Autodesk®
Scaleform®

Scaleform 4.3 렌더러 쓰레딩 가이드

본 문서에서는 Scaleform 4.3 의 멀티 쓰레드 렌더링 구성에 대해 설명합니다.

작성자: Michael Antonov, Bart Muzzin

버전: 1.05

최종 수정일: 4.8.2013

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

Master Editing, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, Pixlr, Pixlr-o-matic, PolarSnap, Powered with Autodesk Technology, Productstream, ProMaterials, RasterDWG, RealDWG, Real-time Roto, Recognize, Render Queue, Retimer, Reveal, Revit, Revit LT, RiverCAD, Robot, Scaleform, Scaleform GFx, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Socialcam, Sparks, SteeringWheels, Stitcher, Stone, StormNET, TinkerBox, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, U-Vis, ViewCube, Visual, Visual LISP, Vtour, WaterNetworks, Wire, Wiretap, WiretapCentral, XSI.

Disclaimer

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

차례

1 소개

Scaleform 4.0 에는 서로 다른 쓰레드에서 동영상 진행 및 렌더링(디스플레이) 로직을 동시에 실행할 수 있는 새로운 멀티 쓰레드 렌더러가 포함되어 있어 전반적인 성능이 향상되었으며 Scaleform SDK 를 멀티 쓰레드 응용 프로그램 및 게임 엔진에 효율적으로 통합할 수 있습니다. 본 문서에서는 새로운 렌더러 설계 구조와 Scaleform 4.0 에서 멀티 쓰레드 렌더링을 적용하기 위해 수행된 API 변경 내용에 대해 설명하고 쓰레드 전반에 걸쳐 Scaleform 렌더링을 안전하게 수행하는 데 필요한 샘플 코드를 제공합니다.

2 멀티 쓰레딩 렌더러 변경 내용

새로운 설계 및 멀티 쓰레드 렌더링을 적용하기 위해 많은 API 가 Scaleform 3.x 에서 Scaleform 4.0 으로 변경되었습니다. 이 목록에서는 변경되었고, 멀티 쓰레딩으로 작업 시 고려해야 할 주요 렌더러 영역을 강조 표시합니다.

1. ***렌더러 생성.*** 렌더러 생성이 렌더러 쓰레드와 연결되었고 또한 플랫폼 독립형 Renderer2D 레이어 생성도 포함됩니다.
2. ***렌더러 캐시 구성.*** 메쉬 및 폰트 캐시가 이제 GFxLoader 가 아닌 렌더러에서 구성됩니다.
3. ***텍스처 리소스 생성.*** 텍스처 생성은 쓰레드-안전(thread-safe)하거나 렌더 쓰레드에 의해 수행되어야 합니다.
4. ***렌더링 루프.*** 동영상 렌더링은 MovieDisplayHandle 을 렌더 쓰레드로 전달하고 Renderer2D::Display 를 통해 렌더링하여 수행합니다.
5. ***동영상 종료.*** GFx::Movie 릴리스 작업은 렌더 쓰레드에 의해 수행되어야 합니다.
6. ***사용자 정의 렌더러.*** 플랫폼 독립형 API 가 하드웨어 꼭지점 버퍼 캐시와 배치 지원을 제공하도록 재설계되었습니다. Scaleform 3.X 의 GRenderer 역할이 Render::HAL 클래스에 의해 처리됩니다.

이러한 변경 내용은 코드 샘플 및 자세한 설명과 함께 아래 섹션에서 설명합니다.

2.1 렌더러 생성

Scaleform 측 렌더러 초기화에는 Render::Renderer2D 및 Render::HAL 의 두 가지 클래스 생성이 포함됩니다. Renderer2D 는 동영상 오브젝트를 렌더링하는 데 사용되는 Display 메소드를 제공하는 벡터 그래픽 렌더러 구현입니다. Render::HAL 은 Renderer2D 에 의해 사용되는 "하드웨어 추상화 레이어" 인터페이스이며, 관련 구현은 Render::D3D9::HAL 및 Render::PS3::HAL 과 같이 플랫폼 관련 네임스페이스에 위치합니다. 두 가지 모두 다음과 유사한 로직을 통해 생성됩니다.

```
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);
```

이 예에서 렌더러에서 사용된 렌더 쓰레드를 식별하는 동안, HALInitParams 클래스는 설정된 디바이스에 따라 pDevice 및 PresentParams 같은 플랫폼 별 초기화 파라미터들을 제공합니다. renderThreadId 는 렌더러에서 사용된 렌더링 쓰레드를 식별합니다. ThreadCommandQueue 는 렌더 쓰레드에서 구현해야 하는 인터페이스 인스턴스이며 본 문서의 후반부에서 자세하게 설명합니다.

Renderer2D 및 HAL 클래스는 쓰레드-안전(thread-safe)하지 않으며 렌더 쓰레드에서만 사용되도록 설계되었습니다. 이러한 클래스는 초기화 요청으로 인해 렌더 쓰레드 또는 렌더 쓰레드 차단 상태에서 메인 쓰레드에서 일반적으로 생성됩니다.

2.2 렌더 캐시 구성.

Scaleform 4.0 의 경우 메시 및 문자 캐시는 Renderer2D 와 연결되고 렌더 쓰레드에서 유지 관리됩니다. 이 동작은 이러한 상태가 GFxLoader 에서 구성되는 Scaleform 3.x 와 다릅니다. 연결된 Render::HAL 이 InitHAL 에 대한 호출에 의해 구성될 때 생성되며 해당 메모리는 ShutdownHAL 호출 시 삭제됩니다.

메시 캐시는 꼭지점 및 인덱스 버퍼 데이터를 저장하고 일반적으로 비디오 메모리에서 직접 할당됩니다. 또한 Render::HAL 의 일부로 시스템 관련 로직에 의해 관리됩니다. 메시 캐시 메모리는 대용량 청크에서 할당되며 첫 번째 청크는 HAL 초기화 시 사전 할당되고 고정 한계까지 증가될 수 있습니다. 메시 캐시는 다음과 같이 구성됩니다.

```
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);
```

여기서, MemReserve 는 할당된 캐시 메모리의 초기 크기를 정의하고 MemLimit 은 최대 배치 가능한 크기를 정의합니다. 이러한 값이 모두 같을 경우 초기화 이후 캐시 할당이 수행되지 않습니다. MeshCache 가 MemGranularity 에 의해 지정된 블록에서 증가하고 LRUTailSize 에 의해 제공되는 것보다 LRU 캐시 콘텐츠가 많을 경우 감소합니다. StagingBufferSize 는 배치 구성을 위해 사용되는 시스템-메모리 버퍼이며 콘솔에서 64K 로 설정될 수 있지만(기본값) 2 차 캐시 역할도 수행하기 때문에 PC 에서 더 커야 합니다.

문자 캐시는 폰트 래스터화에 사용되며 텍스처와 다르게 그릴 수 있습니다. 문자 캐시는 동적으로 업데이트되며 초기화 시점에서 결정된 고정 크기를 가집니다. 다음과 같이 구성됩니다.

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

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

문자 캐시의 구성 설정은 가상적으로 Scaleform 3.3 과 동일합니다. TextureWidth, TextureHeight 및 NumTextures 는 할당될 알파 전용 텍스처의 크기 및 개수를 정의합니다. MaxSlotHeight 는 단일 문자에 대해 할당될 슬롯의 최대 높이를 픽셀 단위를 지정합니다.

초기 설정을 수정하려면 InitHAL 가 호출되기 전에 두 캐시를 구성해야 합니다. InitHAL 이후 SetParams 를 호출할 경우 관련 캐시가 플러싱되고 메모리가 재할당됩니다.

2.3 텍스처 리소스 생성

Scaleform 4.0 에는 이미지를 비디오 메모리에서 직접 로드할 수 있는 재설계된 GFx::ImageCreator 및 새로운 Render::TextureManager 클래스가 포함되어 있습니다. 이전 버전의 Scaleform, ImageCreator 와 유사하며 GFx::Loader 에 설치된 상태 오브젝트가 다음과 같이 이미지 데이터 로딩을 사용자 정의합니다.

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

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

기본 ImageCreator 가 생성자의 옵션 TextureManager 포인터를 가져옵니다. 텍스처 관리자가 지정되고 데이터를 분실하지 않은 경우 이미지 콘텐츠가 직접 텍스처 메모리로 로드됩니다. TextureManager 오브젝트가 Render::HAL::GetTextureManager() 메소드에 의해 반환되며 이 메소드는 렌더 쓰레드에서 호출되어야 합니다. 위의 샘플 코드는 렌더 쓰레드 클래스가 GetTextureManager 접근자 메소드를 제공하는 것으로 가정하며, 이 메소드는 렌더를 초기화한 후에만 호출할 수 있습니다. 이때 이 종속성은 텍스처로 직접 로드해야 하는 경우 콘텐츠를 로딩하기 전에 렌더링을 초기화해야 함을 의미합니다. 텍스처 관리자를 지정하지 않을 경우 이미지 데이터가 시스템 메모리에 먼저 로드되기 때문에 렌더러 초기화 이전에 발생될 수 있습니다.

Render::HAL 렌더링 메소드와 달리, CreateTexture 와 같이 TextureManager 메소드는 다른 로딩 쓰레드로부터 호출될 수 있기 때문에 쓰레드-안전(thread-safe) 상태이어야 합니다. 이 쓰레드 안전성은 다음 두 가지 방법 중 하나로 구현됩니다.

- 예를 들어 콘솔 또는 장치가 D3DCREATE_MULTITHREADED 플래그를 통해 생성된 경우 D3D9 와 같이 텍스처 메모리 할당을 쓰레드-안전(thread-safe) 방법으로 구현합니다.
- ThreadCommandQueue 인터페이스를 통해 텍스처 생성을 렌더 쓰레드에 위임하며, Render::HAL 생성자에 의해 지정됩니다..

두 번째 접근 방법은 ***비 멀티 쓰레드 D3D 장치로 멀티 쓰레드 렌더링을 수행할 경우 ThreadCommandQueue 를 구현해야 하며*** 텍스처 생성에 필요한 경우 제공해야 합니다. 이렇게 하지 않을 경우 2 차 쓰레드로부터 호출될 때 텍스처 생성이 차단됩니다. 샘플 구현에 대해서는 to Platform::RenderHALThread 클래스를 참조하십시오.

2.4 렌더링 루프

멀티 쓰레드 렌더링을 통해 기존 렌더링 루프가 다음 두 부분으로 분할됩니다. advance 쓰레드에 의해 실행되는 advance/입력 프로세싱 로직과 "draw frame"과 같이 드로잉 명령을 실행하는 렌더-쓰레드 루프. Scaleform 4.0 APIs 는 안전하게 렌더링 쓰레드로 전달될 수 있는 GFx::MovieDisplayHandle 을 정의하여 멀티 쓰레딩을 적용합니다. GFx::Movie 오브젝트는 속성상 쓰레드-안전 상태가 아니며 더 이상 Display 메소드를 포함하지 않기 때문에 렌더링 쓰레드로 전달되지 않아야 합니다..

일반적인 동영상 렌더링 프로세스는 다음 단계로 구분될 수 있습니다.

1. ***초기화.*** GFx::Movie 가 GFx::MovieDef::CreateInstance 에 의해 생성된 후 사용자가 뷰포트를 설정하고, 디스플레이 핸들을 가져와서 렌더 쓰레드로 전달하여 구성합니다.
2. ***메인 쓰레드 프로세싱 루프.*** 모든 프레임에서 사용자가 입력을 처리하고 Advance 를 호출하여 타임라인 및 ActionScript 프로세싱을 수행합니다. Advance 호출이 동영상에 대한 Invoke/DirectAccess API 수정 이후 마지막 호출이 되어야 합니다. 즉 프레임에 대해 장면 스냅샷을 캡처한 위치입니다. Advance 이후 동영상은 Scaleform 드로우 프레임 요청을 렌더 쓰레드로 제출합니다.

3. ***렌더링.*** 렌더 쓰레드가 드로우 프레임 요청을 수신하면 MovieDisplayHandle 에 대해 가장 최근에 캡처한 스냅샷을 "가져와서" 화면에 렌더링합니다.

아래 참조에서 이러한 단계에 대해 자세하게 설명합니다.

```
  //--------------------------------------------------------------------------
  // 1. 영상 초기화

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

  // 영상을 만든 후 시점을 설정하고
// 디스플레이 핸들을 가져옵니다.
  pMovie->SetViewport(width, height, 0,0, width, height);
  hMovieDisplay = pMovie->GetDisplayHandle();

  // 핸들을 렌더링 쓰레드에 전달합니다. 전달 방식은 엔진에 따라 다릅니다. Scaleform Player에서
// 핸들을 렌더링 쓰레드에 전달하려면 내부 함수 호출을 큐에 추가합니다.
  pRenderThread->AddDisplayHandle(hMovieDisplay);


  //--------------------------------------------------------------------------
  // 2. 처리 루프


// 입력과 처리 작업은 주 쓰레드에서 처리합니다.
// ExternalInterface와 같은 영상 콜백도 여기서 미리 호출합니다.
  float deltaT = ...;
  pMovie->HandleEvent(...);
  pMovie->Advance(deltaT);

  // 이전 프레임의 렌더링이 끝날 때까지 기다린 다음
// 프레임 그리기 요청을 큐에 추가합니다.
  pRenderThread->WaitForOutstandingDrawFrame();
  pRenderThread->DrawFrame();


  //--------------------------------------------------------------------------
  // 3. 렌더링 – 렌더링 쓰레드 DrawFrame 로직
  Ptr<Render::Renderer2D> pRenderer = ...;

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

대부분의 로직이 보이는 대로 설명할 수 있지만 세부적인 의미는 다음과 같습니다.

- WaitForOutstandingFrame – 이 도움 함수는 렌더 쓰레드가 둘 이상의 프레임에 의해 진행되지 않도록 합니다. 이를 통해 CPU 처리 시간을 절약하고 프레임 렌더 쓰레드 대기 명령이 동영상 콘텐츠와 동기화에서 이탈되지 않도록 합니다. 단일 쓰레드 모드에서는, Advance 이후 바로 드로우 프레임 로직을 실행할 수 있기 때문에 이 작업이 필요 없습니다.
- MovieDisplay.NextCapture – 이 호출은 Advance 에 의해 캡처된 가장 최근 스냅샷을 "가져와서" 렌더링을 최신 상태로 유지하기 위해 필요합니다. 이 함수는 동영상을 더 이상 사용할 수 없는 경우 false 를 반환하며 이 경우 아무것도 드로우하지 않습니다.

2.5 동영상 종료

멀티 쓰레드 렌더링 사용 시 렌더러 오브젝트가 삭제되기 전에 실행된 경우 렌더 쓰레드에 의해 GFx::Movie Release 작업을 제공해야 합니다. 모든 렌더 쓰레드 참조를 동영상에 내부적일 수 있는 데이터 구조에 릴리스하려면 이러한 서비스를 제공해야 합니다. 기본적으로 동영상 종료 명령은 앞서 언급한 Render::ThreadCommandQueue 인터페이스를 통해 Movie 소멸자로부터 대기합니다. 개발자는 적절하게 종료하기 위해 ThreadCommandQueue 인터페이스를 구현해야 합니다.

렌더 쓰레드에서 명령 대기열을 제공하기 위한 대안으로, advance 쓰레드가 동영상 종료 폴링 인터페이스를 사용하여 먼저 종료를 실행한 다음 동영상을 릴리스하기 전에 완료될 때까지 기다립니다. 이를 위해 먼저 Movie::ShutdownRendering(false)을 호출하여 종료를 실행한 다음 Movie::IsShutdownRenderingComplete 를 사용하여 모든 프레임 종료에 대해 테스트합니다. 렌더 쓰레드에서, NextCapture 는 실행된 종료를 처리하고 false 를 반환합니다. 종료가 처리되면 advance 쓰레드가 차단 없이 동영상 릴리스를 실행할 수 있습니다.

3 멀티 쓰레드 렌더링 개념

Scaleform 에서의 멀티 쓰레드 렌더링에는 *advance 쓰레드* 및 *rendering 쓰레드*의 최소 두 가지 쓰레드가 포함됩니다.

Advance 쓰레드는 일반적으로 동영상 인스턴스를 생성하고, 입력 프로세싱을 처리하고, GFx::Movie::Advance 를 호출하여 ActionScript 및 타임라인 애니메이션을 실행하는 쓰레드입니다. 응용 프로그램에 둘 이상의 Advance 쓰레드가 있을 수 있지만 각 동영상은 일반적으로 하나의 advance 쓰레드에 의해서만 액세스됩니다. 또한 대부분의 Scaleform 샘플에서, advance 쓰레드는 메인 응용 프로그램 제어 쓰레드이며, 필요에 따라 렌더 쓰레드에 명령을 내립니다.

렌더링 쓰레드의 주요 역할은 화면에서 그래픽을 렌더링하는 것이며, 일반적으로 Render::HAL 과 같이 추상화 레이어를 통해 그래픽 장치에 명령을 전달합니다. 통신 오버헤드를 최소화하기 위해 Scaleform 렌더 쓰레드는 "draw triangles"와 같은 마이크로 명령이 아닌 "add movie" 및 "draw frame"과 같은 매크로 명령을 처리합니다. 또한 렌더 쓰레드는 메시 및 문자 캐시를 관리합니다.

렌더 쓰레드는 다음과 같이 잠금-단계로 작동하거나 제어 advance 쓰레드와 독립적으로 작동할 수 있습니다.
잠금 단계 렌더링을 통해 제어 쓰레드는 일반적으로 동영상 advance 처리 중에 "draw frame" 명령을 렌더 쓰레드에 발행합니다. 두 개의 쓰레드는 이전 프레임을 드로잉하는 렌더링 쓰레드와 함께 작동하는 반면에 advance 쓰레드는 다음 프레임을 처리합니다. 멀티 코어 시스템의 경우 이 기능으로 인해 전반적인 성능이 향상되는 동시에 쓰레드 프레임 간의 1 대 1 관계가 유지됩니다.
독립적인 작동 모드를 사용하여 렌더 쓰레드는 advance 쓰레드와 독립적으로 프레임을 드로우하며, 잠재적으로 다른 프레임 속도에서 실행됩니다. 동영상에 대한 수정 내용은 완료된 후 잠시 표시되고, 다음에 렌더 쓰레드가 업데이트된 동영상 스냅샷을 드로잉하기 위해 나타납니다.

Scaleform Player 응용 프로그램 및 Scaleform 샘플은 설치하기 쉽고 게임에서 훨씬 일반적인 잠금-단계 렌더링 모드를 사용합니다.

3.1 동영상 스냅샷

멀티 쓰레드 렌더링 동안 Advance 쓰레드는 렌더 쓰레드가 디스플레이 목적으로 액세스할 때 동시에 내부 동영상 상태를 수정할 수 있습니다. 두 개의 쓰레드에 의한 비결정적 데이터 액세스로 인해 충돌 및/또는 프레임 손상이 발생하기 때문에 새 렌더러는 스냅샷을 사용하여 렌더 쓰레드가 항상 일관된 프레임을 표시하는지 확인합니다.

Scaleform 에서, 동영상 스냅샷은 제공된 시점에서 동영상을 캡처한 상태입니다. 기본적으로 스냅샷은 GFx::Movie::Advance 호출 종료 시점에 자동으로 캡처되며 또한 GFx::Movie::Capture 를 호출하여 명시적으로 캡처할 수 있습니다. 스냅샷을 캡처하면 프레임 상태가 렌더 쓰레드에서 사용 가능하게 되며 동영상의 동적 상태와는 별도로 저장할 수 있습니다. 프레임을 드로잉하는 경우 렌더 쓰레드가 동영상 핸들에서 NextCapture 를 호출하여 가장 최근 스냅샷을 가져온 후 렌더링에 사용합니다.

이 설계는 개발자에게 여러 가지 의미를 제공합니다.
Movie::Advance 및 입력 프로세싱 로직은 렌더링과 동시에 실행하는 경우 중요 섹션이 필요 없습니다.
Capture 및 NextCapture 호출을 서로 다른 주기에서 호출할 수 있습니다. Capture 호출을 여러 번 연속적으로 수행할 경우 동영상 스냅샷이 병합됩니다. Capture 호출이 충분하지 않을 경우 동일한 프레임이 여러 차례 렌더링될 수 있습니다.
입력, Invoke 또는 Direct Access APIs 로 인해 Movie 에 적용된 수정 사항은 Capture 또는 Advance 메소드가 호출될 때까지 렌더 쓰레드에 의해 표시되지 않습니다.

마지막 포인트에서 Scaleform 4.0 및 이전 버전 간의 중요 동작 차이점을 보여 줍니다. Scaleform 3.3 의 경우 개발자가 Movie::Invoke 를 호출한 다음 Display 를 호출하여 화면에 변경 내용을 표시할 수 있으며 변경 내용을 표시하기 위해 Capture 또는 Advance 를 호출해야 합니다. 일반적으로 이것은 입력 프로세싱에서 필요 없으며 Advance 에 대한 호출 전에 Invoke 호출을 함께 그룹화할 수 있습니다.

3.2 렌더링 상태 보호

Render::Renderer2D::Display 는 렌더링 장비 상에서 동영상의 프레임을 렌더링하기 위해 장비를 호출합니다. 성능을 위해 혼합 모드 및 텍스처 저장소 설정과 같은 다양한 장비의 상태는 보존되지 않으며 장비의 상태는 Render::Renderer2D::Display 로 호출된 후 달라집니다. 이로 인해 몇몇 어플리케이션은 악영향을 받을 수 있다. 가장 직접적인 해결책은 Display 호출 이전에 디바이스 상태를 저장하고 나중에 이를 복구하는 것이다. Scaleform 렌더링 후 필요한 상태로 게임 엔진을 재초기화 함으로써 더 나은 성능을 얻을 수 있다.

4 렌더러 설계:

멀티 쓰레드 렌더링의 일반적인 이해를 위해 Scaleform 4.0 렌더 설계도를 살펴 보겠습니다.

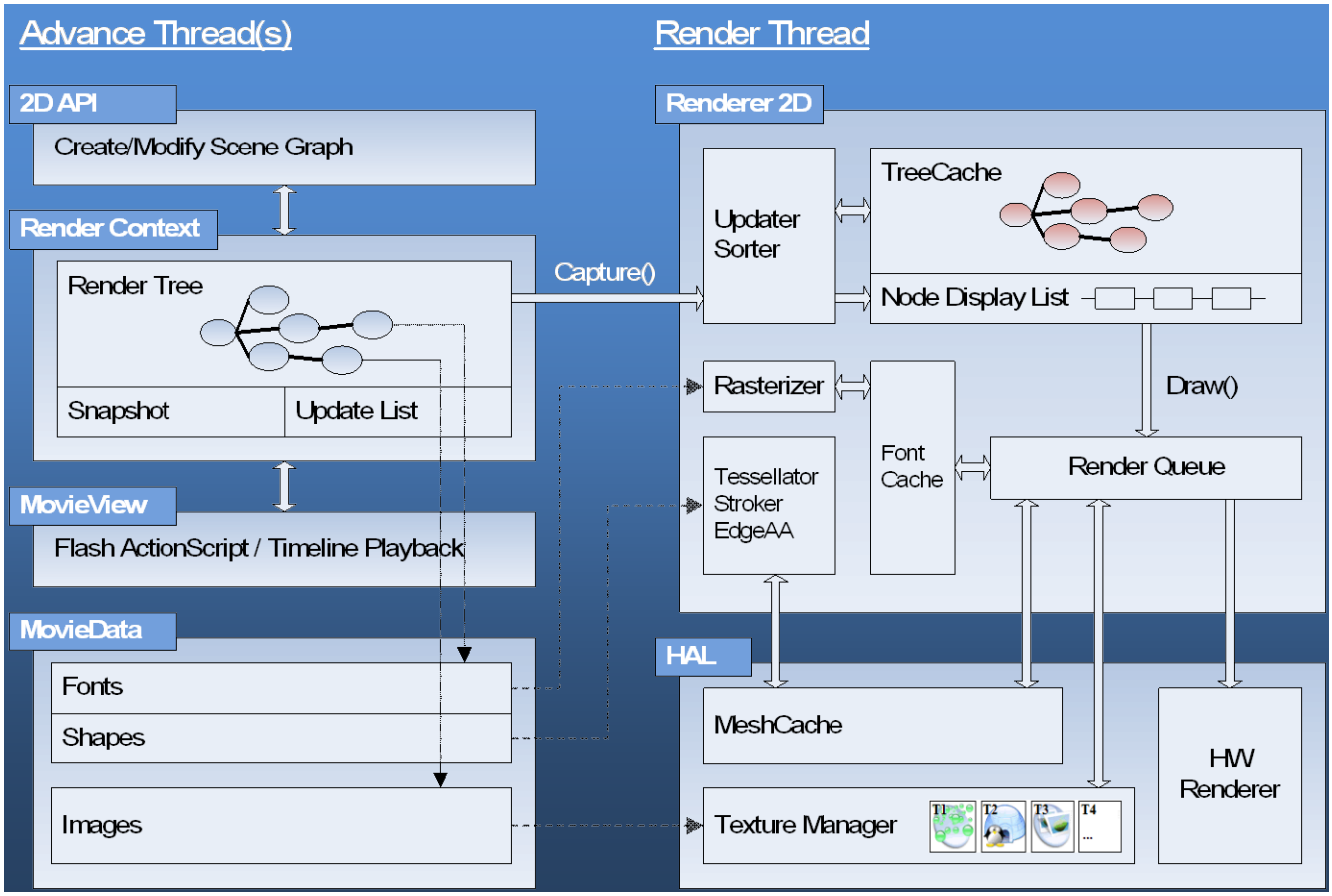

그림 1: 렌더 설계

그림에서는 두 개의 쓰레드로 실행 시 렌더러 하위 시스템의 상호 작용을 보여 줍니다. 좌측에는, MovieView 및 MovieData 오브젝트가 Scaleform 동영상 인스턴스 및 내부 공유 데이터를 각각 표시합니다. Render Context 는 Render Tree 를 관리하는 렌더링 하위 시스템에 대한 프론트 엔드이며, 렌더 트리에서 액세스 가능한 2D 장면 그래프 스냅샷 및 변경 목록을 담당합니다. Scaleform 에서, 렌더 컨텍스트는 Movie 내에 포함되어 있으며 최종 사용자가 액세스할 필요 없습니다.

우측에서, 렌더 쓰레드가 두 개의 오브젝트인 Render::Renderer2D 및 Render::HAL 을 관리합니다. Renderer2D 는 렌더 엔진의 핵심이며 동영상 스냅샷을 장면에 출력하는 데 사용되는 Display 메소드가 포함되어 있습니다. Render::HAL 은 "하드웨어 추상화 레이어"를 의미하며 각 플랫폼에 대해 다른 구현을 가진 추상화 클래스입니다. Render HAL 은 그래픽 명령 실행과 꼭지점 및 텍스처 버퍼와 같은 리소스 관리를 담당합니다.

그림에 표시된 것처럼, 대부분의 렌더링 시스템은 렌더 쓰레드에서 유지되고 실행됩니다. 특히, 렌더 쓰레드는 다음 모든 기능을 담당합니다.

- 정렬된 캐시 버전의 렌더 트리 유지 관리 및 새 스냅샷에 변경 내용이 있는 경우 업데이트.
- Font Cache 유지 관리 및 새 문자를 렌더링해야 하는 경우 업데이트.
- Mesh Cache 유지 관리, 여기에는 모자이크식 벡터 데이터의 꼭지점 및 인덱스 버퍼가 포함됩니다.
- 효율적인 버퍼 업데이트/잠금 관리를 활성화하고 렌더링하는 동안 CPU 정지를 최소화하는 Render Queue 관리.
- HAL APIs 를 통해 장치에 그래픽 명령 발행.

대부분의 경우 렌더 쓰레드에서 캐시 관리 로직을 유지하면 ActionScript 또는 게임 AI 와 같이 다른 부하 집중적 작업에 대한 메인 쓰레드가 제거되기 때문에 성능이 향상됩니다. 또한 렌더링 쓰레드에서 캐시를 유지하면 쓰레드 간의 꼭지점 및/또는 문자 데이터를 복사하는 데 필요한 추가 버퍼가 필요 없기 때문에 동기화 오버헤드 및 메모리 사용이 감소합니다.

5 렌더링 하위 시스템 설명

5.1 TextureCache

TextureCache 시스템은 텍스처 리소스를 언로드하는 작업을 담당합니다. TextureCache 시스템은 모든 플랫폼에서 사용할 수 있으나 모바일 플랫폼과 같이 주로 메모리가 제한된 시스템에서 전체 메모리 공간을 줄이기 위한 목적으로 사용되도록 고안되었습니다. 단, TextureCache 는 HAL::InitHAL 내에서 TextureManager 개체를 암시적으로 생성하는 동안 iOS, Android 및 PS Vita 플랫폼 상에서만 기본적으로 초기화됩니다. TextureCache 는 이미지 데이터가 텍스처 데이터보다 작은 이미지, 즉, GPU 에서 사용하려면 압축되지 않은 RGBA 컬러 데이터로 디코딩되어야 하는 압축 이미지에만 사용할 수 있습니다. 여기에는 PNG, JPEG 및 Flash 내부 이미지 형식이 포함됩니다. 해당 사항은 압축되었으며 GPU 에서 직접 사용할 수 있는 이미지 또는 압축되지 않은 이미지의 경우에는 적용되지 않습니다. 이와 같은 이미지에 대한 해당 데이터는 이미지로부터 텍스처 데이터가 복사되는 즉시 자동으로 언로드됩니다.

TextureManager 가 할당되었으며 HALInitParams 로 전달된 경우, TextureCache 가 생성자의 마지막 매개변수가 되어야 합니다. TextureCache 는 TextureCacheGeneric 이라는 이름으로 한번만 구현할 수 있으나 사용자는 파생된 TextureCache 구현을 생성할 수 있습니다. 이 경우 TextureManager 가 할당되고 파생된 TextureCache 인스턴스가 전달되어야 하며 TextureManager 는 HALInitParams 를 통해 InitHAL 에 전달되어야 합니다.

TextureCache 는 일반적으로 구현될 시 메모리에서 텍스처를 제거하기 위해 단순한 LRU 전략을 사용합니다. TextureCache 는 적용 가능한 텍스처의 임계값이 할당되기 전까지 텍스처를 제거하려고 시도하지 않습니다. 임계값(기본적으로 8MB)에 도달하면 TextureCache 는 해당 값이 임계값 밑으로 내려갈 때까지 LRU 텍스처 제거를 시도합니다. 텍스처 메모리의 임계값보다 많은 메모리가 단일 프레임에서 사용되어 해당 값이 임계값 밑으로 내려갈 수 없는 경우 경고 메시지가 표시됩니다.

임계값은 TextureCache 의 SetLimitSize 기능을 사용하여 런타임 중에 조정할 수 있습니다. 본 시스템은 올바른 렌더링에 필요한 텍스처를 절대 제거하지 않습니다.

6 사용자 정의 데이터로 렌더링

이따금 Scaleform 표준 배포 버전의 메소드과는 다른 특정 셰이프을 렌더링하고자 하는 경우가 있습니다. 꼭지점 변환 또는 특수 단편 셰이더 채우기 효과를 제공합니다.

6.1 교차 셰이프 렌더링

보통의 경우 영상에서 특수 렌더링 효과가 필요한 셰이프는 소규모입니다. 이러한 오브젝트는 AS2 확장 속성인 "rendererString"과 "rendererFloat" 또는 AS3 확장 메소드인 "setRendererString"과 "setRendererFloat"을 사용하여 ActionScript 에서 확인할 수 있습니다. AS2/AS3 확장 사용법은 해당 자료를 참조하십시오. Movie clip 에서 문자열 및/또는 float 를 설정하면 해당 영상이 렌더링되기 전에 UserDataState::Data 인스턴스가 the HAL::PushUserData 함수를 통해 Render::HAL 의 UserDataStack 멤버로 삽입됩니다. 전체 스택은 언제든지 접근이 가능하여 중첩된 데이터를 사용할 수 있습니다. 그러나 movie clip 은 각각 하나의 문자열 및/또는 Float 만 포함할 수 있습니다. Movie clip 의 렌더링이 완료되면 UserDataState::Data 는 HAL::PopUserData 함수를 통해 스택으로부터 빠져나옵니다.

6.2 사용자 정의 렌더링 예시

현재 버전에서 movie clip 의 사용자 데이터 세트를 고려한 사용자 정의 렌더링을 수행하려면 HAL 을 수정해야 합니다. HAL 이 사용자 지정 렌더링을 수행하도록 수정하는 방법에는 여러 가지가 있으나 이 예시에서는 해당 사용자 데이터를 가지고 렌더링한 모든 셰이프에 2D 위치 오프셋을 추가하여 내부 셰이더 시스템을 확장하는 방법을 다루겠습니다.
이 예시에서는 인스턴스 이름이 'm'인 movie clip 과 다음의 AS3 코드를 가진 AS3 SWF 을 생성합니다.

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

4.1 버전의 새 셰이드 스크립트 시스템을 사용하면 셰이더 기능을 간단히 추가할 수 있습니다.

Src/Render/ShaderData.xml 에서 강조 표시된 행을 추가합니다.

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

이 행은 셰이더 시스템에서 꼭지점 셰이더의 위치 데이터를 처리하는 가능성으로 "Position2dOffset" snippet 을 사용하도록 합니다. 파일 후반부에 다음 "Position2dOffset" snippet 을 추가합니다.

```
 <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>
```

이 snippet 은 변환된 꼭지점 x 와 y 값에 'poffset'이라는 균일(uniform)값을 추가한다는 점을 제외하면 "Position2d" snippet 과 동일합니다.

D3D9_Shader.cpp 의 ShaderInterface::SetStaticShader 최상단에 다음 블록을 추가합니다.

```
  // 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);
        }
    }
```

이 블록은 HAL 의 UserDataStack 이 상단("abc"라는 문자열과 float 값을 함께 가짐)에서 교차하려는 데이터를 가지고 있는지 여부를 확인합니다. 일치하는 경우 Position2dOffset 경로를 추가하도록 받는 셰이더 유형을 수정합니다. 이러한 심볼은 ShaderData.xml 가 사용자 정의 빌드 단계를 모두 거쳐 새로운 D3D9_ShaderDescs.h 를 생성한 뒤에 사용할 수 있습니다.

사용된 셰이더 정의를 변경하는 것 외에도 실제 렌더링 전에 'poffset' 균일값(uniform)을 업데이트해야 합니다. 이 작업은 ShaderInterface::Finish 최상단에 다음 블록을 삽입하여 수행할 수 있습니다.

```
   // 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);
            }
        }
    }
```

이 예시에서는 'meshCount' 당 한 번씩 균일값을 설정하였으며 meshCount 는 배치 또는 인스턴스된 draw(위 셰이더 수정에서도 지원됨)의 meshCount 보다 큰 값을 가집니다. 수정된 버전의 HAL 을 실행시키면 movie clip 'm'이 윗 방향 및 오른쪽 방향으로 조금씩 이동됩니다.

6.3 사용자 정의 그리기의 배치 비활성화

HAL 은 또한 Scaleform 셰이더 시스템 외부에서 렌더링을 수행하도록 수정될 수 있습니다. 이러한 수정 방법은 이 문서에서는 다루지 않으며 이러한 방법은 의도와 경우에 따라 달라질 수 있습니다. 외부적으로 렌더링 수행 시에는 배치 및 인스턴스 매쉬 생성을 비활성화하는 것이 좋습니다. 그 이유는 Scaleform 외부에서 작성된 셰이더는 Scaleform 내부 셰이더와 같은 방식으로 배치 및/또는 인스턴스 작업을 지원하지 않는 경우가 많기 때문입니다. 배치 그리기는 간단히 AS2 "disableBatching" 확장 멤버 또는 AS3 "disableBatching" 확장 함수를 사용하여 비활성화할 수 있습니다. AS2/AS3 확장에 대한 정확한 구문은 해당 확장에 대한 문서를 참조하십시오.

7 GFxShaderMaker

7.1 일반 설명

Scaleform 4.1 에 도입된 GFxShaderMaker 는 Scaleform 이 사용하는 셰이더를 구성하고 컴파일하는 유틸리티입니다. 렌더러 프로젝트(예로 Render_D3D9)를 재빌드할 때에는 ShaderData.xml 상에서 일반 빌드 단계로 실행됩니다. 이를 통해 몇몇 파일이 생성되며, 이들을 렌더러가 컴파일하거나 렌더러와 연결하는 데 사용합니다. 이들은 플랫폼에 따라 다릅니다. 예를 들어 D3D9 에서는 다음이 생성됩니다.

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

다른 플랫폼에서는 도구 사슬을 사용하여 셰이더를 포함하는 라이브러리를 직접 생성할 수 있으며, 이것이 실행 파일에 연결되게 됩니다.

7.2 다중 셰이더 API 를 지원하는 HAL

7.2.1 D3D9

D3D9 HAL 은 ShaderModel 2.0 및 ShaderModel 3.0 을 모두 지원합니다. 기본으로, 이들 모든 기능 수준에 대한 지원이 HAL 에 포함됩니다. 이들 중 어느 하나에 대한 지원을 명시적으로 제거할 수 있습니다. 그러면 셰이더 서술자 및 바이너리 셰이더를 생성하기 위한 추가 크기를 절약할 수 있습니다. 이는 GFxShaderMaker '-shadermodel'에 대한 옵션을 사용하고, 셰이더 모델의 콤마로 분리된 목록을 제공함으로써 수행할 수 있습니다. 예제:

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

이는 ShaderModel 2.0 에 대한 지원을 제거합니다. ShaderModel 2.0 만을 지원하는 장치로 InitHAL 을 호출하려고 시도하면, 실패하게 됩니다.

7.2.2 D3D1x

D3D1x HAL 은 장치가 생성할 수 있는 상이한 수준의 셰이더 지원에 해당하는 3 가지 기능 수준을 지원합니다. 이들은 다음과 같습니다(D3D11 및 D3D10.1 에 대해 필요한 접두어 추가)

```
FEATURE_LEVEL_10_0
FEATURE_LEVEL_9_3
FEATURE_LEVEL_9_1
```

기본으로, 이들 모든 기능 수준에 대한 지원이 HAL 에 포함됩니다. 이들 중 어느 하나에 대한 지원을 명시적으로 제거할 수 있습니다. 그러면 셰이더 서술자 및 바이너리 셰이더를 생성하기 위한 추가 크기를 절약할 수 있습니다. 이는 GFxShaderMaker '-featurelevel'에 대한 옵션을 사용하고, 셰이더 모델의 콤마로 분리된 목록을 제공함으로써 수행할 수 있습니다. 예제:

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

이는 FEATURE_LEVEL9_3 에 대한 지원을 제거하게 됩니다. FEATURE_LEVEL_9_3 을 사용하는 장치로 InitHAL 을 호출하려고 시도하면, 다음으로 낮은 지원 수준(이 경우에는 FEATURE_LEVEL_9_1)을 사용하게 됩니다.