Autodesk®
Scaleform®

Scaleform 4.2 渲染器线程指南

本文介绍 Scaleform 4.2 中的多线程渲染配置。

作者： Michael Antonov
版本： 1.05
上次编辑时间： 2013 年 4 月 18 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

How to Contact Autodesk Scaleform:

Document	Scaleform 4.2 Renderer Threading 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

目录

1 引言

Scaleform 4.0 包括一项新的多线程渲染器设计，它可实现在不同线程上同时执行电影推进 (Advance) 和渲染（显示）逻辑，从而提高整体性能并高效地将 Scaleform SDK 集成到多线程应用程序和游戏引擎之中。本文介绍这一新的渲染器设计的结构，概述为实现多线程渲染而在 Scaleform 4.0 中对 API 进行的更改，并提供在线程之间安全地达到 Scaleform 渲染所必需的示例代码。

2 多线程渲染器更改

许多 API 从 Scaleform 3.x 更改为 Scaleform 4.0，以利用新的设计和多线程渲染。此列表突出显示被更改的主要渲染器方面，它们是使用多线程技术时需要考虑的方面。

1. ***渲染器创建。***渲染器创建现在与渲染线程关联，而且还包括创建与平台无关的 Renderer2D 层。
2. ***渲染缓存配置。***现在，渲染器上配置了网格缓存和字体缓存，而未在 GFxLoader 上配置。
3. ***纹理资源创建。***纹理创建需要线程安全，或者由渲染线程提供服务。
4. ***渲染循环。***对电影进行的渲染现在是通过将 MovieDisplayHandle 传递给渲染线程然后在那里通过 Renderer2D::Display 对其进行渲染来完成的。
5. ***电影关闭。***GFx::Movie 发布操作可能需要由渲染线程来提供服务。
6. ***自定义渲染器。***重新设计了与平台无关的 API，以便为硬件顶点缓冲区缓存和分批处理提供支持。GRenderer 在 Scaleform 3.X 中的角色现在由 Render::HAL 类进行处理。

2.1 渲染器创建

Scaleform 端渲染器初始化涉及创建两个类：Render::Renderer2D 和 Render::HAL。Renderer2D 是一种矢量图形渲染器实现，提供渲染电影对象时使用的 Display 方法。Render::HAL 是 Renderer2D 使用的一个“硬件抽象层”接口；其特定实现位于特定平台的命名空间，例如 Render::D3D9::HAL 和 Render::PS3::HAL。这两个命名空间都是通过与下述情况相似的逻辑创建的：

```
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);
```

在此案例中，HALInitParams class 提供针对个别平台的初始化参数。在这当中， pDevice 与 PresentParams 提供予由用户设定的硬件的相应信息、而 renderThreadId 识别渲染器使用的渲染线程。ThreadCommandQueue 是渲染线程中可能需要实现的一个接口实例，本文将对此进行详细介绍。

Renderer2D 和 HAL 类并非线程安全的，而是设计为仅在渲染线程上使用。它们通常是由于初始化请求而在渲染线程上创建的，或者在渲染线程受阻时在主线程上创建。

2.2 渲染缓存配置

在 Scaleform 4.0 中，网格缓存和字形缓存都与 Renderer2D 关联，而且是在渲染线程上维护的。此行为不同于 Scaleform 3.x，因为在那里，这些状态是在 GFxLoader 上配置的。缓存是在通过调用 InitHAL 来配置关联的 Render::HAL 时创建的；而在调用 ShutdownHAL 时释放其内存。

网格缓存存储顶点和索引缓冲区数据，而且通常直接从视频内存直接分配；它是由系统特定的逻辑，作为 Render::HAL 的组成部分进行管理的。网格缓存内存以大块分配，HAL 初始化时预分配第一块，其后可以按一个固定限额增加。网格缓存配置如下：

```
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);
```

在这里，MemReserve 定义分配的缓存内存的初始数量，而 MemLimit 定义最大可分配数量。如果这两个值相同，初始化后就不会发生任何缓存分配。MeshCache 按块增大（由 MemGranularity 指定），一旦存在的 LRU 缓存内容大于 LRUTailSize 所规定的，MeshCache 就会缩小。StagingBufferSize 是一个用于构建批量的系统内存缓冲区；它可以在控制台上设置为 64K（那里的默认值），但在 PC 上必须大于此值，因为它还充当一个二级缓存。

字形缓存用于栅格化字体，以便于高效地自纹理绘制字体。字形缓存动态更新，并具有一个初始化时确定的固定大小。字形缓存配置如下：

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

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

用于字形缓存的配置设置实际上与 Scaleform 3.3 相同。TextureWidth、TextureHeight 和 NumTextures 定义将要分配的仅字形纹理的大小和数量。MaxSlotHeight 指定将为单个字形分配的最大槽高（单位：像素）。

对于要修改的初始设置，调用 InitHAL 之前，两个缓存均应配置。如果在 InitHAL 之后调用 SetParams，关联的缓存会被刷新，而且内存会被重新分配。

2.3 纹理资源创建

Scaleform 4.0 包括一个重新设计的 GFx::ImageCreator 以及一个新的 Render::TextureManager 类，它们允许从视频内存直接加载图像。与 Scaleform 的早期版本相似，ImageCreator 是安装在 GFx::Loader 上的状态对象，用来自定义图像数据加载，具体情况如下：

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

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

默认 ImageCreator 取构造函数中的一个可选 TextureManager 指针；如果指定了一个纹理管理器且该纹理管理器不会丢失数据，就会将图像内容直接加载到纹理内存之中。TextureManager 对象是由 Render::HAL::GetTextureManager() 方法返回的，后者应在渲染线程上调用。上面的代码示例假定一个渲染线程类提供 GetTextureManager 访问器方法，该方法只能在初始化渲染后调用。此时，此相关性表明：如果要将内容直接加载到纹理之中，在加载内容之前需要初始化渲染。如果未指定纹理管理器，就会先将图像数据加载到系统内存中，因而可在初始化渲染器之前发生。

与 Render::HAL 的渲染方法不同的是，TextureManager 方法（如 CreateTexture）必须是线程安全的，因为它们可以从不同加载线程调用。此线程安全通过以下两种方法之一实现：

- 通过线程安全的方式（例如在控制台上）实现纹理内存分配，或者在用 D3DCREATE_MULTITHREADED 旗标包装设备时利用 D3D9 实现。
- 通过借助 ThreadCommandQueue 接口（在 Render::HAL 构造函数中指定）将纹理创建指派给渲染线程。

此第二种方法意味着，***假如用非多线程 D3D 设备进行多线程渲染，就必须实现 ThreadCommandQueue*** 并根据纹理创建需要为其提供服务。若非如此，一旦从第二个线程调用，纹理创建就会阻滞。请参阅 Platform::RenderHALThread 类，了解示例实现情况。

2.4 渲染循环

通过多线程渲染，传统渲染循环被拆分成两个部分：推进线程执行的推进/输入处理逻辑，以及执行绘图命令（如绘制帧摄）的渲染线程循环。Scaleform 4.0 APIs 通过定义可安全地传递到渲染线程的 GFx::MovieDisplayHandle 来实现多线程技术。GFx::Movie 对象本身并不应传递到渲染线程，因为它本质上并非线程安全的，而且不再包含 Display 方法。

一般电影渲染过程可分为以下几个步骤：

1. ***初始化。***由 GFx::MovieDef::CreateInstance 创建 GFx::Movie 之后，用户通过设置视窗、获取显示句柄并将其传递到渲染线程来对 GFx::Movie 进行配置。
2. ***主线程处理循环。***在每个帧上，用户处理输入并调用 Advance 来执行时间线和 ActionScript 处理。在对电影进行一切 Invoke/DirectAccess API 修改之后调用 Advance 是最后一次调用非常重要，因为那是对帧拍摄现场快照的地方。在 Advance 之后，电影将一个 Scaleform 绘制帧请求提交给渲染线程。
3. ***渲染。***一旦渲染线程收到绘制帧请求，它就抓取摄 MovieDisplayHandle 的最新捕获的快照，并在屏幕上对其进行渲染。

以下参考将详细说明这些步骤。

```
    //------------------------------------------------------------------------
// 1. 电影初始化

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

// 创建电影后配置视窗并抓取
// 显示句柄。
    pMovie->SetViewport(width, height, 0,0, width, height);
    hMovieDisplay = pMovie->GetDisplayHandle();

// 将该句柄传递到渲染线程；这是引擎特定的。在 Scaleform Player 中，
// 只是通过将一个内部函数调用排进队列里来完成的。
    pRenderThread->AddDisplayHandle(hMovieDisplay);


    //------------------------------------------------------------------------
// 2. 处理循环

// 在主线程上处理输入和处理操作。电影回调，
// 如 ExternalInterface，也在这里从 Advance 进行调用。
    float deltaT = ...;
    pMovie->HandleEvent(...);
    pMovie->Advance(deltaT);
```

```
// 等待前一个帧渲染完成并将
// 绘制帧请求排入队列。
    pRenderThread->WaitForOutstandingDrawFrame();
    pRenderThread->DrawFrame();


    //-------------------------------------------------------------------------
// 3. 渲染 – 渲染线程 DrawFrame 逻辑
    Ptr<Render::Renderer2D> pRenderer = ...;

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

尽管该逻辑的大部分一目了然，但仍然需要注意以下几个细节：

- WaitForOutstandingFrame – 此帮助函数确保渲染线程不会超前一个帧以上。这不仅节约 CPU 处理时间，而且确保帧渲染线程排队命令不会超越与电影内容同步。在单线程模式中，这并不必要，因为可以直接在 Advance 之后执行绘制帧逻辑。
- MovieDisplay.NextCapture – 要想“抓取”Advance 捕获的最新快照，就必须进行此调用，从而使其保持最新，以便进行渲染。一旦电影不再可用，此函数就会返回 false（假），在这种情况下，不会绘制任何东西。

2.5 电影关闭

当与多线程渲染一起使用时，如果在毁坏渲染器对象之前执行 GFx::Movie Release 操作，则可能需要有渲染线程为其提供服务；要向电影的内部数据结构发布任何渲染线程参考，就必需此服务。默认情况下，将会通过前面提及的 Render::ThreadCommandQueue 接口从 Movie 解构函数对电影关闭命令进行排队。开发者必须实现 ThreadCommandQueue 接口以确保正确关闭。

作为在渲染线程上为命令排队提供服务的替代选择，推进线程可使用电影关闭轮询接口先发起关闭，然后等待其完成之后再发布电影。为此，请先调用 Movie::ShutdownRendering(false) 以发起关闭，然后使用 Movie::IsShutdownRenderingComplete 来针对每个帧测试其是否已经完成。在渲染线程上，NextCapture 将处理发起的关闭并返回 false。一旦处理关闭，推进线程就可无阻滞地执行电影发布。

3 多线程渲染概念

使用 Scaleform 进行的多线程渲染至少包含两个线程 – ***推进线程***和***渲染线程***。

推进线程一般是创建电影实例、处理其输入处理以及调用 GFx::Movie::Advance 以执行 ActionScript 和时间线动画的线程。应用程序中可以有不止一个 Advance 线程；不过，每部电影一般只由一个 Advance 线程访问。在多数 Scaleform 示例中，Advance 线程也是主要应用程序控制线程，根据需要向渲染线程发出命令。

渲染线程的主要任务是渲染屏幕上的图形，渲染线程通过向图形设备输出命令（一般经由像 Render::HAL 这样的应用层）来完成此操作。要最大限度地减少通信开销，Scaleform 渲染线程处理像添加电影摄和绘制帧摄这样的宏命令，而不是像绘制三角形摄这样的宏命令。渲染线程还管理网格缓存和字形缓存。

渲染线程可以锁步工作，或者独立于控制推进线程工作，具体情况如下：

- 利用锁步渲染，控制线程向渲染线程发出绘制帧摄命令，一般在电影推进处理之间。这两个线程并行工作，渲染线程绘制前一个帧，而推进线程则处理下一个帧。在多核系统上，这会提高总体性能，同时还保持线程框架之间的一对一关系。
- 对于独立工作模式，渲染线程在绘制帧时不受可能以不同帧速运行的推进线程影响。对电影的修改会在修改完成之后不久显示出来，这是渲染线程抽出时间绘制已更新的电影快照的下一个时间段。

Scaleform Player 应用程序和 Scaleform 示例利用锁步渲染模式，这一方面因为设置更方便，另一方面因为游戏中更为普遍。

3.1 电影快照

在多线程渲染期间，Advance 线程可在渲染线程出于显示目的访问内部电影状态的同时修改内部电影状态。由于两种线程对数据的非确定性访问会导致崩溃和/或坏帧，新的渲染器利用快照确保渲染线程始终看到一个连贯的帧。

在 Scaleform 中，电影快照是电影在给定时间点的一种被捕获状态。默认情况下，在调用 GFx::Movie::Advance 结束时自动捕获快照，也可以通过调用 GFx::Movie::Capture 来明确地捕获。一旦捕获一个快照，渲染线程就可以使用其帧状态，但与电影的动态状态分开存储。绘制帧时，渲染线程在一个电影句柄上调用 NextCapture 以抓取最近的快照并将其用于渲染。

此设计对于开发者来说具有多种含义：

- 当与渲染同时执行时，Movie::Advance 和输入处理逻辑并不需要一个关键节。
- 可以按照不同频率调用 Capture 和 NextCapture。如果进行多次连续的 Capture 调用，就会合并电影快照。如果没有足够的 Capture 调用，同一帧就可能会渲染多次。
- 调用 Capture 或 Advance 方法之前，渲染线程看不到由于输入、Invoke 或 Direct Access API 对 Movie 进行的修改。

最后一点说明 Scaleform 4.0 与早期版本之间的一个重要行为差异。尽管在 Scaleform 3.3 中开发者可以调用 Display 遵循的 Movie::Invoke 来呈现对屏幕的更改，但现在要呈现更改他们还必须调用 Capture 或 Advance。一般情况下，不需要这样做，因为可以在调用 Advance 之前把输入处理和 Invoke 调用组合在一起。

3.2 保留描绘状态

Render::Renderer2D::Display 进行设备调用来在渲染设备上渲染电影的帧。出于性能原因，不保留混合模式和纹理存储设置等各种设备状态，而且调用 Render::Renderer2D::Display 之后设备的状态会发生变化。
有些应用程序可能受到负面影响。最直接的做法是在调用之前先保存设备状态信息，调用之后再恢复这些信息。游戏引擎在 Scaleform 渲染后重新初始化必要的状态信息，这样能达到优越的性能。

4 渲染器设计

在大体上了解多线程渲染之后，我们现在可以看看 Scaleform 4.0 渲染设计图表。

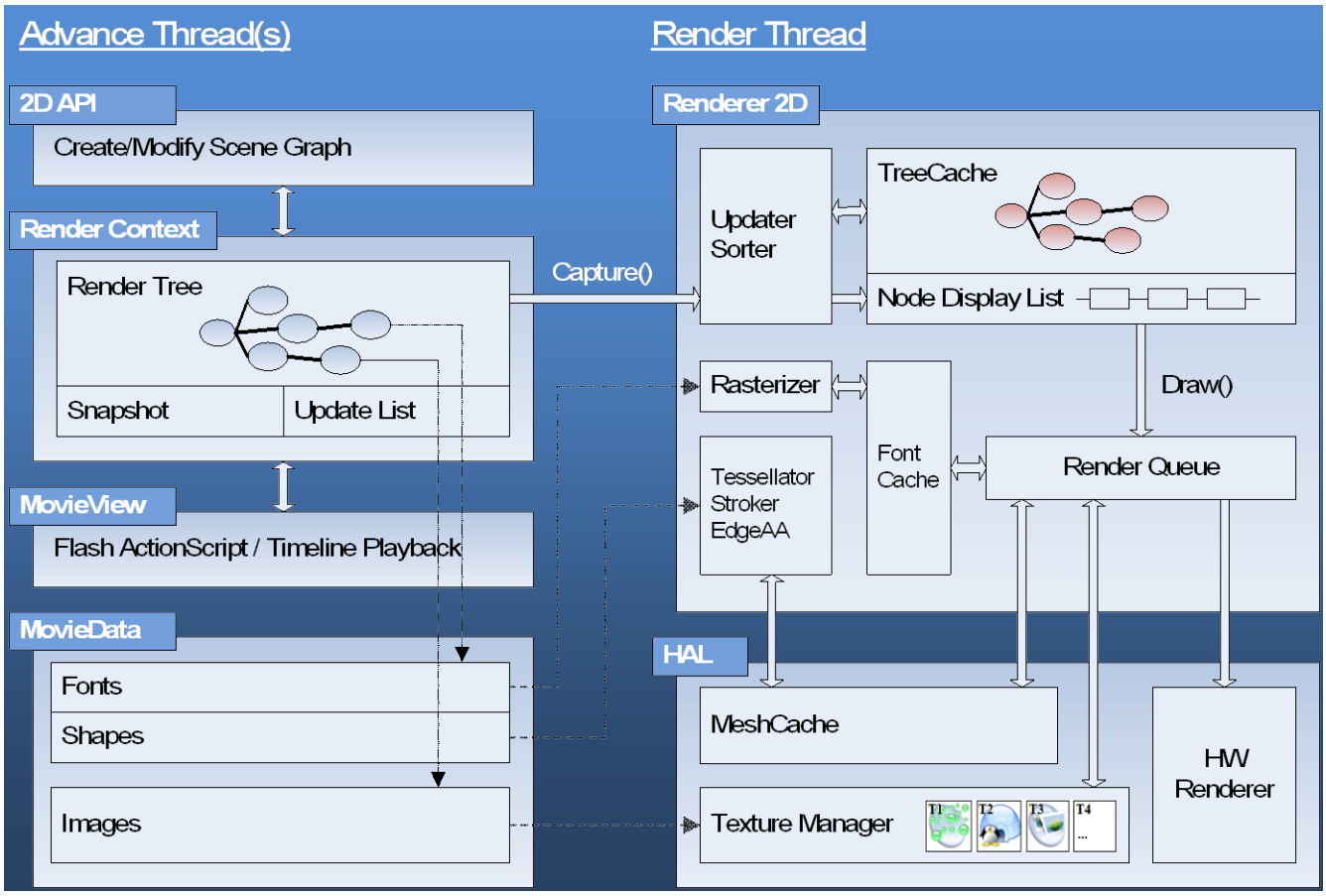

图 1：渲染设计

此图表显示与两个线程一起运行时渲染器子系统交互情况。在左边，Movie 和 MovieDef 对象分别代表 Scaleform 电影实例及其内部共享数据。Render Context 是管理 Render Tree 的渲染子系统的一个前端；它负责渲染树可访问的 2D 场景图表快照和更改列表。在 Scaleform 中，渲染上下文包含在 Movie 内，而且不需要最终用户访问。

在右边，渲染线程维护两个对象：Render::Renderer2D 和 Render::HAL。Renderer2D 是渲染引擎的核心；它包含 Display 方法，该方法用来将电影快照输出到屏幕。Render::HAL 代表硬件抽象层摂，并且是一个针对每个平台具有不同实现的抽象类。Render HAL 负责执行图形命令并管理像顶点和纹理缓冲区这样的资源。

从图表中可以看出，大部分渲染系统是在渲染线程上维护和执行的。具体说来，渲染线程负责所有下述任务：

- 维护渲染树的分类、缓存版本并在来自新快照的更改到达时对该版本进行更新。
- 维护 Font Cache 并在有新字形需要渲染时对 Font Cache 进行更新。
- 维护 Mesh Cache，其中包含细化的矢量数据的顶点和索引缓冲区。
- 管理一个 Render Queue，该 Render Queue 能够实现高效缓冲区更新/锁定管理，并在渲染时最大限度地减少 CPU 停机。
- 通过 HAL API 向设备发出图形命令。

在多数情况下，使缓存管理保持在渲染器线程上可提高性能，因为这可以将主线程解放出来处理其它工作密集型任务，例如，ActionScript 或游戏 AI。把缓存保持在渲染线程上还可以减少同步开销和内存使用，因为这消除了为在线程之间复制顶点和/或字形数据时留出额外缓冲区的必要。

5 渲染子系统说明

5.1 TextureCache

TextureCache 系统处理卸载纹理资源 的问题。该系统尽管可用于各种平台，但它更适用于内存受限的系统，例如手机平台，目的是为了减少总体内存占用量。不过，只有在 HAL::InitHAL 内隐式创建 TextureManager 对象时，默认情况下才在 iOS、Android 和 PS Vita 平台上初始化 TextureCache 系统。TextureCache 仅适用于图像数据小于纹理数据的图像，即压缩的图像必须解码成 GPU 使用的未压缩的 RGBA 颜色数据。其中包括 PNG、JPEG 和 Flash 内部图像格式。TextureCache 不适用于经过压缩并可由 GPU 直接使用的图像，也不适用于未经压缩的图像 – 一旦从图像中复制其纹理数据，这些图像的数据就会被自动卸载。

假如将 TextureManager 分配并传递到 HALInitParams 之中，TextureManager 就会将 TextureCache 作为其构造函数的最后一个参数。只提供 TextureCache 的一个实现，名为 TextureCacheGeneric，不过，用户可以创建一个派生的 TextureCache 实现。如果是这种情况，就必须分配 TextureManager 并传递一个派生的 TextureCache 实例，然后必须通过 HALInitParams 将 TextureManager 传递到 InitHAL 之中。

TextureCache 的一般实现采用一种从内存中驱逐出纹理的简单 LRU 策略。而且，只有给适用的纹理分配一个阈值之后才会驱逐纹理。一旦达到该阈值（默认情况下为 8MB），就会驱逐 LRU 纹理，直到再次低于限值。如果无法达到限值，因为单个帧中使用纹理内存的不止一个阈值，就会发出警告。运行时，可以用 TextureCache 的 SetLimitSize 函数调节该限值。系统从不会驱逐正常渲染所必需的纹理。

6 利用自定义数据进行渲染

用户偶尔希望通过与在 Scaleform 标准版本中可用的方法不同的方法来渲染某些形状。这可能是达到改变的顶点转换或特殊片段着色器填充效果的一种方法。

6.1 拦截形状渲染

一般情况下，只有电影内小的形状子集需要特殊渲染效果。在 ActionScript 中，这些对象是通过使用 AS2 扩展属性 “rendererString” 和 “rendererFloat” 或者 AS3 扩展方法 “setRendererString” 和 “setRendererFloat” 进行标识的。请参阅 “AS2/AS3 扩展” 参考，了解有关其使用方法的详细信息。渲染一个电影剪辑之前，在该电影剪辑上设置一个字符串和/或浮点（Float），会导致通过 HAL::PushUserData 函数将 UserDataState::Data 的一个实例推送到 Render::HAL 的 UserDataStack 成员上。整个堆栈可随时访问，因而允许使用嵌套的数据，不过，每个电影剪辑只能包含一个 “字符串” 和/或 “浮动” 。当该电影剪辑完成渲染时，就会将 UserDataState::Data 从 HAL::PopUserData 函数弹出堆栈。

6.2 自定义渲染执行示例

目前，要执行自定义渲染，同时将电影剪辑上设置的用户数据考虑在内，就必须修改 HAL。尽管为执行自定义渲染而修改 HAL 的方法不少，但本示例还是将重点放在对内部着色器系统的简单扩展上，方法是将一个 2D 位置偏移量添加到所渲染的具有相应用户数据的所有形状中。

对于本示例，创建一个 AS3 SWF，它具有一个实例名为 “m” 的电影剪辑以及下面的 AS3 代码：

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

利用 4.1 中的新的着色器脚本编写系统，我们可以容易地添加新的着色器特性。在 Src/Render/ShaderData.xml 中，添加下面突出显示的一行：

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

该行将使着色器系统使用 “Position2dOffset” 片段，以便实现处理顶点着色器中的位置数据。然后再在文件中添加 “Position2dOffset” 片段：

```
 <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>
```

此片段与 “Position2d” 片段相同 – 除此片段将统一值 ‘poffset’ 添加到顶点的转换后 x 和 y 之外。

现在，在 D3D9_Shader.cpp 中，将下列代码块添加到 ShaderInterface::SetStaticShader 的最上面：

```
// 看看我们是否有该堆栈上的用户数据。
    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)
        {
// 如果发现我们所期望的匹配的用户数据，请修改我们所用的着色器。
            vshader = (VertexShaderDesc::ShaderType)((int)vshader +
VertexShaderDesc::VS_base_Position2dOffset);
            shader  = (FragShaderDesc::ShaderType)  ((int)shader +
FragShaderDesc::FS_base_Position2dOffset);
        }
    }
```

此代码块检查 HAL 的 UserDataStack 是否具有我们要在最上面拦截的数据（有一个 “abc” 字符串，并且还有一个浮点值）。如果匹配，我们就修改要在 Position2dOffset 路径中添加的传入的着色器类型。请注意，在通过自定义构建步骤（生成新的 D3D9_ShaderDescs.h）运行 ShaderData.xml 之前，这些符号不可用。

除更改所用的着色器定义之外，我们还需要在实际渲染之前更新 ‘poffset’ 统一值。这可以通过在 ShaderInterface::Finish 最上面插入下列代码块来完成：

```
// 看看我们是否有该堆栈上的用户数据。
    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)
        {
// 将 poffset 统一值添加到统一数据中。
            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);
            }
        }
    }
```

请注意，此示例为每个 'meshCount' 设置一次统一值。对于批处理或实例化绘制（上面进行的着色器修改也支持这些绘制），meshCount 将会大于 1。运行 HAL 的修改后的版本，电影剪辑 'm' 现在应稍微向上和向右移动了一点。

6.3 对自定义绘制禁用批处理

HAL 也可以修改，这样渲染就完全在 Scaleform 着色器系统之外进行。本文不介绍进行这些修改的确切方法，而且修改方法也会因预期结果的不同而不同。

在外部进行渲染时，可以禁用批处理和实例化网格生成。这是因为在 Scaleform 之外创作的着色器并不像 Scaleform 内部的着色器一样普遍支持批处理和/或实例化。要禁用批处理绘制，只需使用 AS2 "disableBatching" 扩展成员或者使用 AS3 "disableBatching" 扩展函数。请参阅 AS2/AS3 扩展参考文档，了解这些扩展的确切语法。

7 GFxShaderMaker

7.1 一般说明

Scaleform 4.1 中引入 GFxShaderMaker，它是一个构造和编译 Scaleform 使用的着色器的实用工具。重新构造渲染器项目（如 Render_D3D9）时，它是作为一个自定义构造步骤在 ShaderData.xml 上运行的。这会产生多个文件，渲染器使用这些文件编译和/或链接渲染器。这些文件会因平台不同而不同。例如，在 D3D9 上，该实用工具创建：

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

在其他平台上，该工具链可用来直接创建一个包含着色器的库，这些着色器将会被与可执行文件链接在一起。

7.2 支持多个着色器 API 的 HAL

7.2.1 D3D9

D3D9 HAL 同时支持 ShaderModel 2.0 和 ShaderModel 3.0。默认情况下，HAL 中包含了对所有这些特征级的支持。您可以明确地删除对其中任何特征级的支持。这将会为创建着色器描述符和二进制着色器节约出额外大小。而且可以通过使用 GFxShaderMaker '-shadermodel' 的一个选项并提供一个逗点分隔式着色器模型列表来完成。例如：

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

这就会删除对 ShaderModel 2.0 的支持。如果您试图通过一个仅支持 ShaderModel 2.0 的设备调用 InitHAL，该工具会发生故障。

7.2.2 D3D1x

D3D1x HAL 支持三个特征级，它们与用来创建设备的不同等级着色器支持相对应。这些特征级包括（给 D3D11 和 D3D10.1）添加必需的前缀）：
FEATURE_LEVEL_10_0
FEATURE_LEVEL_9_3
FEATURE_LEVEL_9_1

默认情况下，HAL 中包含了对所有这些特征级的支持。您可以明确地删除对其中任何特征级的支持。这将会为创建着色器描述符和二进制着色器节约出额外大小。可以通过使用 GFxShaderMaker '-featurelevel' 的一个选项并提供一个逗点分隔式必需特征级列表来完成。例如：

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

这就会删除对 FEATURE_LEVEL9_3 的支持。如果您试图通过一个使用 FEATURE_LEVEL_9_3 的设备调用 InitHAL，它就会使用支持的下一个最低特征级（在此情况下为 FEATURE_LEVEL_9_1）。