Autodesk®
Scaleform®

Scaleform 4.3 レンダラー スレッディング ガイド

Scaleform 4.3 でのマルチスレッド型レンダリングの構成について説明したドキュメントです。

作者　Michael Antonov, Bart Muzzin

バージョン　1.05
最終更新日　2013 年 4 月 8 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

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 Fax	(301) 446-3200 (301) 446-3199

目 次

1 はじめに 1
2 マルチスレッド型レンダラーへの変更 1
2.1 レンダラーの生成 2
2.2 レンダー キャッシュの設定 2
2.3 テクスチャ リソースの生成 4
2.4 レンダリング ループ 5
2.5 ムービーのシャットダウン 7
3 マルチスレッド型レンダリングのコンセプト 8
3.1 ムービーのスナップショット 9
3.2 レンダリング状態の保存 9
4 レンダラーのアーキテクチャ 11
5 サブシステム記述子のレンダー 13
5.1 TextureCache 13
6 カスタムデータのレンダリング 14
6.1 形状レンダリングの阻止 14
6.2 カスタムレンダリングの例 14
6.3 カスタムドローのバッチを無効にする 16
7 GFxShaderMaker 17
7.1 概要 17
7.2 複数シェーダーAPI をサポートする HAL 17
7.2.1 D3D9 17
7.2.2 D3D1x 18

1 はじめに

Scaleform 4.0 では、ムービーのアドバンス処理およびレンダリング（表示）処理を異なるスレッドで同時に実行し、全体性能の向上を図るとともにマルチスレッド型アプリケーションおよびゲームエンジンへの効率的な Scaleform SDK インテグレーションを実現する、マルチスレッド型のレンダリング アーキテクチャを新たに採用しています。このドキュメントでは、新しいレンダリング アーキテクチャの構造、マルチスレッド型レンダリングに対応するために適用された Scaleform 4.0 API の変更点、および Scaleform のレンダリングを複数スレッドに安全に展開するためのサンプル コードについて説明しています。

2 マルチスレッド型レンダラーへの変更

新しいアーキテクチャおよびマルチスレッド型レンダリングに対応するために、Scaleform 3.x から Scaleform 4.0 で複数の API が変更されました。レンダラーに関連してマルチスレッドでの開発で考慮すべき主な変更点を示します。

1. **レンダラーの生成**：レンダラーの生成はレンダー スレッドに関連付けされるとともに、プラットフォーム非依存の Renderer2D レイヤーの生成が含まれました。
2. **レンダラー キャッシュの構成**：メッシュ キャッシュおよびフォント キャッシュは、GFxLoader 上ではなく、レンダラー上に構成されます。
3. **テクスチャ リソースの生成**：テクスチャ生成は、スレッド セーフとして行われるか、レンダー スレッドによってサービスされなければなりません。
4. **レンダリング ループ**：ムービーのレンダリングは、レンダー スレッドに MovieDisplayHandle を渡したのち、Renderer2D::Display を介して行われます。
5. **ムービーのシャットダウン**：GFx::Movie の解放動作はレンダー スレッドによってサービスされなければなりません。
6. **カスタム レンダラー**：ハードウェア バーテックス バッファ キャッシュとバッチ処理をサポートするために、プラットフォーム非依存 API の再設計を行いました。Scaleform 3.x での GRenderer が担っていた役割は、新たに Render::HAL クラスが担います。

これら変更点の詳細をサンプルコードとともに以下に示します。

2.1 レンダラーの生成

Scaleform 側でのレンダラー初期化によって Render::Renderer2D と Render::HAL のふたつのクラスが生成されます。Renderer2D はベクター グラフィクス レンダラー実装で、ムービー オブジェクトのレンダーに用いられる Display メソッドを与えます。Render::HAL は Renderer2D で使用される「ハードウェア抽象化レイヤー」インタフェースです。具体的な実装は Render::D3D9::HAL や Render::PS3::HAL のようなプラットフォーム固有の名前空間に配置されます。両者とも以下のようなロジックを使って生成されます。

```
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);
```

この例では、PresentParams、hWnd、および pDevice によって構成済みデバイスとそのウインドウを記述しています。また、renderThreadId によってレンダラーで使われるレンダリング スレッドを指定しています。ThreadCommandQueue はレンダー スレッド内に実装されていなければならないインタフェース インスタンスで、詳細は後述します。

Renderer2D と HAL クラスはスレッド セーフではなく、レンダー スレッドのみで使用されることを前提に設計されています。これらクラスは、通常は初期化要求によってレンダー スレッド上に生成されるか、レンダー スレッドがブロックされた状態でメイン スレッド上に生成されます。

2.2 レンダー キャッシュの設定

Scaleform 4.0 では、メッシュ キャッシュとグリフ キャッシュの両方は Renderer2D に関連付けられ、レンダー スレッド上で維持されます。このような挙動は、GFxLoader 上でそれらの状態が設定される Scaleform 3.x の挙動とは異なります。InitHAL の呼び出しによって、関連付けられている Render::HAL が設定されると、キャッシュが生成されます。ShutdownHAL が呼び出されるとメモリは解放されます。

メッシュ キャッシュにはバーテックスとインデックス バッファ データが格納されます。通常はビデオメモリから直接割り当てられ、Render::HAL の一部であるシステム固有ロジックによって管理されます。メッシュ キャッシュメモリは大きなチャンク内に割り当てられ、そのうち最初のチャンクは HAL 初期化に割り当てられ、決められた上限まで成長します。メッシュ キャッシュは次のように設定します。

```
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);
```

ここで、MemReserve は割り当てるキャッシュ メモリの初期サイズを定義し、MemLimit は配置可能な最大サイズを定義します。両方の値が同じ場合は初期化後にキャッシュは割り当てられません。MeshCache は MemGranularity で指定されたブロックを単位として成長し、LRUTailSize で与えられた以上の LRU キャッシュ コンテンツが存在すると縮小します。StagingBufferSize はバッチの構築に使用されるシステムメモリ バッファで、コンソールから 64K（デフォルト値）に設定できますが、二次キャッシュとしての役割も担うため、PC よりも大きくなければなりません。

グリフ キャッシュは、テクスチャよりも効率的な描画を目的として、フォントのラスタライズに使用されます。グリフ キャッシュは動的に更新され、サイズは初期化時に決まり固定です。設定手順は次のとおりです。

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

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

グリフ キャッシュの構成設定は Scaleform 3.3 と実質的に同じです。TextureWidth、TextureHeight、および NumTextures によって、割り当てるアルファ オンリー テクスチャのサイズと個数を定義しています。また MaxSlotHeight によって、単一グリフに割り当てるスロットの最大高さをピクセルを単位として定義しています。

初期設定を変更する場合は、InitHAL が呼び出される前に両方のキャッシュを設定しておかなければなりません。仮に InitHAL のあとに SetParams が呼び出されると、関連キャッシュはフラッシュされ、メモリが再度割り当てられます。

2.3 テクスチャ リソースの生成

Scaleform 4.0 には、再設計した GFx::ImageCreator と、ビデオメモリからイメージを直接読み込める新しい Render::TextureManager クラスが設けられています。Scaleform の従来のバージョンと同様に、ImageCreator は GFx::Loader にインストールされる状態オブジェクトであり、次のようにイメージデータの読み込みをカスタマイズします。

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

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

デフォルトの ImageCreator はコンストラクタ内にあるオプションの TextureManager ポインタを取得します。テクスチャ マネージャが指定されていて、かつ、データの消失が許されない場合、イメージ コンテンツはテクスチャ メモリ内に直接読み込まれます。レンダー スレッド上で呼び出されるべき Render::HAL::GetTextureManager() メソッドによって TextureManager オブジェクトが返されます。上記のサンプル コードでは、レンダーの初期化後にのみ呼び出せる GetTextureManager アクセサ メソッドは、レンダー スレッド クラスによって提供されていると仮定しています。その時点において、このような依存性は、コンテンツがテクスチャ内に直接読み込まれる場合は、コンテンツを読み込む前にレンダリングの初期化が必要であることを意味します。テクスチャ マネージャが指定されない場合、イメージ データはシステムメモリに最初に読み込まれるため、レンダラー初期化の前に読み込みが可能です。

CreateTexture のような TextureManager メソッドは、Render::HAL のレンダリング メソッドとは違って、別々の読み込みスレッドから呼び出される可能性があるため、スレッド セーフでなければなりません。スレッド セーフにするには次のふたつの方法があります。

- テクスチャ メモリの割り当てを、コンソール上で、あるいはデバイスが D3DCREATE_MULTITHREADED フラグで生成された場合は D3D9 を使うなどして、スレッド セーフな方法で実装する
- テクスチャ生成を、Render::HAL コンストラクタで指定される ThreadCommandQueue インタフェースを介して、レンダー スレッドに委任する

二番目の方法は、マルチスレッド非対応の **D3D デバイスを使ってマルチスレッド レンダリングを行うには、テクスチャの生成に対して、ThreadCommandQueue の実装と必要に応じてそのサービスが必要**であることを意味します。これが行われない場合、テクスチャ生成はセカンダリ スレッドから一度でも呼び出されるとブロックされます。サンプル実装については Platform::RenderHALThread を参照してください。

2.4 レンダリング ループ

マルチスレッド レンダリングでは、従来のレンダリング ループはふたつの要素に分割されます。ひとつはアドバンス スレッドで実行されるアドバンス/入力処理ロジックで、もうひとつは「draw frame」のような描画コマンドを実行するレンダー スレッド ループです。Scaleform 4.0 API では、レンダリング スレッドにセーフティに渡す GFx::MovieDisplayHandle を定義することで、マルチスレッディングに対応しています。GFx::Movie オブジェクトは本質的にスレッド セーフではなく、また Display メソッドを含んでいないことから、GFx::Movie オブジェクト自体をレンダリング スレッドに渡してはなりません。

一般的なムービー レンダリング プロセスは次のステップに落とし込むことができます。

1. **初期化** – GFx::MovieDef::CreateInstance によって GFx::Movie が生成されたのち、ユーザーによってビューポイントを設定し、ディスプレイ ハンドルを取得してレンダー スレッドに渡します。
2. **メイン スレッド処理ループ** – それぞれのフレームで、ユーザーは入力を処理するとともに Advance を呼び出してタイムラインおよび ActionScript 処理を実行します。Advance は、フレームのシーンのスナップショットが取得されるため、Invoke/DirectAccess API によるムービー変更後の最後に呼び出される点に注意してください。Advance 後にムービーは Scaleform 描画フレーム要求をレンダー スレッドに発行します。

3. **レンダリング** – レンダー スレッドが描画フレーム要求を受信すると、レンダー スレッドは MovieDisplayHandle でキャプチャされた最新のスナップショットを「取り込み」、画面上にレンダリングします。

これらのステップは以下のサンプル コードで詳しく記述しています。

```
//-------------------------------------------------------------------------
// 1. ムービーの初期化

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

// ムービーの生成とディスプレイハンドルの取得後に
// ビューポートを設定します。
pMovie->SetViewport(width, height, 0,0, width, height);
hMovieDisplay = pMovie->GetDisplayHandle();

// レンダースレッドにハンドルを渡します。この処理はエンジンによって異なります。
// Scaleform Playerでは内部関数コールのキューイングによって対応しています。
pRenderThread->AddDisplayHandle(hMovieDisplay);


//-------------------------------------------------------------------------
// 2. 処理ループ

// 入力とメインスレッド上の処理を扱います。ExternalInterfaceのような
// ムービーコールバックもここでAdvanceからコールされます。
float deltaT = ...;
pMovie->HandleEvent(...);
pMovie->Advance(deltaT);

// ひとつ前のフレームのレンダリングの完了を待つともに、
// 描画フレーム要求をキューイングします。
pRenderThread->WaitForOutstandingDrawFrame();
pRenderThread->DrawFrame();


//-------------------------------------------------------------------------
// 3. レンダリング - レンダースレッドのDrawFrameロジック
Ptr<Render::Renderer2D> pRenderer = ...;

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

ほとんどのロジックの意味は自明と思われますが、二点ほど詳細を付記しておきます。

- WaitForOutstandingFrame – レンダー スレッドが 2 フレーム以上進まないようにするヘルパー関数です。CPU の処理時間を節約するとともに、コマンドがキューイングされるフレーム レンダー スレッドとムービー コンテンツとの同期が外れないように保証します。シングル スレッド モードでは Advance 後に描画フレームロジックを直接実行できるため必要ありません。
- MovieDisplay.NextCapture – Advance によってキャプチャされた最新のスナップショットをレンダリング対象として「取り込む」ために必要な呼び出しです。ムービーが利用できず描画が何もない場合、この関数は false を返します。

2.5 ムービーのシャットダウン

マルチスレッド型レンダリングの使用にあたって、レンダラー オブジェクトが破壊される前にレンダー スレッドが実行された場合、GFx::Movie Release 動作をレンダー スレッドによってサービスしなければならない可能性が生じます。このサービスは、ムービー内部に存在するデータ構造を参照するすべてのレンダー スレッドを解放するために必要です。デフォルトでは、ムービー シャットダウン コマンドは、前述の Render::ThreadCommandQueue インタフェースを介して、ムービー デストラクタからキューイングされます。適切なシャットダウンが行われるように、開発者は ThreadCommandQueue インタフェースを実装しなければなりません。

レンダー スレッド上でコマンド キューをサービスするには、アドバンス スレッドがムービー シャットダウン ポーリング インタフェースを使って最初にシャットダウンを開始し、その完了を待ってからムービーを解放するという方法もあります。このような動作を行うには、最初に Movie::ShutdownRendering(false) を呼び出してシャットダウンを開始し、続いて Movie::IsShutdownRenderingComplete を使って各フレームでの完了をチェックします。開始されたシャットダウンはレンダー スレッド上で NextCapture が処理し false を返します。シャットダウン処理が完了するとアドバンス スレッドはブロックされることなくムービー解放を実行します。

3 マルチスレッド型レンダリングのコンセプト

Scaleform のマルチスレッド型レンダリングには少なくとも、**アドバンス スレッド**と**レンダリング スレッド**のふたつのスレッドが関与します。

アドバンス スレッドとは、一般に、ムービー インスタンスを生成し、入力処理を扱い、GFx::Movie::Advance を呼び出して ActionScript とタイムライン アニメーションを実行するスレッドです。アプリケーション内に複数のアドバンス スレッドが存在しても構いませんが、それぞれのムービーは、通常はひとつのアドバンス スレッドのみがアクセスします。多くの Scaleform サンプルで、アドバンス スレッドはメイン アプリケーション コントロール スレッドにもなっており、必要に応じてレンダー スレッドにコマンドを発行します。

レンダー スレッドの主な役割は、Render::HAL のようなハードウェア抽象化レイヤーを介してグラフィック デバイスに対する出力コマンドを処理し、グラフィクスを画面上にレンダリングすることです。Scaleform レンダー スレッドは、やりとりのオーバーヘッドを抑えるために、「トライアングルの描画」のようなマイクロ コマンドではなく、「ムービーの追加」や「フレームの描画」のようなマクロコマンドを処理します。また、レンダー スレッドは、メッシュ キャッシュとグリフ キャッシュを管理します。

以下に述べるようにレンダー スレッドは、コントローリング アドバンス スレッドに対して、ロックステップまたは独立して動作します。

- ロックステップ レンダリングでは、一般にムービー アドバンスの処理と処理の間に、コントローリング スレッドが「フレーム描画」コマンドをレンダー スレッドに発行します。レンダリング スレッドがひとつ前のフレームを描画しながら、アドバンス スレッドが次のフレームを処理し、ふたつのスレッドは並行に動作します。マルチコア システムでは性能の向上につながる一方で、それぞれのスレッド フレーム間の一対一の関係は維持されます。
- 独立動作モードでは、場合によっては異なるフレームレートで動作しながら、レンダー スレッドはアドバンス スレッドとは独立してフレームを描画します。ムービーへの変更は描画の完了後に時間を置かずに表示に反映され、更新済みムービー スナップショットへのレンダー スレッドによる次回の描画は回避されます。

Scaleform Player アプリケーションと Scaleform サンプルは、セットアップが簡単であるとともにゲームで一般的であるという理由から、双方ともロックステップ レンダリングを使用しています。

3.1 ムービーのスナップショット

マルチスレッドでのレンダリング中は、レンダー スレッドが表示を目的としてムービーにアクセスしている間、同時に Advance スレッドもムービーの内部状態を変更することができてしまいます。ふたつのスレッドによるデータ アクセスはいつ発生するか分からず、クラッシュあるいはフレーム破壊が起こることも想定されるため、レンダー スレッドが整合の取れたフレームを常にアクセスできるように、新しいレンダラーではスナップショットを利用しています。

Scaleform においてムービー スナップショットとは、所定の時点においてキャプチャしたムービーの状態を指します。デフォルトでは GFx::Movie::Advance 呼び出しの終了時にスナップショットが自動的にキャプチャされます。あるいは GFx::Movie::Capture を呼び出して明示的にキャプチャすることも可能です。スナップショットがキャプチャされると、フレームはレンダー スレッドから利用可能な状態となる一方で、ムービーの動的状態とは別に保存されます。フレームを描画するとき、レンダー スレッドはムービー ハンドル上で NextCapture を呼び出し、もっとも新しいスナップショットを取り込んでレンダリングに使用します。

このような方法に対していくつかの補足が必要です。

- Movie::Advance と入力処理ロジックは、レンダリングと並行して実行するにあたって、クリティカル セクションとはなりません。
- Capture と NextCapture 呼び出しは異なる頻度で呼び出しが可能です。Capture を複数回にわたって連続的に呼び出すとムービー スナップショットはマージされます。十分な Capture 呼び出しがない場合、同じフレームが複数回にわたってレンダリングされる可能性があります。
- 入力、Invoke、あるいは Direct Access API に起因してムービーに与えられた変更は、Capture メソッドまたは Advance メソッドが呼び出されるまで、レンダー スレッドからは見えません。

最後の項目は Scaleform 4.0 の挙動と以前のバージョンとの大きな違いのひとつです。Scaleform 3.3 では Movie::Invoke を呼び出したあと Display を呼び出して変更を画面上に表示することができましたが、これからは変更を表示するには Capture か Advance のいずれかを呼び出さなければなりません。一般に入力処理のときは不要であり、また、複数の Invoke 呼び出しは Advance への呼び出しの前にグループ化することができます。

3.2 レンダリング状態の保存

Render::Renderer2D::Display は、レンダーをするデバイス上でムービーのフレームをレンダーするためのデバイス呼び出しを行います。パフォーマンスを損なわないために、ブレンドモードやテクス

チャ格納設定などの様々なデバイスステートは保存されず、デバイスのステートは Render::Renderer2D::Display を呼び出した後では異なります。 アプリケーションによっては、これによって不都合が生じます。最も直接的な解決方法は、Display への呼び出し前にデバイス ステートを保存しておいて、その後復元することです。Scaleform レンダリング後、同じゲーム エンジンに、必要なステートを再初期化させれば、パフォーマンスをさらに向上できます。

4 レンダラーのアーキテクチャ

マルチスレッド型レンダリングの正しい理解を図るために Scaleform 4.0 レンダー アーキテクチャの構成図を説明します。

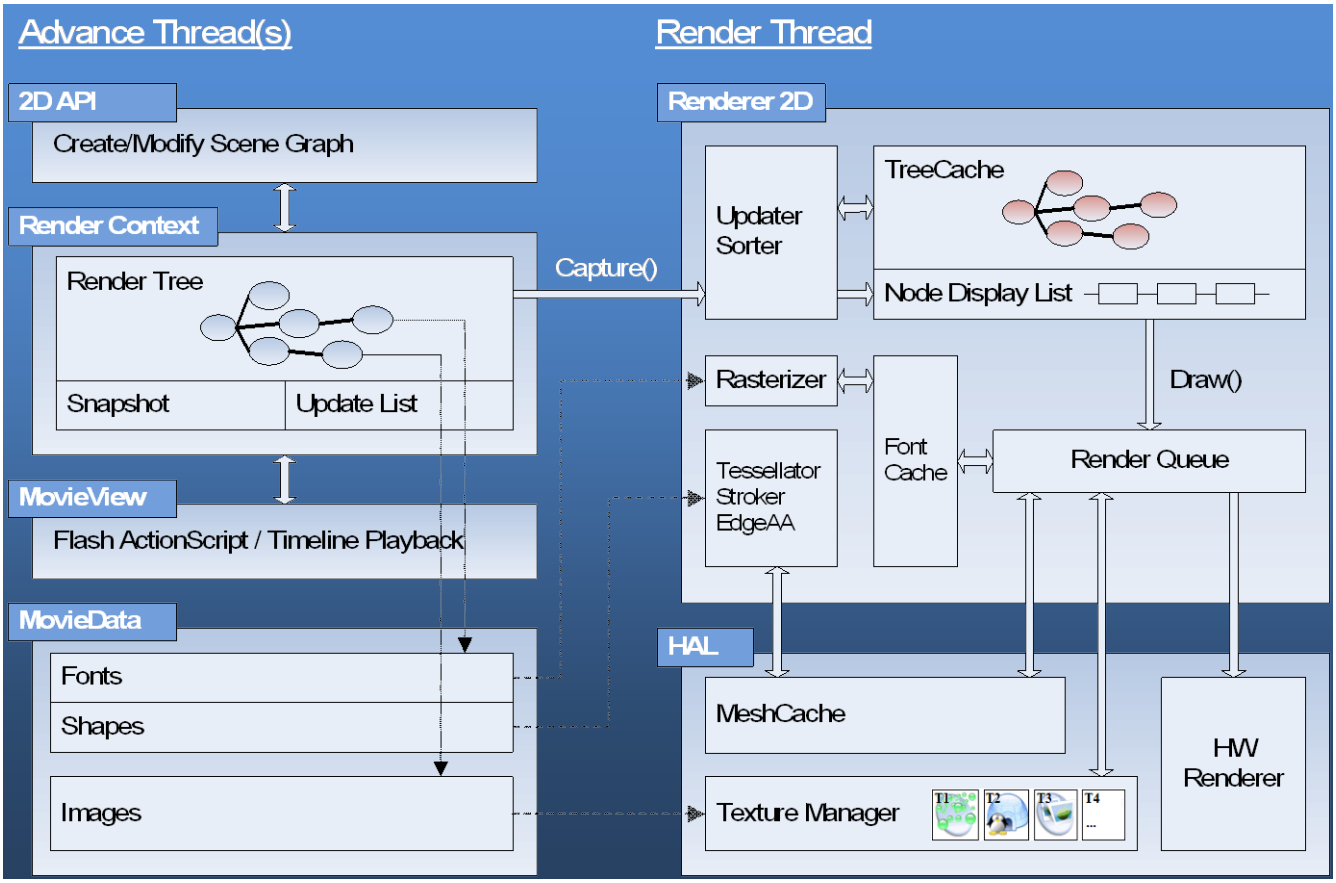

図 1. レンダーのアーキテクチャ

この図はふたつのスレッドを実行しているときのレンダラー サブシステムの関係を表しています。左側にある Movie と MovieDef オブジェクトは、それぞれ Scaleform ムービー インスタンスとその内部共有データです。Render Context はレンダリング サブシステムのフロントエンドであり、Render Tree を管理しています。Render Context は、2D シーン グラフのスナップショットと、レンダーツリーからアクセス可能な更新リストに対する責任を担っています。Scaleform では Render Context は Movie に包含されているため、エンドユーザーがアクセスする必要はありません。

右側のレンダー スレッドは Render::Renderer2D と Render::HAL のふたつのオブジェクトを管理しています。Renderer2D はレンダー エンジンの中核となるもので、ムービー スナップショットを画面

に出力する Display メソッドを包含しています。Render::HAL の HAL とは「ハードウェア抽象化レイヤー」の略で、各プラットフォームごとに実装が異なる抽象化クラスです。Render HAL は、グラフィック コマンドの実行のほか、バーテックスやテクスチャ バッファなどのリソース管理に責任を担っています。

図からもわかるように、レンダリング システムの大半はレンダー スレッドによって管理および実行されています。とくにレンダー スレッドは次のすべての責任を担います。

- キャッシュされたソート済みレンダー ツリーを管理し、新しいスナップショットから変更が届いたときに更新します。
- Font Cache を管理し、新しいグリフをレンダーする必要があるときに Font Cache を更新します。
- テッセレートされたベクター データのバーテックスおよびインデックス バッファで構成される Mesh Cache を管理します。
- Render Queue を管理してバッファ更新/ロック管理を効率化するとともに、レンダリング中の CPU の待ちを最小限に抑えます。
- HAL API を介してグラフィック コマンドをデバイスに発行します。

キャッシュ管理ロジックをレンダー スレッド上に維持することで、メイン スレッドは ActionScript やゲーム AI など処理量の多いタスクから解放されるため、多くの場合において性能向上に効果があります。また、キャッシュをレンダリング スレッド上に維持することで、スレッド間でのバーテックスやグリフ データのコピーに必要となる追加バッファを省略できるため、同期オーバーヘッドやメモリ使用量の低減が図れます。

5 サブシステム記述子のレンダー

5.1 TextureCache

TextureCache システムは、texture リソースのアンロードを取り扱います。これは、モバイルプラットフォームなどのメモリーの制限されたシステムでの使用を目的としており、メモリー総使用量を削減しますが、どのプラットフォームでも使用可能です。しかし、デフォルトでは iOS、Android、PS Vita プラットフォームのみで初期化され、これは HAL::InitHAL 内部での TextureManager オブジェクトの黙示的な生成中に行われます。TextureCache は、画像データがそのテクスチャデータよりも小さな画像にのみ適用され、このテクスチャデータとは、GPU の使用のためには非圧縮 RGBA カラーデータにデコードされる必要のある圧縮画像を意味します。これには PNG、JPEG、Flash 内部の画像フォーマットがあります。これは圧縮された画像には当てはまらず、GPU ディレクトリや、圧縮されていない画像で使用できます。これらの画像に対するこれらのデータは、そのテクスチャデータが画像からコピーされると自動的にアンロードされます。

TextureManager が割り当てられていて、HALInitParams に受け渡されている場合、そのコンストラクターの最後のパラメーターとして TextureCache が使用されます。TextureCacheGeneric と言う名前で、TextureCache のただひとつの実装が提供されますが、ユーザーはここから派生する TextureCache 実装を生成できます。この場合には、TextureManager は割り当てられていて、派生した TextureCache インスタンスが受け渡されている必要があり、その後 TextureManager は HALInitParams 経由で InitHAL に受け渡される必要があります。

TextureCache のジェネリックな実装では、簡単な LRU ストラテジーを用いてメモリーからテクスチャを削除します。該当するテクスチャのしきい値が割り当てられるまでは、テクスチャを削除しようとすることはありません。しきい値に到達すれば（デフォルトでは 8MB）、再び制限内に収まるまで LRU テクスチャを削除しようとします。制限内に収まらなければ、単一のフレーム内でしきい値を超えるテクスチャメモリーが使用されているので、警告が発せられます。制限値はランタイムで調整でき、これには TextureCache の SetLimitSize 関数を使用します。システムは、適切なレンダリングに必要なテクスチャを削除することは決してありません。

6 カスタムデータのレンダリング

まれに、ユーザーはある形状を Scaleform の標準リリースではできない別の形にレンダリングしたい場合があります。これが、頂点変換の変化や、特殊フラグメントシェーダー塗りつぶし効果を達成する方法になる場合もあります。

6.1 形状レンダリングの阻止

通常、動画では形状のほんの一部分にしか特殊なレンダリング効果を施しません。これらのオブジェクトは、ActionScript で識別され、AS2 の拡張プロパティ「rendererString」や「rendererFloat」、または AS3 拡張メソッドの「setRendererString」や「setRendererFloat」が使われます。それぞれの詳しい使い方は「AS2/AS3 拡張リファレンス」を参照してください。ムービークリップに string や float を設定すると、ムービークリップがレンダリングされる前に UserDataState::Data のインスタンスが HAL::PushUserData 機能を経由して Render::HAL's UserDataStack メンバにプッシュされます。いつでもスタック全体にアクセス可能であるため、ネストしたデータを使うことができます。ただし、各ムービークリップには 1 つの String または Float しか含めることができません。ムービークリップのレンダリングが終了すると、HAL::PopUserData 機能のスタックから UserDataState::Data が取り出されます。

6.2 カスタムレンダリングの例

現在、カスタムレンダリングの実行はムービークリップのユーザーデータセットを参照するため、HAL を変更する必要があります。HAL を変更してカスタムレンダリングを行う方法はいくつかありますが、ここでは、2D ポジションオフセットをユーザーデータに対応してレンダリングする全ての形状に追加することで、内部シェーダーシステムを拡張します。
この例では、インスタンス名に「m」の付いたムービークリップ AS3 SWF を作成してから、以下の AS3 コードを追加します。

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

新しいシェーダースクリプトシステ 4.4 を使用すると、簡単に新しいシェーダー機能が追加できます。Src/Render/ShaderData.xml に以下の色付きラインを追加します。

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

このラインによりシェーダーシステムが「Position2dOffset」スニペットを使用してバーテックスシェーダの位置データを処理します。その後、ファイルに「Position2dOffset」スニペットを追加します。

```
<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>
```

このスニペットは、バーテックスに変化した後の x と y に均一値「poffset」を追加する点を除いて、「Position2d」スニペットと同一です。

ここで、D3D9_Shader.cpp で以下のブロックを ShaderInterface::SetStaticShader:の上に追加します。

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

このブロックは、HAL の UserDataStack の上部に阻止しようとしているデータが入っていること（「abc」というストリング、フロートに値が入っていること）を確認します。一致する場合、受信シェーダタイプを変更して、Position2dOffset パスに追加します。これらの記号は、ShaderData.xml がカスタムビルドステップを実行するまで確認できません。この処理により、』新しく D3D9_ShaderDescs.h が生成されます。

使用するシェーダー定義を変更するほかに、実際にレンダリングする前に「poffset」ユニフォームを更新する必要があります。これは、以下のブロックを ShaderInterface::Finish:の上部に挿入して行います。

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

この例では、「meshCount」ごとにユニフォームを設定しています。meshCount は、バッチまたはインスタンスドロー（シェーダー変更サポート）より大きくなります。HAL の変更バージョンを実行すると、ムービークリップ「m」が少し右に移動します。

6.3 カスタムドローのバッチを無効にする

HAL は Scaleform シェーダーシステム以外で実行されたレンダリングを変更することもできます。この変更を行う具体的な方法は、期待する結果などで大きく異なるためここでは説明を省略します。
外部でレンダリングを実行する場合は、バッチおよびインスタンスメッシュの生成を無効にしておくことをお勧めします。これは、Scaleform 以外でオーサリングされたシェーダーは通常バッチやインスタンス化を Scaleform 内部のシェーダーと同じ方法でサポートしていないからです。バッチドローを無効にする場合は、AS2 “disableBatching”エクステンションメンバーを使うか、AS3 “disableBatching”エクステンション機能を使います。これらエクステンションの具体的なシンタックスは、AS2/AS3 エクステンションリファレンスガイドを参照してください。

7 GFxShaderMaker

7.1 概要

Scaleform 4.1 で導入した GFxShaderMaker は、Scaleform の使用するシェーダーを構築、コンパイルするユーティリティです。レンダラープロジェクト（例えば Render_D3D9）をビルドし直す場合、これは ShaderData.xml のカスタムビルド ステップとして実行されます。これはレンダラーがレンダラーをコンパイル、リンクするために使用するいくつかのファイルを生成します。これらはプラットフォームによって異なります。例えば、D3D9 では、次のファイルが生成されます。

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

他のプラットフォームでは、ツールチェーンを使用して、実行可能ファイルにリンクされているシェーダーを含むライブラリを直接生成できます。

7.2 複数シェーダー*API* をサポートする *HAL*

7.2.1D3D9

D3D9 HAL は ShaderModel 2.0 と ShaderModel 3.0 の両方をサポートします。デフォルトでは、これら全ての機能のレベルは HAL に含まれています。これらどちらかに対するサポートは明示的に削除可能です。こうするとシェーダー記述とバイナリ シェーダーの生成のための余分なサイズを節約できます。これには、GFxShaderMaker の'-shadermodel'オプションを使用して、シェーダーモデルのリストをコンマで区切って並べます。たとえば、

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

こうすると ShaderModel 2.0 へのサポートは削除されます。ShaderModel 2.0 しかサポートしないデバイスで InitHAL を呼び出そうとしても、うまく行きません。

7.2.2D3D1x

D3D1x HAL はデバイスを生成できるシェーダーの異なるサポートレベルをに対応した 3 つの機能レベルをサポートします。これらは次の通りです（D3D11 と D3D10.1 に必要なプリフィックスを付けてください）。
FEATURE_LEVEL_10_0
FEATURE_LEVEL_9_3
FEATURE_LEVEL_9_1

デフォルトでは、これら全ての機能のレベルは HAL に含まれています。どれに対するサポートも明示的に削除可能です。こうするとシェーダー記述とバイナリ シェーダーの生成のための余分なサイズを節約できます。これには、GFxShaderMaker の'-featurelevel'オプションを使用して、必要とするレベルのリストをコンマで区切って並べます。たとえば、

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

こうすると FEATURE_LEVEL9_3 へのサポートは削除されます。FEATURE_LEVEL_9_3 を使用するデバイスで InitHAL を呼び出そうとすると、次に低いサポートレベル（この場合は FEATURE_LEVEL_9_1）が使用されます。