Autodesk®
Scaleform®

Scaleform 4.3 渲染器線程指南

本文介紹Scaleform 4.3 中的多線程渲染配置。

作者： Michael Antonov
版本： 1.05
上次编辑时间： 2013 年 4 月 8日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

How to Contact Autodesk Scaleform:

Document	Scaleform 4.3 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 包括一項新的多線程渲染器設計，它可實現在不同線程上同時執行電影推進(Advance)和渲染（顯示）邏輯，從而提高整體性能並高效地將Scaleform SDK
成到多線程應用程序和遊戲引擎之中。本文介紹這一新的渲染器設計的結構，概述為實現多線程渲染而在Scaleform 4.0 中對API 進行的更改，並提供在線程之間安全地達到Scaleform 渲染所必需的示例代碼。

2 多線程渲染器更改

許多API從Scaleform3.x更改為Scaleform4，以利用新的設計和多線程渲染。此列表突出顯示被更改的主要渲染器方面，它們是使用多線程技術時需要考慮的方面。

1. **渲染器創建。**渲染器創建現在與渲染線程關聯，而且還包括創建與平台無關的Renderer2D 層。
2. **渲染緩存配置**。現在，渲染器上配置了網格緩存和字體緩存，而未在GFxLoader 上配置。
3. **紋理資源創建**。紋理創建需要線程安全，或者由渲染線程提供服務。
4. **渲染循環。**對電影進行的渲染現在是通過將MovieDisplayHandle
   傳遞給渲染線程然後在那里通過Renderer2D::Display 對其進行渲染來完成的。
5. **電影關閉**。 GFx::Movie 發布操作可能需要由渲染線程來提供服務。
6. **自定義渲染器。**重新設計了與平台無關的API，以便為硬件頂點緩衝區緩存和分批處理提供支持。GRenderer 在Scaleform 3.X 中的角色現在由Render::HAL 類進行處理。

2.1 渲染器創建

Scaleform 端渲染器初始化涉及創建兩個類：Render::Renderer2D 和Render::HAL。 Renderer2D 是一種矢量圖形渲染器實現，提供渲染電影對象時使用的Display 方法。 Render::HAL 是Renderer2D 使用的一個"硬件抽象層"接口；其特定實現位於特定平台的命名空間，例如Render::D3D9::HAL 和Render::PS3::HAL。這兩個命名空間都是通過與下述情況相似的邏輯創建的：

```
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 提供針對個別平台的初始化參數。在這當中， pDevice 與PresentParams 提供予由用戶設定的硬件的相應信息、而renderThreadId
識別渲染器使用的渲染線程。ThreadCommandQueue是渲染線程中可能需要實現的一個接口實例，本文將對此進行詳細介紹。

Renderer2D和HAL類並非線程安全的，而是設計為僅在渲染線程上使用。它們通常是由於初始化請求而在渲染線程上創建的，或者在渲染線程受阻時在主線程上創建。

2.2 渲染緩存配置

在Scaleform4中，網格緩存和字形緩存都與Renderer2D關聯，而且是在渲染線程上維護的。此行為不同於 Scaleform 3.x，因為在那裡，這些狀態是在 Loader上配置的。緩存是在通過調用InitHAL
來配置關聯的Render::HAL 時創建的；而在調用ShutdownHAL 時釋放其內存。

網格緩存存儲頂點和索引緩衝區數據，而且通常直接從視頻內存直接分配；它是由系統特定的邏輯，作為Render::HAL 的組成部分進行管理的。網格緩存內存以大塊分配，HAL
初始化時預分配第一塊，其後可以按一個固定限額增加。網格緩存配置如下：

```
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);
```

在這裡，MemReserve 定義分配的緩存內存的初始數量，而MemLimit
定義最大可分配數量。如果這兩個值相同，初始化後就不會發生任何緩存分配。 MeshCache
按塊增大（由MemGranularity 指定），一旦存在的LRU 緩存內容大於LRUTailSize
所規定的，MeshCache就會縮小。StagingBufferSize是一個用於構建批量的系統內存緩衝區；它可以在控制台上設置為64K（那裡的默認值），但在PC 上必須大於此值，因為它還充當一個二級緩存。

字形緩存用於柵格化字體，以便於高效地自紋理繪製字體。字形緩存動態更新，並具有一個初始化時確定的固定大小。字形緩存配置如下：

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

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

用於字形緩存的配置設置實際上與Scaleform 3.3 相同。 TextureWidth、TextureHeight 和NumTextures 定義將要分配的僅字形紋理的大小和數量。MaxSlotHeight指定將為單個字形分配的最大槽高（單位：像素）。

對於要修改的初始設置，調用InitHAL之前，兩個緩存均應配置。如果在InitHAL 之後調用SetParams，關聯的緩存會被刷新，而且內存會被重新分配。

2.3 紋理資源創建

Scaleform 4.0 包括一個重新設計的GFx::ImageCreator 以及一個新的Render::TextureManager 類，它們允許從視頻內存直接加載圖像。與Scaleform 的早期版本相似，ImageCreator 是安裝在GFx::Loader 上的狀態對象，用來自定義圖像數據加載，具體情況如下：

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

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

默認ImageCreator取構造函數中的一個可選TextureManager指針；如果指定了一個紋理管理器且該紋理管理器不會丟失數據，就會將圖像內容直接加載到紋理內存之中。 TextureManager 對像是由Render::HAL::GetTextureManager()方法返回的，後者應在渲染線程上調用。上面的代碼示例假定一個渲染線程類提供GetTextureManager訪問器方法，該方法只能在初始化渲染後調用。此時，此相關性表明：如果要將內容直接加載到紋理之中，在加載內容之前需要初始化渲染。如果未指定紋理管理器，就會先將圖像數據加載到系統內存中，因而可在初始化渲染器之前發生。

與Render::HAL的渲染方法不同的是，TextureManager方法（如CreateTexture）必須是線程安全的，因為它們可以從不同加載線程調用。此線程安全通過以下兩種方法之一實現：

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。

- 通過線程安全的方式（例如在控制台上）實現紋理內存分配，或者在用D3DCREATE_MULTITHREADED 旗標包裝設備時利用D3D9 實現。
- 通過借助ThreadCommandQueue接口（在Render::HAL構造函數中指定）將紋理創建指派給渲染線程。

此第二種方法意味著，假如用非多線程D3D 設備進行多線程渲染，就必須實現ThreadCommandQueue並根據紋理創建需要為其提供服務。若非如此，一旦從第二個線程調用，紋理創建就會阻滯。請參閱Platform::RenderHALThread 類，了解示例實現情況。

2.4 渲染循環

通過多線程渲染，傳統渲染循環被拆分成兩個部分：推進線程執行的推進/輸入處理邏輯，以及執行繪圖命令（如繪製幀摄）的渲染線程循環。Scaleform 4.0 APIs通過定義可安全地傳遞到渲染線程的GFx::MovieDisplayHandle來實現多線程技術。GFx::Movie對象本身並不應傳遞到渲染線程，因為它本質上並非線程安全的，而且不再包含Display 方法。

一般電影渲染過程可分為以下幾個步驟：

1. 初始化。由GFx::MovieDef::CreateInstance 創建GFx::Movie
   之後，用戶通過設置視窗、獲取顯示句柄並將其傳遞到渲染線程來對GFx::Movie 進行配置。
2. 主線程處理循環。在每個幀上，用戶處理輸入並調用Advance 來執行時間線和ActionScript
   處理。在對電影進行一切Invoke/DirectAccess API 修改之後調用Advance
   是最後一次調用非常重要，因為那是對幀拍攝現場快照的地方。在Advance
   之後，電影將一個Scaleform 繪製幀請求提交給渲染線程。
3. 渲染。一旦渲染線程收到繪製幀請求，它就抓取摄MovieDisplayHandle
   的最新捕獲的快照，並在屏幕上對其進行渲染。

以下參考將詳細說明這些步驟。

```
    //------------------------------------------------------------------------
// 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();
```

儘管該邏輯的大部分一目了然，但仍然需要注意以下幾個細節：

- WaitForOutstandingFrame –此幫助函數確保渲染線程不會超前一個幀以上。這不僅節約CPU處理時間，而且確保幀渲染線程排隊命令不會超越與電影內容同步。在單線程模式中，這並不必要，因為可以直接在Advance 之後執行繪製幀邏輯。
- MovieDisplay.NextCapture:要想"抓取"Advance捕獲的最新快照，就必須進行此調用，從而使其保持最新，以便進行渲染。一旦電影不再可用，此函數就會返回false（假），在這種情況下，不會繪製任何東西。

2.5 電影關閉

當與多線程渲染一起使用時，如果在毀壞渲染器對象之前執行GFx::Movie Release操作，則可能需要有渲染線程為其提供服務；要向電影的內部數據結構發布任何渲染線程參考，就必需此服務。默認情況下，將會通過前面提及的Render::ThreadCommandQueue 接口從Movie解構函數對電影關閉命令進行排隊。開發者必須實現ThreadCommandQueue 接口以確保正確關閉。

作為在渲染線程上為命令排隊提供服務的替代選擇，推進線程可使用電影關閉輪詢接口先發起關閉，然後等待其完成之後再發布電影。為此，請先調用Movie::ShutdownRendering(false)以發起關閉，然後使用Movie::IsShutdownRenderingComplete來針對每個幀測試其是否已經完成。在渲染線程上，NextCapture將處理髮起的關閉並返回false。一旦處理關閉，推進線程就可無阻滯地執行電影發布。

3 多線程渲染概念

使用Scaleform 進行的多線程渲染至少包含兩個線程– 推進線程和渲染線程。

推進線程一般是創建電影實例、處理其輸入處理以及調用GFx::Movie::Advance 以執行ActionScript 和時間線動畫的線程。應用程序中可以有不止一個Advance線程；不過，每部電影一般只由一個Advance線程訪問。在多數Scaleform示例中，Advance線程也是主要應用程序控制線程，根據需要向渲染線程發出命令。

渲染線程的主要任務是渲染屏幕上的圖形，渲染線程通過向圖形設備輸出命令（一般經由像Render::HAL這樣的應用層）來完成此操作。要最大限度地減少通信開銷，Scaleform渲染線程處理像添加電影摂和繪製幀摂這樣的宏命令，而不是像繪製三角形摂這樣的宏命令。渲染線程還管理網格緩存和字形緩存。

渲染線程可以鎖步工作，或者獨立於控制推進線程工作，具體情況如下：

- 利用鎖步渲染，控制線程向渲染線程發出繪製幀摂命令，一般在電影推進處理之間。這兩個線程並行工作，渲染線程繪製前一個幀，而推進線程則處理下一個幀。在多核系統上，這會提高總體性能，同時還保持線程框架之間的一對一關係。
- 對於獨立工作模式，渲染線程在繪製幀時不受可能以不同幀速運行的推進線程影響。對電影的修改會在修改完成之後不久顯示出來，這是渲染線程抽出時間繪製已更新的電影快照的下一個時間段。

Scaleform
Player應用程序和Scaleform示例利用鎖步渲染模式，這一方面因為設置更方便，另一方面因為遊戲中更為普遍。

3.1 電影快照

在多線程渲染期間，Advance線程可在渲染線程出於顯示目的訪問內部電影狀態的同時修改內部電影狀態。由於兩種線程對數據的非確定性訪問會導致崩潰和/或壞幀，新的渲染器利用快照確保渲染線程始終看到一個連貫的幀。

在Scaleform
中，電影快照是電影在給定時間點的一種被捕獲狀態。默認情況下，在調用GFx::Movie::Advance 結束時自動捕獲快照，也可以通過調用GFx::Movie::Capture來明確地捕獲。一旦捕獲一個快照，渲染線程就可以使用其幀狀態，但與電影的動態狀態分開存儲。繪製幀時，渲染線程在一個電影句柄上調用 NextCapture 以抓取最近的快照並將其用於渲染。

此設計對於開發者來說具有多種含義：

- 當與渲染同時執行時，Movie::Advance 和輸入處理邏輯並不需要一個關鍵節。
- 可以按照不同頻率調用Capture 和NextCapture。如果進行多次連續的Capture 調用，就會合併電影快照。如果沒有足夠的Capture 調用，同一幀就可能會渲染多次。
- 調用Capture 或Advance 方法之前，渲染線程看不到由於輸入、Invoke 或Direct Access API 對Movie 進行的修改。

最後一點說明Scaleform 4.0 與早期版本之間的一個重要行為差異。儘管在Scaleform 3.3 中開發者可以調用Display 遵循的Movie::Invoke 來呈現對屏幕的更改，但現在要呈現更改他們還必須調用Capture 或Advance。一般情況下，不需要這樣做，因為可以在調用Advance 之前把輸入處理和Invoke 調用組合在一起。

3.2 保留描繪狀態

Render::Renderer2D::Display 進行設備調用來在渲染設備上渲染電影的幀。出於性能原因、不保留混合模式和紋理存儲設置等各種設備狀態、 而且調用 Render::Renderer2D::Display 之後設備的狀態會發生變化。有些應用程式可能受到負面影響。最直接的做法是在調用之前先保存設備狀態資訊調用之後再恢復這些資訊。遊戲引擎在Scaleform渲染後重新初始化必要的狀態資訊這樣能達到優越的性能。

4 渲染器設計

在大體上了解多線程渲染之後，我們現在可以看看Scaleform 4.0 渲染設計圖表。

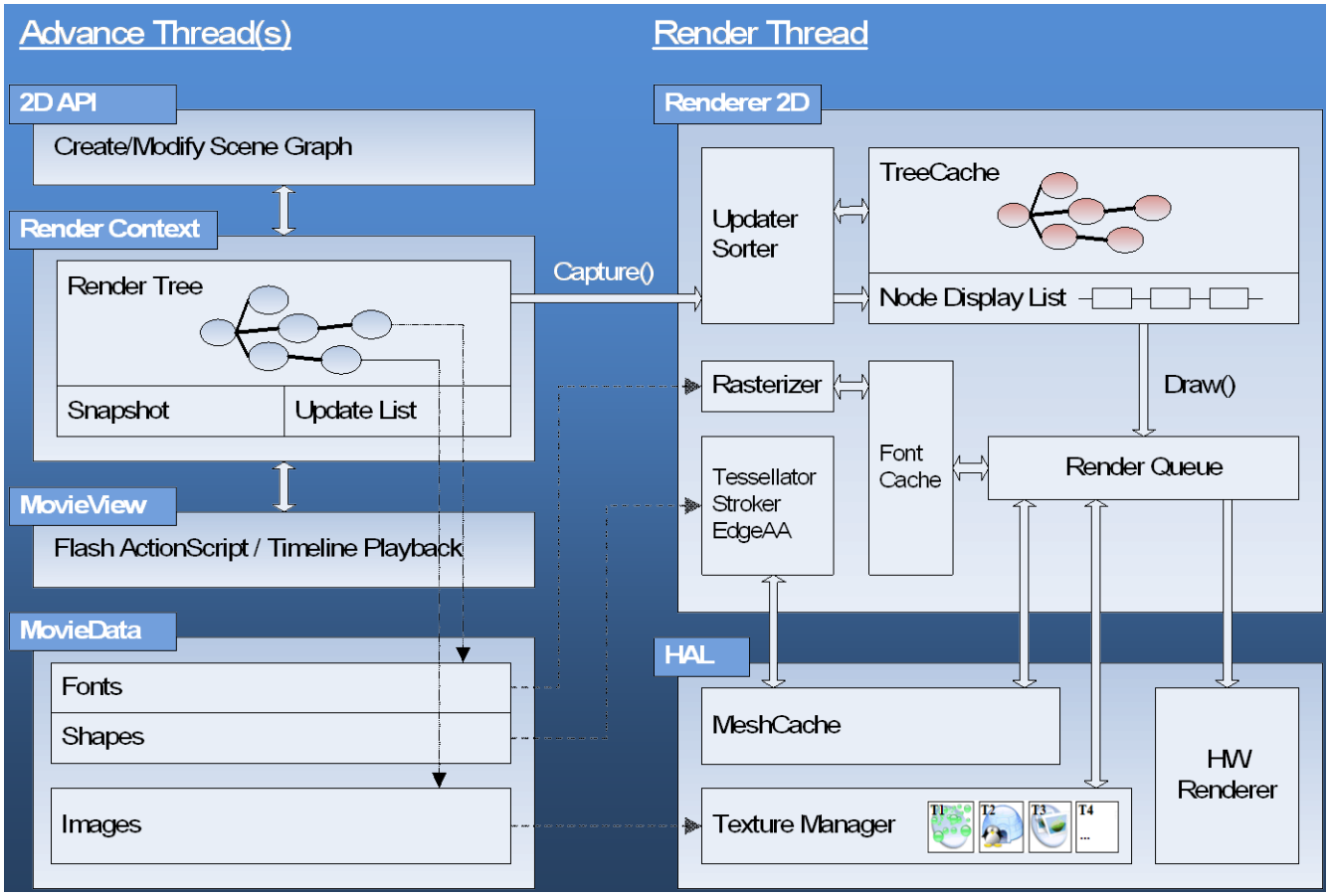

图 1：渲染设计

此圖表顯示與兩個線程一起運行時渲染器子系統交互情況。在左邊，Movie 和MovieDef 對象分別代表Scaleform 電影實例及其內部共享數據。 Render Context 是管理Render Tree 的渲染子系統的一個前端；它負責渲染樹可訪問的2D 場景圖表快照和更改列表。在Scaleform 中，渲染上下文包含在Movie 內，而且不需要最終用戶訪問。

在右邊，渲染線程維護兩個對象：Render::Renderer2D 和Render::HAL。 Renderer2D 是渲染引擎的核心；它包含Display 方法，該方法用來將電影快照輸出到屏幕。 Render::HAL 代表硬件抽象層摂，並且是一個針對每個平台具有不同實現的抽像類。 Render HAL 負責執行圖形命令並管理像頂點和紋理緩衝區這樣的資源。

從圖表中可以看出，大部分渲染系統是在渲染線程上維護和執行的。具體說來，渲染線程負責所有下述任務：

- 維護渲染樹的分類、緩存版本並在來自新快照的更改到達時對該版本進行更新。
- 維護Font Cache 並在有新字形需要渲染時對Font Cache 進行更新。
- 維護Mesh Cache，其中包含細化的矢量數據的頂點和索引緩衝區。
- 管理一個Render Queue，該Render Queue
  能夠實現高效緩衝區更新/鎖定管理，並在渲染時最大限度地減少CPU 停機。
- 通過HAL API 向設備發出圖形命令。

在多數情況下，使緩存管理保持在渲染器線程上可提高性能，因為這可以將主線程解放出來處理其它工作密集型任務，例如，ActionScript或遊戲AI。把緩存保持在渲染線程上還可以減少同步開銷和內存使用，因為這消除了為在線程之間複製頂點和/或字形數據時留出額外緩衝區的必要。

5 渲染子系统说明

5.1 TextureCache

TextureCache 系統處理卸載紋理資源
的問題。該系統儘管可用於各種平臺,但它更適用于記憶體受限的系統,例如手機平臺,目的是為了減少總體記憶體佔用量。不過,只有在 HAL::InitHAL 內隱式創建 TextureManager 物件時,預設情況下才在 iOS、Android 和 PS Vita 平臺上初始化 TextureCache 系統。TextureCache
僅適用于圖像資料小於紋理資料的圖像,即壓縮的圖像必須解碼成 GPU 使用的未壓縮的 RGBA
顏色資料。其中包括 PNG、JPEG 和 Flash 內部圖像格式。TextureCache 不適用於經過壓縮並可由 GPU 直接使用的圖像,也不適用於未經壓縮的圖像 –
一旦從圖像中複製其紋理資料,這些圖像的資料就會被自動卸載。

假如將 TextureManager 分配並傳遞到 HALInitParams 之中,TextureManager 就會將 TextureCache 作為其建構函式的最後一個參數。只提供 TextureCache 的一個實現,名為 TextureCacheGeneric,不過,使用者可以創建一個派生的 TextureCache 實現。如果是這種情況,就必須分配 TextureManager 並傳遞一個派生的 TextureCache 實例,然後必須通過 HALInitParams 將 TextureManager 傳遞到 InitHAL 之中。

TextureCache 的一般實現採用一種從記憶體中驅逐出紋理的簡單 LRU
策略。而且,只有給適用的紋理分配一個閾值之後才會驅逐紋理。一旦達到該閾值(預設情況下為 8MB),就會驅逐 LRU
紋理,直到再次低於限值。如果無法達到限值,因為單個幀中使用紋理記憶體的不止一個閾值,就會發出警告。運行時,可以用 TextureCache 的 SetLimitSize
函數調節該限值。系統從不會驅逐正常渲染所必需的紋理。

6 利用自定义数据进行渲染

使用者偶爾希望通過與在 Scaleform
標準版本中可用的方法不同的方法來渲染某些形狀。這可能是達到改變的頂點轉換或特殊片段著色器填滿效果的一種方法。

6.1 拦截形状渲染

一般情況下,只有電影內小的形狀子集需要特殊渲染效果。在 ActionScript 中,這些物件是通過使用 AS2 擴充屬性「rendererString」和「rendererFloat」或者 AS3 擴充方法「setRendererString」和「setRendererFloat」進行標識的。請參閱「AS2/AS3
擴展」參考,瞭解有關其使用方法的詳細資訊。渲染一個電影剪輯之前,在該電影剪輯上設置一個字串和/或浮點 (Float),會導致通過 HAL::PushUserData 函數將 UserDataState::Data 的一個實例推送到 Render::HAL 的 UserDataStack
成員上。整個堆疊可隨時訪問,因而允許使用嵌套的資料,不過,每個電影剪輯只能包含一個「字串」和/或「浮動」。當該電影剪輯完成渲染時,就會將 UserDataState::Data 從 HAL::PopUserData 函數彈出堆疊。

6.2 自定义渲染执行示例

目前,要執行自訂渲染,同時將電影剪輯上設置的使用者資料考慮在內,就必須修改
HAL。儘管為執行自訂渲染而修改 HAL
的方法不少,但本示例還是將重點放在對內部著色器系統的簡單擴展上,方法是將一個 2D
位置偏移量添加到所渲染的具有相應使用者資料的所有形狀中。 對於本示例,創建一個 AS3
SWF,它具有一個實例名為「m」的電影剪輯以及下面的 AS3 代碼:

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

利用 4.1 中的新的著色器腳本編寫系統,我們可以容易地添加新的著色器特性。在
Src/Render/ShaderData.xml 中,添加下面突出顯示的一行:

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

該行將使著色器系統使用「Position2dOffset」
片段,以便實現處理頂點著色器中的位置資料。然後再在檔中添加「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>
```

此片段與「Position2d」片段相同 – 除此片段將統一值 'poffset' 添加到頂點的轉換後 x 和 y 之外。

現在,在 D3D9_Shader.cpp 中,將下列代碼塊添加到 ShaderInterface::SetStaticShader 的最上面：

```
//看看我們是否有該堆疊上的使用者資料。
    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)
        {
// 如果發現我們所期望的匹配的使用者資料,請修改我們所用的著色器。
            vshader = (VertexShaderDesc::ShaderType)((int)vshader +
VertexShaderDesc::VS_base_Position2dOffset);
            shader  = (FragShaderDesc::ShaderType)  ((int)shader +
FragShaderDesc::FS_base_Position2dOffset);
        }
    }
```

此代碼塊檢查 HAL 的 UserDataStack 是否具有我們要在最上面攔截的資料(有一個「abc」
字串,並且還有一個浮點值)。如果匹配,我們就修改要在 Position2dOffset
路徑中添加的傳入的著色器類型。請注意,在通過自訂構建步驟(生成新的 D3D9_ShaderDescs.h)運行 ShaderData.xml 之前,這些符號不可用。

除更改所用的著色器定義之外,我們還需要在實際渲染之前更新 'poffset' 統一值。這可以通過在 ShaderInterface::Finish 最上面插入下列代碼塊來完成：

```
//看看我們是否有該堆疊上的使用者資料。
```

```
  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)
        {
//將 poffset 統一值添加到統一資料中。
            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);
            }
        }
    }
```

請注意,此示例為每個 'meshCount'
設置一次統一值。對於批次處理或具現化繪製(上面進行的著色器修改也支援這些繪製),meshCount
將會大於 1。運行 HAL 的修改後的版本,電影剪輯 'm' 現在應稍微向上和向右移動了一點。

6.3 对自定义绘制禁用批处理

HAL 也可以修改,這樣渲染就完全在 Scaleform
著色器系統之外進行。本文不介紹進行這些修改的確切方法,而且修改方法也會因預期結果的不同而不同
。 在外部進行渲染時,可以禁用批次處理和具現化網格生成。這是因為在 Scaleform
之外創作的著色器並不像 Scaleform
內部的著色器一樣普遍支援批次處理和/或具現化。要禁用批次處理繪製,只需使用 AS2
「disableBatching」 擴展成員或者使用 AS3 「disableBatching」 擴展函數。請參閱 AS2/AS3
擴展參考文檔,瞭解這些擴展的確切語法。

7 GFxShaderMaker

7.1 一般說明

Scaleform 4.1 中引入 GFxShaderMaker,它是一個構造和編譯 Scaleform
使用的著色器的公用程式。重新構造渲染器專案(如 Render_D3D9)時,它是作為一個自訂構造步驟在
ShaderData.xml
上運行的。這會產生多個檔,渲染器使用這些檔編譯和/或連結渲染器。這些檔會因平臺不同而不同。例如
,在 D3D9 上,該公用程式創建：

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

在其他平臺上,該工具鏈可用來直接創建一個包含著色器的庫,這些著色器將會被與可執行檔連結在一起。

7.2 支持多个着色器 API 的 HAL

7.2.1 D3D9

D3D9 HAL 同時支援 ShaderModel 2.0 和 ShaderModel 3.0。預設情況下,HAL
中包含了對所有這些特徵級的支援。您可以明確地刪除對其中任何特徵級的支援。這將會為創建著色器
描述項和二進位著色器節約出額外大小。而且可以通過使用 GFxShaderMaker '-shadermodel'
的一個選項並提供一個逗點分隔式著色器模型清單來完成。例如：

GFxShaderMaker.exe –platform D3D1x –shadermodel SM30

這就會刪除對 ShaderModel 2.0 的支援。如果您試圖通過一個僅支援 ShaderModel 2.0 的設備調用
InitHAL,該工具會發生故障。

7.2.2 D3D1x

D3D1x HAL 支援三個特徵級、 它們與用來創建設備的不同等級著色器支援相對應。這些特徵級包括 (給
D3D11 和 D3D10.1） 添加必需的首碼）：
FEATURE_LEVEL_10_0
FEATURE_LEVEL_9_3
FEATURE_LEVEL_9_1

預設情況下,HAL
中包含了對所有這些特徵級的支援。您可以明確地刪除對其中任何特徵級的支援。這將會為創建著色器描述項和二進位著色器節約出額外大小。可以通過使用 GFxShaderMaker '-featurelevel'
的一個選項並提供一個逗點分隔式必需特徵級清單來完成。例如：

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

這就會刪除對 FEATURE_LEVEL9_3 的支援。如果您試圖通過一個使用 FEATURE_LEVEL_9_3 的設備調用 InitHAL,它就會使用支援的下一個最低特徵級(在此情況下為 FEATURE_LEVEL_9_1)。