Autodesk®
Scaleform®

Scaleform 4.3 Renderer Guide

This document describes multi-threaded rendering configuration in Scaleform 4.3.

Author: Michael Antonov, Bart Muzzin

Version: 1.05
Last Edited: April 8, 2013

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

How to Contact Autodesk Scaleform:

Document	Scaleform 4.3 Renderer Guide
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

Scaleform 4.0 and higher includes a multi-threaded renderer design that allows movie advance and rendering (display) logic to be executed simultaneously on different threads, improving overall performance and enabling efficient Scaleform SDK integration into multi-threaded applications and game engines. This document describes the structure of the new renderer design, outlines the API changes made in Scaleform 4.0 to accommodate multi-threaded rendering, and provides the sample code necessary to get Scaleform rendering safely across threads.

2 Multi-Threading Renderer Changes

A number of APIs were changed from Scaleform 3.x to Scaleform 4.0 to accommodate the new design and multi-threaded rendering. This list highlights the major renderer areas that changed, that you will need to consider when working with multi-threading.

1. ***Renderer creation.*** Renderer creation is now associated with the render thread and also includes creation of platform-independent Renderer2D layer.
2. ***Render cache configuration.*** Mesh and font caches are now configured on the renderer and not on GFxLoader.
3. ***Texture resource creation.*** Texture creation needs to be either thread-safe or serviced by a render thread.
4. ***Rendering loop.*** Rendering of a movie is now done by passing a MovieDisplayHandle to the render thread and rendering it there through Renderer2D::Display.
5. ***Movie Shutdown.*** The GFx::Movie release operation may need to be serviced by the render thread.
6. ***Custom Renderers.*** Platform-independent APIs were redesigned to provide support for hardware vertex buffer caches and batching. The role of GRenderer in Scaleform 3.x is now handled by Render::HAL class.

These changes are outlined in the sections below, with code samples and more detailed explanation.

2.1 Renderer Creation

Renderer initialization on the Scaleform side involves the creation of two classes, Render::Renderer2D and Render::HAL. Renderer2D is a vector graphics renderer implementation that provides the Display method used to render movie objects. Render::HAL is a "hardware abstraction layer" interface used by Renderer2D; its specific implementations are located in platform-specific namespaces such as Render::D3D9::HAL and Render::PS3::HAL. Both of these are created with logic similar to the following:

```
D3DPRESENT_PARAMETERS   PresentParams;
IDirect3DDevice9*       pDevice = ...;
ThreadId                renderThreadId = Scaleform::GetCurrentThreadId();
Render::ThreadCommandQueue *commandQueue = ...;
Ptr<Render::D3D9::HAL>  pHal;
Ptr<Render::Renderer2D> pRenderer;

pHal = *SF_NEW Render::D3D9::HAL(commandQueue);
if (!pHal->InitHAL(Render::D3D9::HALInitParams(pDevice, PresentParams, 0,
                                               renderThreadId)))
{
    return false;
}
pRenderer = *SF_NEW Render::Renderer2D(pHal);
```

In this example, HALInitParams class provides platform-specific initialization parameters, with pDevice and PresentParams describing a user-configured device, while renderThreadId identifies the rendering thread used by the renderer. ThreadCommandQueue is an interface instance that may need to be implemented in the render thread which is discussed in more detail later in this document.

Renderer2D and HAL classes are not thread-safe and are designed to be used on the render thread only. They will usually be created on the render thread due to an initialization request or on the main thread while the render-thread is blocked.

2.2 Render Cache Configuration

In Scaleform 4.0 both the mesh and glyph caches are associated with Renderer2D and are maintained on the render thread. This behavior is different from Scaleform 3.x where these states were configured on GFxLoader. Caches are created when the associated Render::HAL is configured by a call to InitHAL; their memory is freed when ShutdownHAL is called.

The mesh cache stores vertex and index buffer data and is usually allocated directly from video memory; it is managed by system-specific logic as part of Render::HAL. Mesh cache memory is allocated in large chunks, with the first chunk pre-allocated on HAL initialization and can then grow up to a fixed limit. Mesh cache is configured as follows:

```
MeshCacheParams mcp;
mcp.MemReserve          = 1024 * 1024 * 3;
mcp.MemLimit            = 1024 * 1024 * 12;
mcp.MemGranularity      = 1024 * 1024 * 3;
mcp.LRUTailSize         = 1024 * 1024 * 4;
mcp.StagingBufferSize   = 1024 * 1024 * 2;

pRenderer->GetMeshCacheConfig()->SetParams(mcp);
```

Here, MemReserve defines the initial amount of cache memory allocated and MemLimit defines the maximum locatable amount. If both of these values are the same, no cache allocation will take place after initialization. MeshCache grows in blocks, specified by MemGranularity, and shrinks once there is more LRU cache content than provided for by LRUTailSize. StagingBufferSize is a system-memory buffer used for building batches; it can be set to 64K on consoles (the default value there), but must be larger on PC as it also serves as a secondary cache.

The glyph cache is used for rasterizing fonts, allowing them to be drawn efficiently from textures. Glyph cache is updated dynamically and has a fixed size determined at initialization time. It is configured as follows:

```
GlyphCacheParams gcp;
gcp.TextureWidth  = 1024;
gcp.TextureHeight = 1024;
gcp.NumTextures   = 1;
gcp.MaxSlotHeight = 48;

pRenderer->GetGlyphCacheConfig()->SetParams(gcp);
```

The configuration settings for glyph cache are virtually identical to Scaleform 3.3. TextureWidth, TextureHeight, and NumTextures define the size and number of alpha-only textures that will be allocated. MaxSlotHeight specifies the maximum height of a slot, in pixels, that will be allocated for a single glyph.

For initial settings to be modified, both caches should be configured before InitHAL is called. If SetParams is called after InitHAL, the associated cache will be flushed and memory re-allocated.

2.3 Texture Resource Creation

Scaleform 4.0 includes a redesigned GFx::ImageCreator and a new Render::TextureManager class that allow images to be loaded directly from video memory. Similar to an earlier version of Scaleform, ImageCreator is a state object installed on GFx::Loader to customize image data loading, as follows:

```
GFx::Loader loader;
...

SF::Ptr<GFx::ImageCreator> pimageCreator =
    *new GFx::ImageCreator(pRenderThread->GetTextureManager());
loader.SetImageCreator(pimageCreator);
```

The default ImageCreator takes an optional TextureManager pointer in the constructor; if a texture manager is specified and can't lose data, image content will be loaded directly into texture memory. The TextureManager object is returned by Render::HAL::GetTextureManager() method, which should be called on the render thread. The above sample code assumes that a render thread class provides GetTextureManager accessor method, which can only be called after render has been initialized. At the moment, this dependency implies that the rendering needs to be initialized before loading content if it is to be loaded directly into textures. If no texture manager is specified, image data will be loaded into system memory first and thus can take place before renderer initialization.

Unlike the rendering methods of Render::HAL, TextureManager methods such as CreateTexture must thread-safe since they may be called from different loading threads. This thread safety is achieved in one of two ways:

- By implementing texture memory allocation in a thread-safe manner, such as on consoles or with D3D9 when the device is crated with D3DCREATE_MULTITHREADED flag.
- By delegating texture creation to the render thread through the ThreadCommandQueue interface, specified in Render::HAL constructor.

The second approach means that ***if doing multi-threaded rendering with a non multi-threaded D3D device, you must implement ThreadCommandQueue*** and service it as necessary for texture creation. If this is not done, texture creation will block once called from a secondary thread. Please refer to Platform::RenderHALThread class for sample implementation.

2.4 Rendering Loop

With multi-threaded rendering, the traditional rendering loop is split up into two pieces: the advance/input processing logic executed by the advance thread and the render-thread loop that executes drawing commands such "draw frame". Scaleform 4.0 APIs accommodate multi-threading by defining GFx::MovieDisplayHandle that can be safely passed to the rendering thread. The GFx::Movie object itself should not be passed to the rendering thread since it is not inherently thread-safe and no longer contains the Display method.

The general movie rendering process can be broken down into the following steps:

1. ***Initialization.*** After GFx::Movie is created by GFx::MovieDef::CreateInstance, the user configures it by setting the viewport, obtaining the display handle and passing it to the render thread.
2. ***Main thread processing loop.*** On every frame, the user handles input and calls Advance to perform timeline and ActionScript processing. It is important that the Advance call be the last call after any Invoke/DirectAccess API modifications to the movie, as that is where the scene snapshot is taken for the frame. After Advance, the movie submits a Scaleform draw frame request to the render thread.
3. ***Rendering.*** Once the render thread receives a draw frame request, it "grabs" the most recently captured snapshot for the MovieDisplayHandle and renders it on screen.

The following reference demonstrates these steps in detail.

```
//------------------------------------------------------------------------
// 1. Movie Initialization

Ptr<GFx::Movie>          pMovie = ...;
GFx::MovieDisplayHandle  hMovieDisplay;

// Configure viewport after movie creation and grab the
// display handle.
pMovie->SetViewport(width, height, 0,0, width, height);
hMovieDisplay = pMovie->GetDisplayHandle();

// Pass the handle to render thread; this is engine-specific. In Scaleform Player,
// this is done by queuing up an internal function call.
pRenderThread->AddDisplayHandle(hMovieDisplay);


//------------------------------------------------------------------------
// 2. Processing loop

// Handle input and processing on the main thread. Movie callbacks
// such as ExternalInterface are also called here from Advance.
float deltaT = ...;
pMovie->HandleEvent(...);
```

```
    pMovie->Advance(deltaT);

    // Wait for previous frame rendering to complete and queue up
    // a draw frame request.
    pRenderThread->WaitForOutstandingDrawFrame();
    pRenderThread->DrawFrame();


    //-------------------------------------------------------------------------
    // 3. Rendering – Render thread DrawFrame logic
    Ptr<Render::Renderer2D> pRenderer = ...;

    pRenderer->BeginFrame();
    bool hasData = hMovieDisplay.NextCapture(pRenderer->GetContextNotify());
    if (hasData)
        pRenderer->Display(hMovieDisplay);
    pRenderer->EndFrame();
```

While most of the logic is self-explanatory, a couple of details stand out:

- WaitForOutstandingFrame – this helper function ensures that the render thread doesn't get ahead by more than one frame. This both saves CPU processing time and ensures that frame render thread queued up commands don't get out of sync with movie content. In single-threaded modes this is not necessary, as draw frame logic can be executed directly after Advance.
- MovieDisplay.NextCapture – This call is necessary to "grab" the most recent snapshot captured by Advance, making it current for rendering. This function will return false once the movie is no longer available, in which case nothing is drawn.

2.5 Movie Shutdown

When used with multi-threaded rendering, the GFx::Movie Release operation may need to be serviced by the render thread if it is executed before renderer objects are destroyed; servicing is required to release any render thread references to data structures that may be internal to the movie. By default, the movie shutdown command will be queued from Movie destructor through the Render::ThreadCommandQueue interface mentioned earlier. Developers must implement the ThreadCommandQueue interface to ensure proper shutdown.

As an alternative to servicing the command queue on the render thread, advance thread can use movie shutdown polling interface to first initiate a shutdown and then wait for it to be complete before releasing the movie. To do so, first call Movie::ShutdownRendering(false) to initiate shutdown and then use Movie::IsShutdownRenderingComplete to test for its completion every frame. On the render thread, NextCapture will handle the initiated shutdown and return false. Once shutdown is processed, advance thread can execute movie release without blocking.

3 Multi-threaded Rendering Concepts

Multithreaded rendering with Scaleform involves at least two threads - the ***advance thread*** and the ***rendering thread***.

Advance thread is typically the thread that creates a movie instance, handles its input processing, and calls GFx::Movie::Advance to execute the ActionScript and timeline animation. There can be more than one Advance thread in the application; however, each movie is typically accessed by only one advance thread. In most Scaleform samples, the advance thread is also the main application control thread, issuing commands to the render thread as needed.

The main job of the rendering thread is to render graphics on screen, which it does by outputting commands to the graphics device, typically through an abstraction layer such as Render::HAL. To minimize communication overhead, the Scaleform render thread processes macro commands such as "add movie" and "draw frame" rather than micro-commands such as "draw triangles". The render thread also manages mesh and glyph caches.

The render thread can either operate in lock-step or independently of the controlling advance thread, as follows:

- With Lock-step rendering, the controlling thread issues "draw frame" commands to the render thread, typically between movie advance processing. The two threads work in parallel with the rendering thread drawing the previous frame while the advance thread processes the next frame. On multi-core systems this results in overall improved performance, while the one-to-one relationship between thread frames is maintained.
- With independent operation mode, render thread draws frames independently from the advance thread, potentially running at a different frame-rate. Modifications to the movie become displayed shortly after they are complete, which is the next time the render thread gets around to drawing the updated movie snapshot.

The Scaleform Player application and Scaleform samples make use of the lock-step rendering mode both because it is easier to setup and more common in games.

3.1 Movie Snapshots

During multi-threaded rendering, the Advance thread can modify internal movie state at the same time as the render thread accesses it for display purposes. Since non-deterministic access of data by two threads would result crashes and/or corrupted frames, the new renderer makes use of snapshots to ensure that the render thread always sees a coherent frame.

In Scaleform, a movie snapshot is a captured state of a movie at a given point in time. By default, a snapshot is captured automatically at the end of the GFx::Movie::Advance call; it can also be captured explicitly by calling GFx::Movie::Capture. Once a snapshot is captured, its frame state becomes available for the render thread, while being stored separately from the dynamic state of the movie. When drawing a frame, the render thread calls NextCapture on a movie handle to grab the most recent snapshot and use it in rendering.

This design has several implications for developers:

- Movie::Advance and input processing logic does not need a critical section when executing concurrently with rendering.
- Capture and NextCapture calls can be called at different frequencies. If several consecutive Capture calls are made, movie snapshots are merged. If there are not enough Capture calls, the same frame may be rendered several times.
- Modifications made to the Movie due to input, Invoke, or Direct Access APIs are not visible by the render thread until Capture or Advance methods are called.

The last point exhibits an important behavior difference between Scaleform 4.0 and earlier versions. While in Scaleform 3.3 developer could call Movie::Invoke followed by Display to present changes to screen, they are now required to either call Capture or Advance for the changes to show up. Typically this is not required as input processing and Invoke calls can be grouped together before the call to Advance.

3.2 Preserving Rendering States

Render::Renderer2D::Display makes device calls to render a frame of the movie on the rendering device. For performance reasons, various device states, such as blending modes and texture storage settings, are not preserved and the state of the device will be different after the call to Render::Renderer2D::Display. Some applications may be adversely affected by this. The most straightforward solution is to save device state before the call to Display and restore it afterwards. Greater performance can be achieved by having the game engine re-initialize its required states after Scaleform rendering.

4 Renderer Design

With the general understanding of multi-threaded rendering in place, we can now look at the Scaleform 4.0 render design diagram.

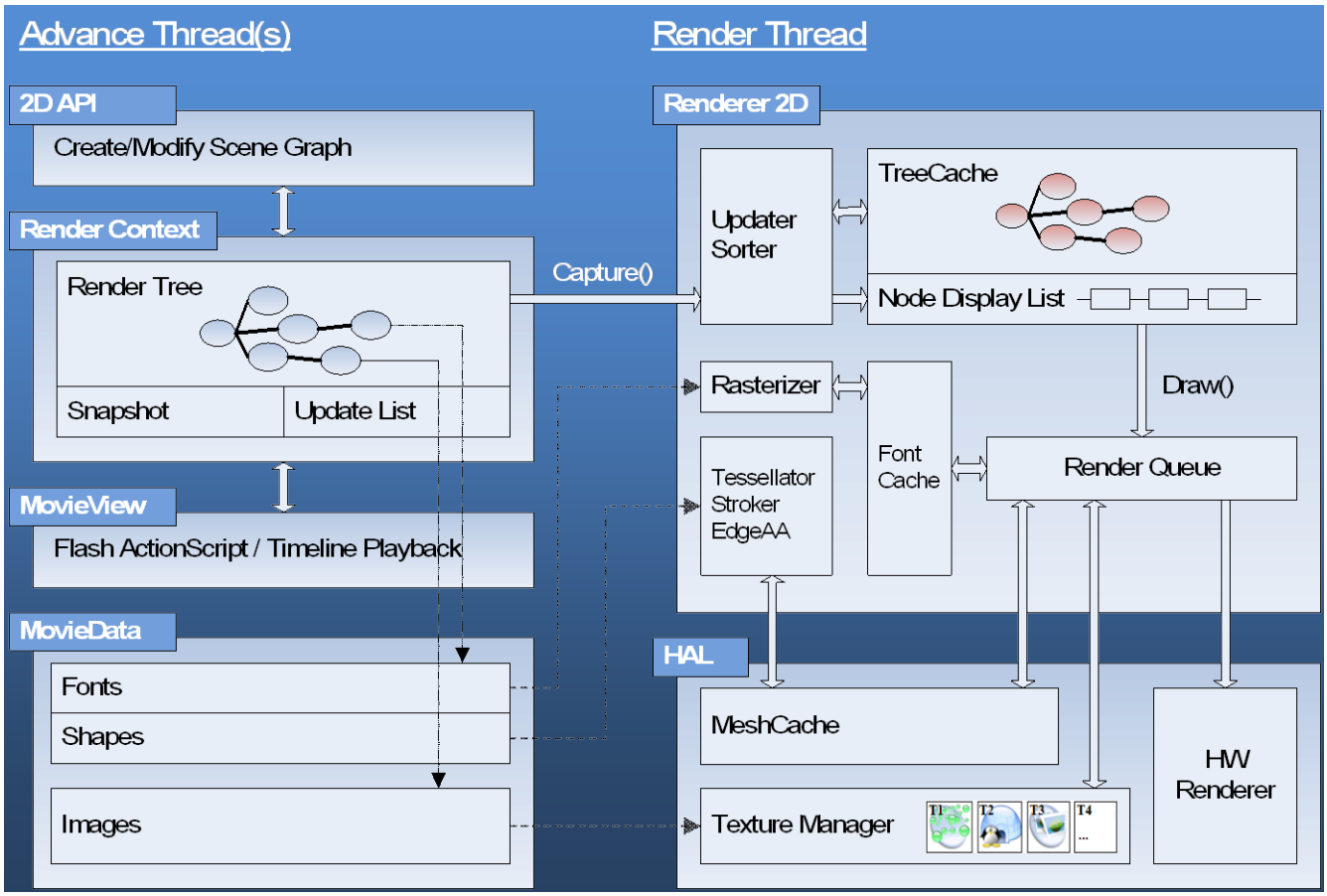

Figure 1: Render design

The diagram illustrates renderer subsystem interactions when running with two threads. On the left side, the Movie and MovieDef objects represent Scaleform movie instance and its internal shared data, respectively. The Render Context is a front-end to the rendering subsystem that manages the Render Tree; it is responsible for 2D scene graph snapshots and change lists accessible by the render tree. In Scaleform, render context is contained within the Movie and doesn't need to be accessed by end users.

On the right side, the render thread maintains two objects: Render::Renderer2D and Render::HAL. Renderer2D is the core of the render engine; it contains the Display method used to output movie snapshots to screen. Render::HAL stands for a "hardware abstraction layer" and is an abstract class having different implementation for each platform. Render HAL is responsible for executing graphics commands and managing resources such as vertex and texture buffers.

As seen from the diagram, the majority of the rendering systems are maintained and executed on the render thread. Specifically, the render thread is responsible for all of the following:

- Maintaining a sorted, cached version of the render tree and updating it when changes arrive from a new snapshot.
- Maintaining the Font Cache and updating it when new glyphs need to be rendered.
- Maintaining the Mesh Cache, which contains vertex and index buffers of tessellated vector data.
- Managing a Render Queue that enables efficient buffer update/lock management and minimizes CPU stalls while rendering.
- Emitting graphics commands to the device though HAL APIs.

In most cases keeping cache management logic on the render thread improves performance, as it frees up main thread for other work-intensive tasks such as ActionScript or game AI. Keeping caches on the rendering thread also reduces synchronization overhead and memory use, as it eliminates the need for extra buffers that would be needed to copy vertex and/or glyph data between threads.

5 Rendering Subsystem Description

5.1 TextureCache

The TextureCache system deals with unloading texture resources. It is intended to be used on memory constrained systems, such as mobile platforms, to reduce the total memory footprint, although it is available on every platform. However, it is only initialized by default on iOS, Android and PS Vita platforms, during the implicit creation of the TextureManager object, inside HAL::InitHAL. The TextureCache is only applicable on images whose image data is smaller than its texture data, meaning compressed images that must be decoded into uncompressed RGBA color data to be used by the GPU. These include PNG, JPEG and Flash-internal image formats. It does not apply to images that are compressed and can be used by the GPU directly, or images that are uncompressed – these data for these images is automatically unloaded once their texture data is copied from the image.

If the TextureManager is being allocated and passed into HALInitParams, it takes a TextureCache as the last parameter of its constructor. Only one implementation of the TextureCache is provided, named TextureCacheGeneric, however the user may create a derived TextureCache implementation. If this is the case, the TextureManager must be allocated and passed a derived TextureCache instance, and then the TextureManager must be passed into the InitHAL via the HALInitParams.

The generic implementation of TextureCache employs a simple LRU strategy for evicting textures from memory. It will not attempt to evict any textures until a threshold of applicable textures are allocated. Once the threshold has been reached (8MB by default), it will attempt to evict LRU texture, until it is under the limit again. If the limit cannot be achieved, because more than threshold of texture memory is used in a single frame, then a warning is emitted. The limit is adjustable at runtime, using the SetLimitSize function of the TextureCache. The system will never evict textures that are required for proper rendering.

6 Rendering with Custom Data

Occasionally, users want to render certain shapes differently than the methods available in the standard release of Scalefrom. This can be a way to achieve altered vertex transformations, or special fragment shader fill effects.

6.1 Intercepting Shape Rendering

Generally, only small subsets of shapes within a movie need special rendering effects. These objects are identified in ActionScript by using the AS2 extension properties "rendererString" and "rendererFloat", or the AS3 extension methods "setRendererString" and "setRendererFloat". Refer to the AS2/AS3 Extensions reference for details on their usage. Setting a string and/or float on a movie clip causes an instance of UserDataState::Data to be pushed on the Render::HAL's UserDataStack member via the HAL::PushUserData function, before the movie clip is rendered. The entire stack is accessible at any time, allowing for the usage of nested data; however, each movie clip can only contain a single String and/or Float. When the movie clip is finished rendering, the UserDataState::Data is popped off the stack, from the HAL::PopUserData function.

6.2 Example Performing Custom Rendering

Currently, to perform custom rendering taking into account the user data set on a movie clip, the HAL must be modified. While there are several ways to modify the HAL to perform custom rendering, this example will focus on a simple extension to the internal shader system, by adding a 2D position offset to all shapes that are rendered that have corresponding user data.

For this example create an AS3 SWF, which has a movie clip with instance name "m", and the following AS3 code:

```
import scaleform.gfx.*;
DisplayObjectEx.setRendererString(m, "Abc");
DisplayObjectEx.setRendererFloat(m, 17.1717);
```

Using the new shader scripting system in 4.1, we can easily add a new shader feature. In Src/Render/ShaderData.xml, add the highlighted line:

```
    <ShaderFeature id="Position">
      <ShaderFeatureFlavor id="Position2d" hide="true"/>
      <ShaderFeatureFlavor id="Position3d"/>
      <ShaderFeatureFlavor id="Position2dOffset"/>
    </ShaderFeature>
```

This line causes the shader system to use the "Position2dOffset" snippet as a possibility for processing the position data in the vertex shader. Later in the file, add the "Position2dOffset" snippet:

```
 <ShaderSource id="Position2dOffset" pipeline="Vertex">
    Position2d(
    attribute float4 pos    : POSITION,
    varying   float4 vpos   : POSITION,
    uniform   float4 mvp[2],
    uniform   float  poffset)
    {
    vpos = float4(0,0,0,1);
    vpos.x = dot(pos, mvp[0]) + poffset;
    vpos.y = dot(pos, mvp[1]) + poffset;
    }
  </ShaderSource>
```

This snippet is identical to the "Position2d" snippet, except that it adds the uniform value 'poffset' to the post-transformed x and y of the vertex.

Now, in D3D9_Shader.cpp, add the following block at the top of ShaderInterface::SetStaticShader:

```
   // See if we have user data on the stack.
    if (pHal->UserDataStack.GetSize() > 0 )
    {
        const UserDataState::Data* data = pHal->UserDataStack.Back();
        if (data->Flags &
(UserDataState::Data::Data_Float|UserDataState::Data::Data_String) &&
            data->RendererString.CompareNoCase("abc") == 0)
        {
            // Modify the shader we use, if we find the matching user data we were
expecting.
            vshader = (VertexShaderDesc::ShaderType)((int)vshader +
VertexShaderDesc::VS_base_Position2dOffset);
            shader  = (FragShaderDesc::ShaderType)  ((int)shader +
FragShaderDesc::FS_base_Position2dOffset);
        }
    }
```

This block checks to see whether the HAL's UserDataStack has the data we are trying to intercept at the top (has a string which is "abc", and has a float value as well). If it matches, then we modify the incoming shader types to add in the Position2dOffset path. Note that these symbols will not be available until ShaderData.xml has been run through its custom build step, which generates a new D3D9_ShaderDescs.h.

In addition to altering the shader definition used, we will also need to update the 'poffset' uniform before actually rendering. This can be accomplished by inserting the following block at the top of ShaderInterface::Finish:

```
   // See if we have user data on the stack.
    if (pHal->UserDataStack.GetSize() > 0 )
    {
        const UserDataState::Data* data = pHal->UserDataStack.Back();
        if (data->Flags &
(UserDataState::Data::Data_Float|UserDataState::Data::Data_String) &&
            data->RendererString.CompareNoCase("abc") == 0)
        {
            // Add the poffset uniform into the uniform data.
            for ( unsigned m = 0; m < Alg::Max<unsigned>(1, meshCount); ++m)
            {
                float poffset = data->RendererFloat / 256.0f;
                SetUniform(CurShaders, Uniform::SU_poffset, &poffset, 1, 0, m);
            }
        }
    }
```

Note that this example sets the uniform once per 'meshCount'. meshCount will be greater than one for batched or instanced draws (which the shader modifications made above will also support). Running the modified version of the HAL, the movie clip 'm' should now be moved slightly up and to the right.

6.3 Disabling Batching For Custom Drawing

The HALs can also be modified such that rendering is performed completely outside of the Scaleform shader system. The exact method for performing these modifications is not covered here, and can vary from case to case depending on the desired results.

When performing rendering externally, it can be desirable to disable batched and instanced mesh generation. This is because shaders authored externally from Scaleform do not generally support batching and/or instancing in the same way as the shaders internal to Scaleform. To disable batched drawing, simply use the AS2 "disableBatching" extension member, or the AS3 "disableBatching" extension function. Refer to the AS2/AS3 extension reference documentation for exact syntax.

7 GFxShaderMaker

7.1 General Description

Introduced in Scaleform 4.1, GFxShaderMaker is a utility that constructs and compiles shaders that Scaleform uses. When rebuilding the renderer projects (e.g. Render_D3D9), it is run as a custom build step on ShaderData.xml. This produces several files, which the renderer uses to compile and/or link the renderer. These vary from platform to platform. For example, on D3D9, it creates:

```
Src/Render/D3D9_ShaderDescs.h
Src/Render/D3D9_ShaderDescs.cpp
Src/Render/D3D9_ShaderBinary.cpp
```

On other platforms, the tool chain may be used to directly create a library containing the shaders, which will be linked with the executable.

7.2 HALs Supporting Multiple Shader APIs

7.2.1 D3D9

The D3D9 HAL supports both ShaderModel 2.0, and ShaderModel 3.0. By default, support for all these features levels is included in the HAL. You can explicitly remove support for either of these. This will save the extra size for creating the shader descriptors and binary shaders. This can be done by using an option to `GFxShaderMaker '-shadermodel'`, and providing a comma separated list of shader models. For example:

```
GFxShaderMaker.exe –platform D3D1x –shadermodel SM30
```

This will remove support for ShaderModel 2.0. If you attempt to call InitHAL with a device that only supports ShaderModel 2.0, it will fail.

7.2.2 D3D1x

The D3D1x HAL supports three feature levels, corresponding to the different levels of shader support that devices can be created with. These are (add required prefix for D3D11 and D3D10.1):
`FEATURE_LEVEL_10_0`
`FEATURE_LEVEL_9_3`
`FEATURE_LEVEL_9_1`

By default, support for all these features levels is included in the HAL. You can explicitly remove support for any of these. This will save the extra size for creating the shader descriptors and binary shaders. This can be done by using an option to `GFxShaderMaker '-featurelevel'`, and providing a comma separated list of required levels. For example:

```
GFxShaderMaker.exe –platform D3D1x –featurelevel
        FEATURE_LEVEL_10_0,FEATURE_LEVEL_9_1
```

This will remove support for `FEATURE_LEVEL9_3.` If you attempt to call InitHAL with a device that uses `FEATURE_LEVEL_9_3`, it will use next lowest supported level (`FEATURE_LEVEL_9_1` in this case).