key events to Scaleform. It also exposes a convenient API that can be used for receiving callbacks from Scaleform, communicating with various scripts and objects in the game and so on.

- C++ Layer (Unmanaged Code): This layer acts as a wrapper for the Scaleform API and performs some of the marshaling needed to communicate with the C# layer.



On Windows, the binary component of the integration consists of a DLL that wraps Scaleform runtime and exports a series of functions that can be invoked from the scripting layer. On iOS, Scaleform libraries are linked into the application. Android is similar to Windows, with Scaleform runtime contained in a shared object file.

See the “Interop” section in the appendix for more information about communication between managed and unmanaged code.

7 Using the Integration

This section details the actual steps needed to implement Scaleform in Unity. For an actual proof-of-concept implementation of these steps, see the demo that comes packaged with the plugin.

1. Create a Camera game object and attach a script that derives from SFCamera. The SFCamera base class initializes the Scaleform runtime and instantiates the Scaleform Manager. In the derived class, the user can create the flash movie as explained later in this document.

Note: SFCamera exposes an InitParams structure, which is inspectable in the editor and can be used to set various Scaleform states. For example, you can set whether you want to initialize Video/Sound subsystems, which version of ActionScript (AS2/3/both) to use etc.

2. Create Scaleform movies by instantiating Movie derived classes. Movie already contains the core functionality necessary to perform various movie related tasks such as creating/destroying movies, calling advance/display by interacting with the Scaleform plugin.

3. Add custom event handlers in your Movie derived classes. Through the use of reflection in C#, the integration makes it easy to add event handlers. You just need to add functions that have the same name and accept the same parameters as the corresponding externalInterface callback.

In the next section we highlight many important considerations that are useful to be aware of while using the integration.

8 Key Considerations

8.1 Multithreaded Architecture

Unity introduced a multithreaded renderer so that rendering and game updates occur in different threads. This architecture ties in neatly with the Scaleform 4.x, which uses different threads for advance and display as well. In the multithreaded architecture, all rendering related resources (such as HAL) must be created in the render thread. Since plugins are usually driven by calling an exported function from game script which runs on the main game thread, any rendering related calls cannot be issued from script. In order to make it possible for low level plugins such as ours that perform rendering to work, Unity introduced some new API which calls an exported function with a predefined name and signature from the render thread. We use this API to support both D3Dx as well as multithreaded rendering.

For more information on this topic, please refer to Unity documentation:
http://docs.unity3d.com/Documentation/Manual/Advanced.html

Also look at SFExports.cpp (this file is distributed only with the iOS distribution):

```
void UnitySetGraphicsDevice (void* pdevice, int deviceType, int eventType)
```

And in SFCamera:

```
StartCoroutine("CallPluginAtEndOfFrames")
GL.IssuePluginEvent(0)
```

The GL.IssuePluginEvent method can be used to put commands on Unity's render queue. These commands will be executed on Unity's render thread.

Also, as stated in Unity documentation, rendering is multithreaded only on PC not on iOS/Android. The integration supports both multithreaded and single threaded rendering.

8.1.1 Use of Namespaces

All of the Scaleform integration related C# code is contained in the Scaleform or Scaleform::GFx namespace. This is consistent with namespace and class naming convention used in the Scaleform SDK. For example, the Movie class is defined in Scaleform::GFx namespace in both C++ and C#. The C# classes for which there is no counterpart in C++ (for example SFManager) are enclosed in the Scaleform namespace. This helps to prevent conflicts with Unity classes that might have the same names.

8.1.2 Creation and Destruction of Scaleform Runtime

In our integration, there are three initialization steps. First is the creation of Scaleform runtime, SFManager, Scaleform Renderer and the Hardware Abstraction Layer (HAL).

Initializing Scaleform runtime on PC is done in UnitySetGraphicsDevice, which is called from within Unity. If D3D renderer is being used, the pointer to the D3D device is also passed as a parameter in this function. On iOS/Android, the initialization of Scaleform runtime is performed in SF_Init.

The second step is setting various states on the Scaleform Loader as well as passing the settings specified in InitParams structure (SFCamera.cs) to Scaleform runtime, so that the runtime is initialized according to the settings specified by the user. This is done in the SF_Init function, which is called from Script in SFCamera:Start.

The third step is the initialization of the HAL object created in the first step. On multithreaded systems, the HAL initialization must take place on the renderer thread. This is done by issuing a GL.IssuePluginEvent on Windows and UnityRenderEvent on iOS/Android call with an eventId = 0.

The SFManager class contains most of the implementation of the C# part of the plugin. This class must be initialized before creating any movies.

Movies can be created in two ways: by calling SFManager:CreateMovie, or by creating a new instance of a Movie derived class. Both methods take a SFMovieCreationParams object as argument that encapsulates the movie name, viewport parameters etc. A typical movie creating call looks like this:

```
demo1 = new UI_Scene_Demo1(SFMgr, CreateMovieCreationParams("Demo1.swf"));
```

As noted below, a movie is identified by its ID, which is simply the integer representation of the C++ Movie pointer corresponding to the movie. The SFManager maintains a list of movies internally that is used to process events, call Advance/Display etc.

8.1.3 Advance/Display

Flash animations are advanced during Movie:Advance. In the integration, C++ Advance is called during SFManager.Advance which is called during SFCamera.Update. Movies are rendered during Display which on PC is called on the render thread internally by Unity through the UnityRenderEvent function and by calling UnityRenderEvent with eventId = 1 on iOS/Android.

Advance takes the time elapsed since last Update as an argument. This can be used to control the rate at which the movies are advanced. If you want finer control over the rate of advance of a particular movie, you can override the Advance method in your Movie derived class.

In most common use cases, the default implementation of Advance will suffice. Any game-movie communication such as invoke per tick should be implemented by overriding the Movie.Update function, which is called just before Advance. This provides a nice parallel between game update and Movie update.

8.1.4 Hit-Testing

Hit-Testing is used to check if an event (for example a mouse click) occurred over a flash element. The result of the hit test can be used to determine if the event should be passed to the game engine or not. For example, if you click over a UI button, you might want to prevent the mouse click from being processed further by the game. This support is provided through the SFManager::DoHitTest function. This function returns true if the input event occurred over any of the Flash movies currently being displayed.

SFManager::DoHitTest function takes the HitTestType as a parameter, which can have the following values:

```
HitTest_Bounds = 0,
HitTest_Shapes = 1,
HitTest_ButtonEvents = 2,
HitTest_ShapesNoInvisible = 3
```

These can be used to specify if the hit test should:

- Consider only the area contained within a shape or the entire shape bounding box
- Consider only the visible part of the shape

8.1.5 Invoking C# functions from ActionScript

The integration makes it easy to declare C# functions that can be called from ActionScript using ExternalInterface. For example, suppose you have a simple Flash file Button.swf that contains a movieclip. You want the position of the movieclip to change according to the position of the player every time you click a button. This logic can be implemented as follows:

Define a mouseclick handler in ActionScript:

```
function handleClick(event:MouseEvent):void
{
```

```
	if (ExternalInterface.available) {
		ExternalInterface.call("UpdatePosition", player_mc);
	}
};
```

Now in C#, create a class that derives from Movie and declare a function “UpdatePosition” that takes a Value as a parameter.

```
public class UI_Scene_Movie: Movie
{
	public UpdatePosition(Value movieRef)
	{
		// Get Player Position using Game API, use GetDisplayInfo to get the
		displayInfo object corresponding to movieRef and use SetDisplayInfo to
		reset the displayinfo after modifying it according to the player
		position.
	}
}
```

Internally, Unity integration uses C# reflection to figure out the correct C# function to call during an ExternalInterface callback. For this to work, you need to declare a function in a Movie derived class that has the same signature (same name and parameters) as that used in the ExternalInterface.call invocation. If no matching function is found, the ExternalInterface invocation will silently fail.

8.1.6 Passing Objects between C# and AS3

We have provided some helper functions to quickly pass objects between Unity’s C# and Scaleform’s AS3. This technique utilizes reflection in both runtimes to automatically serialize custom class objects and reconstruct them on the other end.

ConvertFromASObject

This function makes it easy to convert an ActionScript (AS) object into the corresponding C# object. For example, if you have an AS class defined like this:

```
	package
	{
		public class Scores
		{
			public	var Name:String;
			public	var Score:int;
			public	var Code:Number;
		}
```

```
}
```

And the corresponding C# class:

```
public class Scores
{
            public String Name { get; set; }
            public int Score { get; set; }
            public float Code {get; set; }
}
```

You can create an instance of Scores in C# like this:

```
Scores csScore = val.ConvertFromASObject(typeof(Scores)) as Scores;
```

Here *val* is a Value that corresponds to the AS scores object. This technique removes the need to manually create an instance of the Score class in C#, iterating over its properties and calling the appropriate getter/setting functions. The function handles nested classes as well.

ConvertToASObject

This function takes a C# object and creates the corresponding AS3 object.

Please note:

- If the composition of the C# class doesn't match that of the AS class (the Name/Type of the member variables are different), the functions will attempt to get/set the variables that match. The variables whose name or type are different will be left untouched.
- The AS class must have a default constructor. Since function overloading is not permitted in AS3, you can create a default constructor by simply providing default values of function parameters.

Here is an example of a default constructor required for using this technique:

```
public function Scores(name:String = "", score:int = 0, code: Number =
0)
{
      Name    = name;
      Score     = score;
      Code     = code;
}
```

8.1.7 Event Handling

Mouse and keyboard events are sent to Scaleform in SFCamera:OnGui. Each Movie can have its own viewport, and the transformation of mouse position to a movie viewport is performed internally by each Movie before passing the event to Scaleform runtime. The default transformation can be changed by overriding Movie.HandleMouseEvent. You can also prevent the movie from not responding to Mouse/Key events by overriding AcceptMouse/CharEvents and returning false.

8.1.8 Representation of Movie in Script

An integer ID is used to represent a Movie in script. This ID corresponds to the integer value represented by the Movie pointer in C++. This technique makes it easy to identify the Movie object represented by an ID by simply casting it to a Movie pointer.

```
SF_EXPORT void SF_HandleMouseEvent(int movieId, float x, float y)
{
      Movie* pmovie = reinterpret_cast<Movie*>(movieId);
      pManager->HandleMouseEvent(pmovie, x,y, icase);
}
```

8.1.9 Rendering Unity texture

It is possible to replace a named flash texture with a Unity texture by using the public function SFManager::ReplaceTexture. In order to do so, you must first assign an export identifier to the flash texture you wish to replace. You can then use the ReplaceTexture function to replace the flash texture with the desired Unity texture.

```
public bool ReplaceTexture(long movieId, String textureName, Texture texture)
```

The parameters are described as follows:

`movieId`: The ID of the SFMovie in which you want to replace a texture.
`textureName`: Corresponds to the resource name in the SWF that you want to replace, but for the time being, the resource must be named “texture1”.
`Texture` : An instance of the Unity texture

8.1.10 Lifetime Issues

The lifetime of objects created in C# is managed automatically. The garbage collector is invoked periodically by the C# runtime in order to destroy objects that don’t have any active references. The

Movie and Value class are used to represent Scaleform Movies and Values respectively. Therefore, the lifetime of the C++ movies and values are tied to the lifetimes of the corresponding C# objects. In order to make sure that the Scaleform Movies and Values are destroyed properly, we override the Finalize method of the corresponding C# classes and manually destroy the C++ objects.

Another point to be aware of is that the C# garbage collector runs on a different thread than the main game thread. Therefore, invocation of C++ Values and Movie methods must be made thread-safe wrt to the garbage collector thread. This is handled internally by the integration.

8.1.11 Direct Access API

Direct Access API (DAPI) is used to directly access and modify flash objects and their properties. DAPI eliminates the need for creating functions in ActionScript and invoking them every time some property of a flash object needs to be accessed, thereby dramatically improving speed and efficiency. The Value interface is used by Scaleform to represent simple types such as Int, Bool as well as complex types such as DisplayObject. The Unity Integration provides a wrapper layer for the entire DAPI interface, making it possible for the user to directly access flash objects in C#. DAPI enables you to write most of your UI logic in C# without having to write a lot of ActionScript code. For more information, please refer to Value.cs and look at the implementation of the Joystick demo in the ScaleformTutorial level.

8.1.12 Movie Destruction

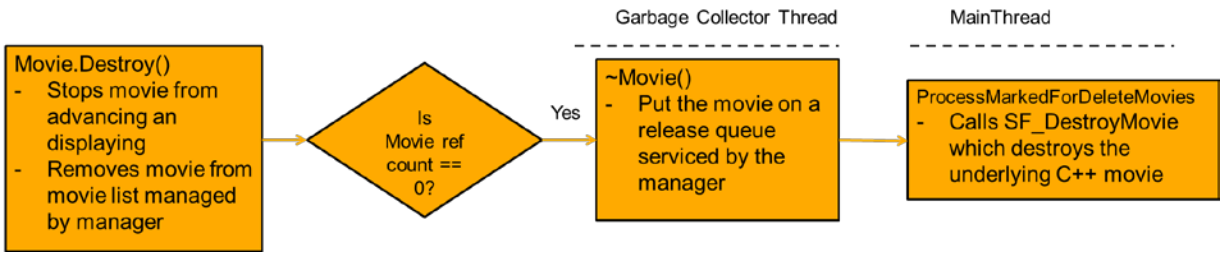
Movies are destroyed by calling Destroy() on the movie derived class. However, the destruction of Movie objects is an asynchronous process and movie resources might not be released right away. The timing of the movie resource release depends on how movies are destroyed. All the cases that can be encountered are described below.

1. Editor Mode

    a. The user creates and destroys movies as the game simulation is running. A movie can be destroyed by calling Destroy() on the movie derived class. Doing so removes the movie from the list of movies maintained by the manager, so that there are no more references pointing to the movie. It also instructs the runtime to not advance/display the movie anymore. If there are no more references pointing to the movie, the Movie destructor is called by the C# garbage collector. This however occurs on the garbage collector thread, instead of on the advance thread. This means that the underlying C++ movie object can't be destroyed during the C# movie destructor as the movie must be destroyed on the same thread on which movie creation and advance occurred. Therefore, when the movie destructor is called, the movie is put on a release list instead

of being destroyed immediately. The release list is serviced during game update, which is where the destruction of the actual C++ movie object occurs. If a movie has allocated rendering resources (which most movies do), C++ movie destruction requires those resources to be properly released. Scaleform is a multithreaded system with rendering resources allocated and released on the render thread. Hence the C++ movie destruction requires the render thread to tick a couple of times before the rendering resources can be released and the C++ movie object destroyed. This is not particularly important for this case as the Scaleform render thread is ticked frequently by Unity, but becomes an important consideration for case 1.b described below.

Note that C# garbage collection is determined by the C# runtime and may not take place immediately. If you want the garbage collection cycle to run immediately, call GC.Collect().



b. The user creates a number of movies during the game simulation, and then stops the game simulation by pushing the play button in the editor. In this case, the game object to which the manager is attached gets destroyed. During the OnDestroy() call of the gameobject, we call destroy on the movies and call SF_UnInit() which tries to destroy the underlying C++ movies. However, as mentioned above, since the movies can hold rendering resources, the actual destruction of the C++ movie could need the render thread to tick a couple of times. The render thread stops ticking when the user stops the game simulation and hence you might see the movies still being held in memory even after pausing the game simulation. These movies will get destroyed when you run the game simulation again or close the editor. In case of render-texture movies, you might see `refCount == (rs->m_Texture ? 1 : 0)` messages. This is because the underlying C++ movie keeps a reference to the render texture and may not be destroyed right away when the simulation is stopped. These movies will be destroyed when the simulation is run again or if the editor window is closed.

2. Standalone application

   a. Same as 1.a above.

b. In standalone mode, there is no notion of starting and stopping the game simulation and the destruction of movies is handled internally when the Scaleform runtime is destroyed.

8.1.13 Trace Statements

Actionscript trace statements are automatically directed to Unity Console output for easy debugging.

8.1.14 Deployment

In our demos, the flash assets are placed in Assets/StreamingAssets folder. Any files in this folder are copied over to the deployment platform automatically without modification.

9 Special Considerations on iOS

There are many differences in both development and deployment on iOS platforms. This section outlines many of these differences.

9.1.1 Behavior of pinvoke

As mentioned above, pinvoke is used for communication between managed and unmanaged code. There are significant differences in implementation of pinvoke between iOS and Windows platforms. We'll highlight a couple of these differences here:

1. Delegates don't work on iOS: As stated earlier, on Windows, we use delegates to call managed code methods from native code. Using delegates is a convenient since they execute right away and can return values. Unfortunately, due to limitations stemming from AOT (ahead of time compilation) on iOS Mono, delegates don't work. In order to address this limitation, we put the external interface notifications and value arguments on a shared queue. This queue is polled every frame and any commands stored on it are processed just like regular ExternalInterface callbacks. However, the queuing of ExternalInterface callbacks implies that they are no longer executed immediately, but during Unity's Update (look at SFManager::ProcessCommands() for implementation details), which makes these callbacks asynchronous. Furthermore, callbacks cannot have return values. We have found that these limitations are inconvenient, but not much of an issue.

2. The Marshal.PtrToStructure method doesn't work. This implies that marshaling of data from unmanaged heap pointers into classes has to be done manually. From a user's perspective, this limitation is not very important.

10 Render To Texture

Rendering to a texture is an important computer graphics technique that can be used to create many interesting and visually pleasing effects. This technique enables you to draw your Flash content to a game texture instead of the back buffer. The texture can then be applied to an arbitrary 3D object in your scene. The screenshots below show some RenderTexture use cases.

Figure 1: TV monitor with Flash movie drawn on it. This Flash movie accepts key input

Figure 2: Another TV monitor with Flash movie on it. This object has transparency enabled, so the background shows up through the areas where flash content is not rendered

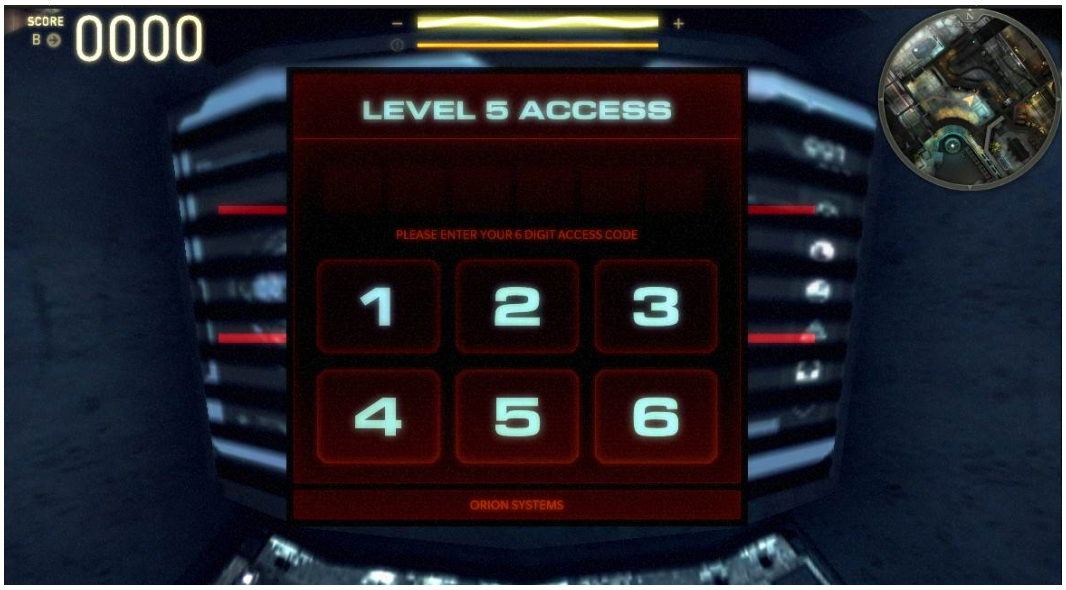


Figure 3: Another example of Render Texture. The Flash movie can accept mouse input for clicking on the keypad buttons

10.1 Using Render Texture in the Unity Integration

The Scaleform-Unity Integration makes it very simple to set your flash movies to render to a texture instead of the back buffer. Please follow the steps below to render to a texture.

1. Subclass SFRTT (defined in Plugins\SF\SFRTT.cs) and add code for creating the RenderTexture movie. You can follow the code in MyRTT.cs in the ScaleformTutorial to see an example. If your Flash movie doesn't send any external interface callbacks, you can use the Movie class directly without sub-classing.

2. If you expect your Render Texture movie to respond to Flash events such as mouse clicks, you must override the Movie class and provide the name of the subclass while creating the RenderTexture (RTT) movie. This is similar to the set up for an overlay movie.

3. In the Unity editor, drag and drop the derived class you created in step 1 to the object you want the texture to be drawn on.

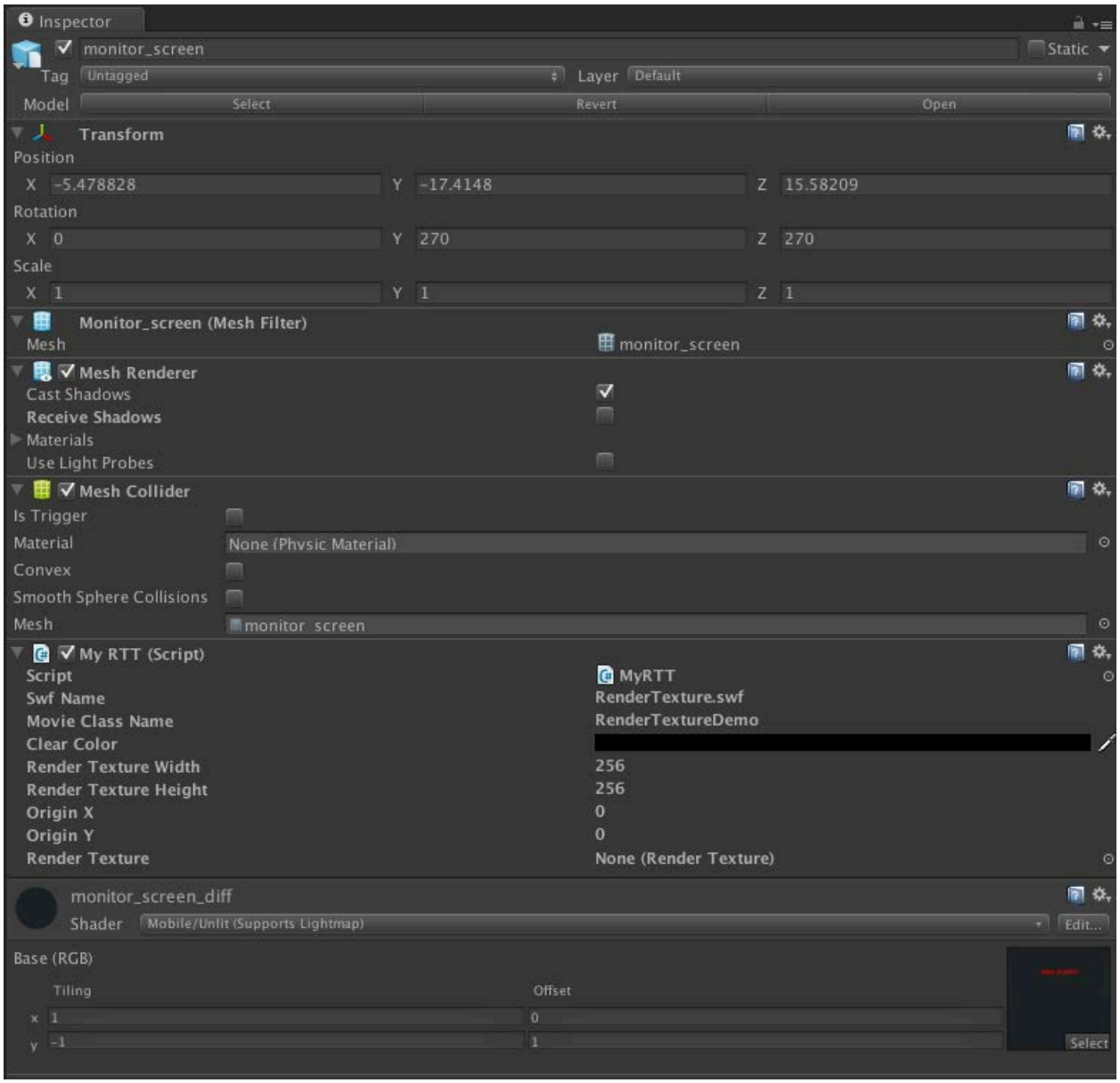


Figure 4: RenderTexture setup in ScaleformTutorial

You can specify the Movie subclass name that implements external interface callbacks, the name of the Flash movie (don't forget the .swf) and the width and height of the render texture in the editor itself. IMPORTANT: Make sure that the width and height are the same. If this is not the case, RenderToTexture doesn't work. This appears to be a limitation of Unity. You can also specify the clear color for the destination texture in the editor.

10.2 Hit Testing

We have implemented logic that converts mouse events to the coordinate space of the RTT movie (HandleMouseEvents in SFMovie.cs). This logic uses the MeshCollider components to calculate the point in texture space where the mouse click occurred. For this to work, your 3D object that uses the RTT must have a MeshCollider component attached to it.

This logic is intended to demonstrate one way to do hit testing. You can add your own custom hit testing logic if you so desire. Internally, Scaleform just uses the coordinates that are passed to it from C#.

10.3 Transferring Focus to the RTT movies

A SWF movie requires focus to receive user input. It is not always desirable for a movie to receive input, especially when rendered to a texture that is not in view. Scaleform makes it possible to control whether a particular movie is to receive input from the user. This can be achieved by toggling a movie's focus (SetFocus in SFMovie.cs).

It is up to the gameplay programmer to decide when it is appropriate for a render texture movie to gain or lose focus. A texture that can be hit tested is a good candidate for receiving mouse input. On the other hand, keyboard input does not have such a straightforward inclusion test.

There are several ways to determine whether it is appropriate for the user's keyboard input to be sent to the render texture. For example, one could test to see if an object is in the viewing frustum. In some cases there may be multiple render textures in the scene simultaneously, so it might not make sense to send keyboard events to all visible movies.

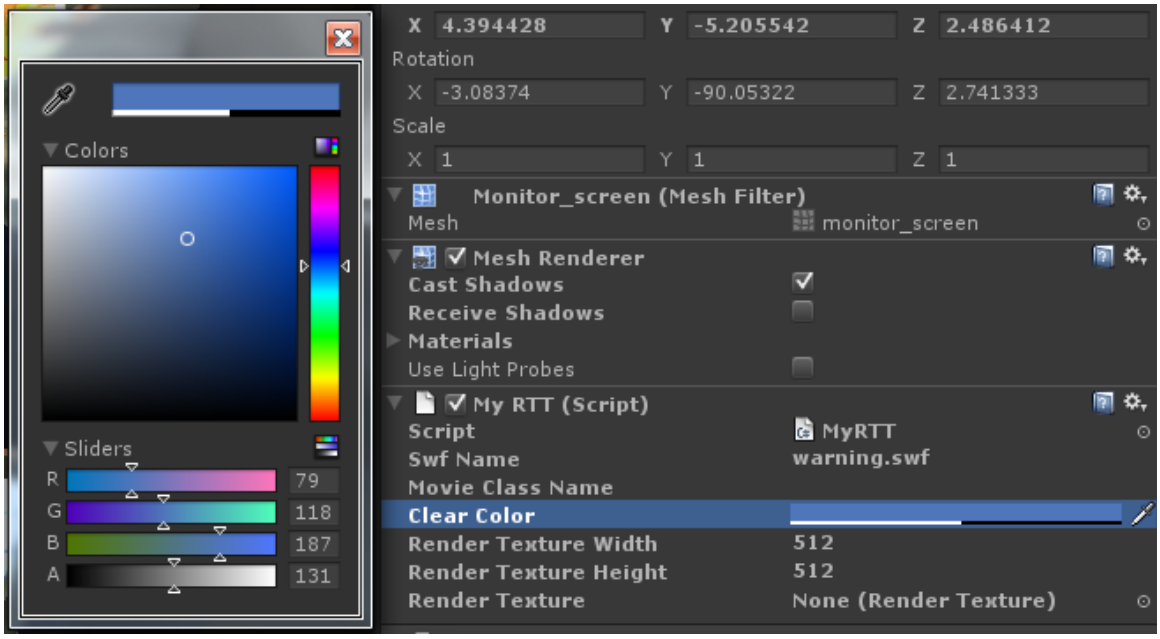
10.4 Adding Alpha Blending

10.4.1 Transparent Movies

In addition to the normal render-to-texture functionality present in the Scaleform plugin, it is also possible to have an alpha-blended movie with little to no effort. The following steps will cover how to do this:

1. Create a render-to-texture object using the instructions earlier in this document.

2. Ensure that the “Clear Color” parameter for the object has an alpha value of less than 255. For example, in the following image, the SWF will be rendered with a light blue background that has approximately 50% transparency:



3. Ensure that the material assigned to the object is transparent. For example, one might use the “Vertex Color” material in the Mobile -> Transparent category:

11 Render Layering

Support for layering has been added to this version of the plugin. Please note that layering only works in GL, D3Dx support is still under development.

11.1 Layer Levels

SFCamera now has a parameter, "Render Layer", which can be EndOfFrame, PreCamera, or PostCamera.

EndOfFrame has the same behavior that the plugin has always had. PreCamera will render the camera's Scaleform movie before the camera renders its contents. PostCamera will render the movie after the camera renders its contents (but before the next Camera in the hierarchy, if any).

SFCamera now has a pair of additional overrides for CreateMovieCreationparams, one of which allows the user to specify a background alpha value, and the other which allows the user to specify a full background color and alpha. This is necessary for PreCamera mode to work if Scaleform is the first thing that gets rendered during the frame since the camera must be set to Clear Depth Only. Otherwise the scene would never be cleared between frames.

12 Appendix

12.1 Interop

Managed code and unmanaged code interact with each other through the Platform Interface Services (PInvoke) system. On Windows, the system works as follows:

Suppose you want to call the function C++ `foo(char*)` from C#. First, implement this function in a .cpp file and export it in a dll.

```
__declspec(dllexport)void foo (char* str)
{
      printf("str\n");
}
```

Then, in unmanaged code (C#), declare this function using the DllImport attribute:
`[DllImport("DllName")]  public static extern void foo(String str);`

Now this function can be called from unmanaged code just like a regular function:
`foo("Hello World");`

Note that the conversion from C# `String` to C++ `char*` is handled by the pInvoke marshaling layer. Marshaling for simple data types is automatically implemented by pInvoke, however more complex types may have to be marshaled manually. An example is the DisplayInfo structure that encapsulates display attributes of Flash display objects. For detailed information about marshaling and pinvoke, please consult MSDN documentation.

Now let us consider the converse- calling a function in managed code from unmanaged code. This is needed in order to call a script function in response to an ExternalInterface callback. On Windows, this is implemented through delegates.

Managed Code:

```
// Step 1: Declare a Delegate. A Delegate is similar to a function pointer in C++
public delegate void ReceiveMessageDelegate(String message);

// Exported function to install the delegate
[DllImport("DllName")]  public static extern void
InstallDelegate(ReceiveMessageDelegate del);
```

```
// Step 2: The function we intend to call from C++
public void ReceiveMessage(String msg)
{
      Console.WriteLine("Message Received: " + msg);
}

// Step 3: Create and Install the delegate by passing it to C++
ReceiveMessageDelegate del = new ReceiveMessageDelegate (ReceiveMessage);
InstallDelegate(del);
```

Now consider the C++ (Unmanaged) side:

```
Imported function that receives the delegate:
// Declare a function pointer that has the same signature as the C# delegate
typedef void (ReceiveMessageDelegate*)(char*)
__declspec(dllexport)void InstallDelegate (ReceiveMessageDelegate func)
{
      // Call the delegate just like calling a function from a regular
     // function pointer
      func("Hello World");
}
```

12.2 Building on Windows (For Source Customers)

In order to build and debug the Starship demo on a Windows platform, please follow the following instructions. The build instructions for the iOS are provided in the next section.

12.2.1 Running the Demo without Debugging

1. Launch Unity.exe from the command prompt where you installed Unity (Typically C:\Program Files (x86)\Unity\Editor). By default, Unity editor is launched using D3D renderer. If you wish to use OpenGL renderer, launch Unity.exe with the following command line options "-force-opengl". Note that on PC, the default dll that is included in ScaleformTutorial is release-D3D, so if you wish to use OpenGL rendering, you will have to copy the appropriate OpenGL dll from Unity/Bin.

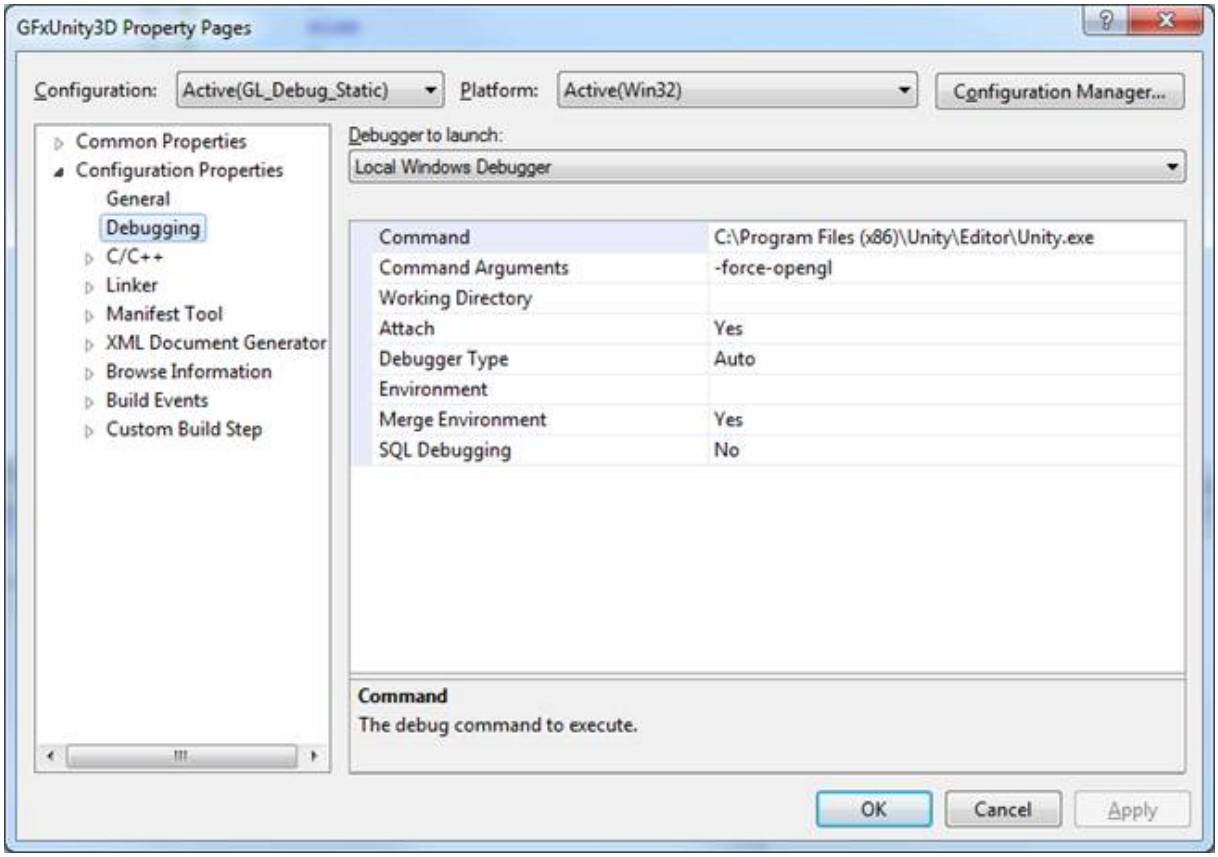
2. Open main_level.unity from "ScaleformTutorial/Assets/scenes/".

3. Push play. You should now see the ScaleformTutorial's MainMenu.

12.2.2 Debugging using Visual Studio

NOTE: The integration source code is only available to customers with Src access. Please write to Scaleform Support if you wish to receive source access. If you are not a source customer, this section is not relevant to you.

Open GFxUnity3D from Integrations\Unity\Projects\Win32\Msvc90 and change the Configuration Properties as the following:



Now push Cntrl+F5. This should launch Unity. Note that if you haven't launched Unity with the DummyLevel before, it will not be the default level that is launched during Unity startup, and you will have to browse to the folder where the DummyLevel is located manually, as described above.

In order to attach to debugger, push F5, now you should be able to set breakpoints in the integration code.

12.3 Building on the iOS

The process of building your Unity application on the iOS is significantly different than on Windows. The biggest difference is that the Scaleform plugin is actually linked in to your application, instead of being used as a dll. Therefore, the DllImport("libgfxunity3d") statements in the C# files have to be replaced by DllIImport("__Internal"). In order to make this easier and more transparent, we have split the implementation of classes that use imported functions into two files. For example, the implementation of Movie is split between SFMovie.cs and SFMovie_Imports.cs. The Imports file contains ifdef's so that the correct version of the imported functions is used automatically.

The Xcode project for the demo is generated by Unity and then automatically modified by a post-process build script (Assets/Editor/) to support Scaleform by adding the integration code and the Scaleform libraries.

Whenever any Unity content has changed (Scene, Mesh, Material, Unity Script, etc…), the Xcode project needs to be updated before it can viewed on the iOS device.

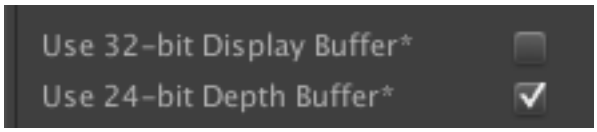
If this is your first time running the Project on iOS, note that you must setup a valid Provisioning Profile and Bundle Identifier within the Unity Project Settings for iOS. More information on properly configuring a Provisioning Profile and Bundle Identifier can be found on Unity's website and Apple Developer support.

By default, Unity does not provide a Stencil Attachment in the OpenGLES2 implementation for iOS. This prevents Scaleform from drawing masks. To add mask support for Unity 3.5 to Unity 4.1, you will need to make the following code change at line 145 of iPhone_GlesSupport.cpp (a source file provided by Unity and located in the generated Xcode project). Note that the changes are highlighted in yellow:

```
if(surface->depthFormat)
{
	GLES_CHK( glGenRenderbuffersOES(1, &surface->depthbuffer) );
	GLES_CHK( glBindRenderbufferOES(GL_RENDERBUFFER_OES, surface->depthbuffer) );
	GLES_CHK( glRenderbufferStorageOES(GL_RENDERBUFFER_OES, 0x88F0, surface->w, surface->h) );

	UNITY_DBG_LOG ("glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES, GL_RENDERBUFFER_OES, %d) :: AppCtrl\n", surface->depthbuffer);
	GLES_CHK( glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES, GL_RENDERBUFFER_OES, surface->depthbuffer) );
	GLES_CHK( glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_STENCIL_ATTACHMENT_OES, GL_RENDERBUFFER_OES, surface->depthbuffer) );
}
```

For Unity 4.2, iPhone_GlesSupport.cpp was removed, however an equivalent change can be made by checking the “Use 24-bit Depth Buffer” box in the Editor’s “Per-Platform Settings”.



12.4 Building on Android

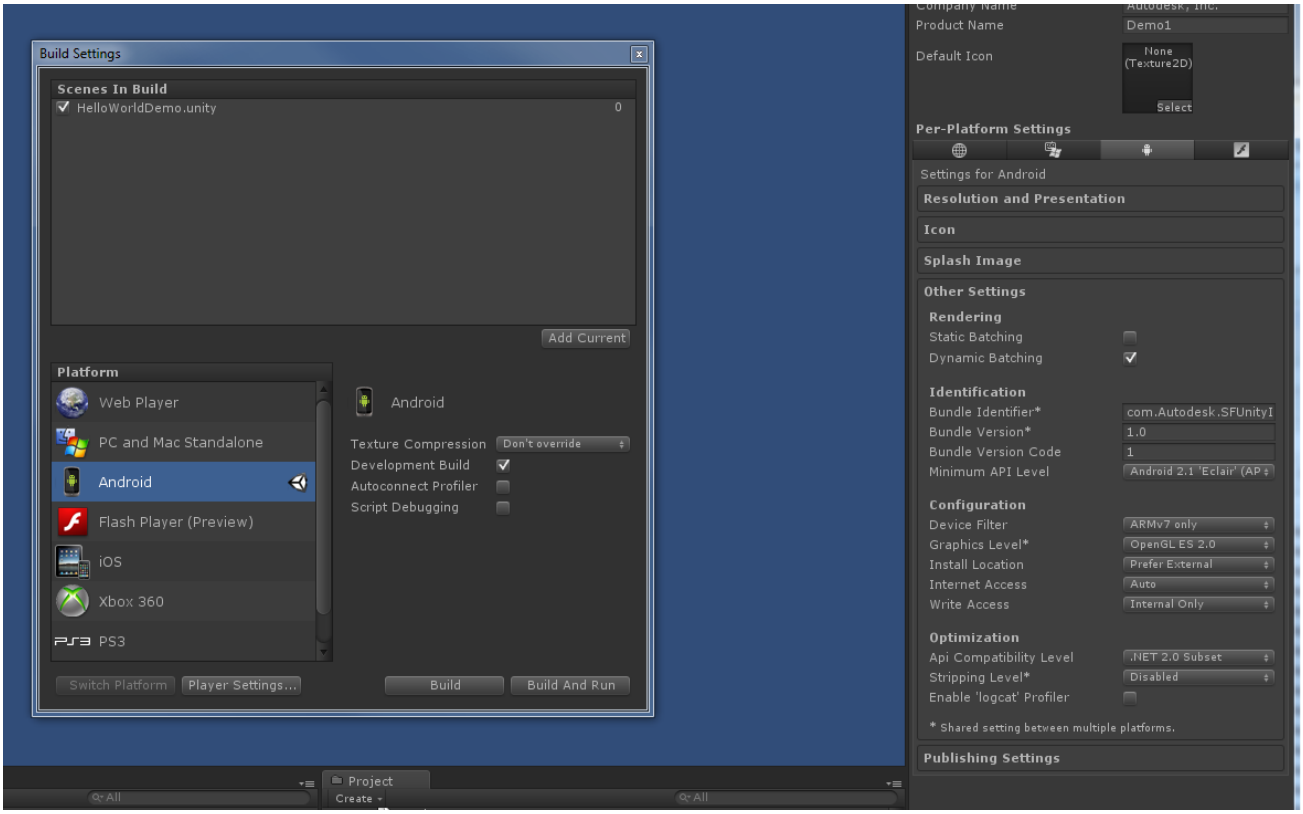
Running your Scaleform enabled Unity application on Android is similar to that on Windows. No C++ code needs to be compiled. As mentioned before, the Scaleform runtime is contained in the libgfxunity3d.so shared object file located in Assets/Scaleform/Bin/Android. For convenience, these files are copied into the Assets/Plugins/Android directory of ScaleformTutorial.

12.4.1 Creating your own Application

The key steps in creating an Android application are:

1. Copy the core integration script files into the Assets\Plugins\SF
2. Create a script that overrides SFCamera and attach it to a camera object in your scene
3. Copy the libgfxunity3d.so to Assets\Plugins\Android folder
4. Ensure that your flash assets are located in Assets\StreamingAssets

Now, in Unity editor switch to the Unity platform and make sure that the player settings are set as shown in the figure below.



Important Settings:

Device Filter: ARMv7
Graphics Level: OpenGL ES 2.0
Minimum API Level: API level 7 (This should be adjusted according to your target device range)