Autodesk®
Scaleform®

Memory System Overview

本書では Scaleform 3.0 以降のバージョンで、メモリを構成、最適化、さらに管理する方法について説明しています。

著者: Michael Antonov、Maxim Shemanarev

バージョン: 4.01

最終更新日: 2011 年 2 月 17 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

© 2012 Autodesk, Inc. All rights reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

The following are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries: 123D, 3ds Max, Algor, Alias, AliasStudio, ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Homestyler, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, Beast (design/logo) Built with ObjectARX (design/logo), Burn, Buzzsaw, CAiCE, CFdesign, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Creative Bridge, Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWFX, DXF, Ecotect, Evolver, Exposure, Extending the Design Team, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, Freewheel, GDX Driver, Green Building Studio, Heads-up Design, Heidi, Homestyler, HumanIK, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Instructables, Instructables (stylized robot design/logo),Inventor, Inventor LT, Kynapse, Kynogon, LandXplorer, Lustre, MatchMover, Maya, Mechanical Desktop, MIMI, Moldflow, Moldflow Plastics Advisers, Moldflow Plastics Insight, Moondust, MotionBuilder, Movimento, MPA, MPA (design/logo), MPI (design/logo), MPX, MPX (design/logo), Mudbox, Multi-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, RiverCAD, Robot, Scaleform, Scaleform GFx, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Sparks, SteeringWheels, Stitcher, Stone, StormNET, Tinkerbox, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, U-Vis, ViewCube, Visual, Visual LISP, Vtour, WaterNetworks, Wire, Wiretap, WiretapCentral, XSI.

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer
THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. “AS IS.” AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF

MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Autodesk Scaleform の連絡先:

ドキュメント	Memory System Overview (のメモリ システムの概要)
住所	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
ホームページ	www.scaleform.com
電子メール	info@scaleform.com
電話	(301) 446-3200
Fax	(301) 446-3199

目次

1 Scaleform 3.x メモリシステム

Autodesk® Scaleform® 3.0 から、メモリのアロケーションがヒープ方式へと変更されるとともに、アロケーションおよびヒープに関する詳細なメモリレポート機能が導入されました。メモリレポート機能はプログラムから利用できるほか、Scaleform 3.2 以降でスタンドアロンアプリケーションとして配布されている AMP（Analyzer for Memory and Performance）ツールから利用することができます。また、最新の Scaleform 3.3 リリースから、効率的ではなかった以前のヒープ インプリメンテーションに対処するために、メモリインタフェースが更新されています。この変更についてはセクション 1.2 で説明します。

このドキュメントの残り部分で、以下のポイントを含む Scaleform 3.3 のメモリシステムを説明します。

- オーバーライド（上書き）可能なメモリアロケーションインタフェース
- Movie と Scaleform サブシステムのヒープベースのメモリマネージメント
- ActionScript のガーベージコレクション
- 利用できるメモリレポート機能

1.1 メモリのコンセプト

このセクションでは、Scaleform メモリマネージメントの主なコンセプト、相互のやりとり、および効率的なメモリの利用方法についてその概要を説明します。アロケータをカスタマイズする前に、あるいは Scaleform メモリ利用状況をプロファイリングする前に、これらコンセプトを理解しておいてください。

1.1.1オーバーライド可能なアロケータ

Scaleform のすべての外部メモリアロケーションはスタートアップ時に GFx::System にインストールされるセントラルインタフェースを介して実行される一方で、独自のメモリマネージメントシステムをプラグインできるようになっています。メモリマネージメントをカスタマイズするには以下のいずれかの方法を使用します。

1. メモリアロケータを委任するには Alloc/Free/Realloc 関数を実装した独自の Scaleform::SysAlloc インタフェースをインストールします。アプリケーションが dlmalloc のようなセントラルアロケータ インプリメンテーションを使う場合に最適な方法です。
2. ひとつ以上の固定メモリブロックを、あらかじめアロケートした状態で、スタートアップ時に Scaleform 与えます。追加ブロックは「Pause Menu」のようなテンポラリ画面用に予約される場合があります。メモリブロックはメモリアリーナの使用を通じて完全に解放されます。

3. ハードウェアページングのメリットを享受するには、Scaleform::SysAllocWinAPI のような Scaleform 提供のシステムアロケータ インプリメンテーションを使用します。

アプリケーションが採用するメモリ戦略に応じていずれかの方法を選択します。方法(2)はサブシステムに固定的なメモリバジェットを割り当てるコンソールで一般的です。それぞれのインプリメンテーションについてはセクション 2 の「メモリアロケーション」で説明します。

1.1.2メモリヒープ

いずれのメモリマネージメント方法を選択したかに関わらず、すべての内部 Scaleform アロケーションは、ファイルと目的に応じて各ヒープに体系付けられます。AMP 出力または MemReport 出力からその様子がわかります。一般的なヒープは次のとおりです。

- Global ヒープ – すべての共有アロケーションとコンフィギュレーションオブジェクトを保持しています。
- MovieData ヒープ – 特定の SWF/GFx ファイルからロードされた読み取り専用データを格納しています。
- MovieView ヒープ – タイムラインとインスタンスデータを含む単一の GFx::Movie ActionScript サンドボックスを表します。このヒープは内部ガーベージコレクションの対象になります。
- MeshCache – テッセレートされたシェープトライアングルデータはここにアロケートされます。

メモリヒープ インプリメンテーションとは、所定のファイルサブシステムのアロケーションを「ページ」と呼ぶ専用メモリブロックからサービスすることです。これには複数のメリットがあります。

- 複数の外部システムアロケーションが不要となるため、性能が向上します。
- 単一のスレッドしかヒープをアクセスしないため、ヒープ内でのスレッド同期が不要です
- 関連データを構成単位としてフリーにすることで、外部フラグメンテーションを削減します
- メモリレポートを論理構造で出力します

ただしヒープは、内部フラグメンテーションの対象になるというデメリットももたらします。内部フラグメンテーションは、ヒープ内でフリーとなっている小さいメモリブロックが残りのアプリケーションに対して解放されていないときに起こります。その理由は、それらの関連ページすべてがフリーではないためで、結果的に「未使用」メモリがヒープに保持されることになるからです。

Scaleform 3.3 では、メモリをより積極的にフリーにする Scaleform::SysAlloc ベースの新しいヒープ インプリメンテーションを提供することで、この課題に対処しました。また Scaleform 3.3 では、単一のメモリヒープを共有することを目的に、同一スレッドに複数の GFx::Movie を配置できるようになっています。結果として、部分的に使用されたブロックが保持するメモリ量が少なくなります。

1.1.3ガーベージコレクション

AS データ構造内での循環参照によって発生するメモリリークを排除するために、ActionScript（AS）のガーベージコレクションを Scaleform 3.0 から導入しました。CLIK および関連 ActionScript プログラムの動作には。適切なコレクションが不可欠です。

Scaleform においてガーベージコレクションは、AS Virtual Machine の実行サンドボックスとして機能する GFx::Movie オブジェクトの一部として組み込まれています。Scaleform 3.3 では、関連ガーベージコレクタを共有および単一化する Movie ヒープを共有することが可能です。多くの場合、ガーベージコレクションは、ActionScript のアロケーションによって Movie ヒープが成長したときに自動的にトリガーされます。ただし、明示的にトリガーしたり、フレームを基準とする間隔で明示的にコレクションを Scaleform に指示することも可能です。コレクションの詳細とその設定オプションについては、セクション 4 の「ガーベージコレクション」で説明しています。

1.1.4メモリレポートとデバッグ

Scaleform メモリシステムは、アロケーションの目的を示す「stat ID」タグを使って、アロケーションにマークできるようになっています。アロケートされたメモリの情報は、stat ID カテゴリで分類したヒープ別に、または、ヒープで分類した stat ID 別に、レポートとして出力されます。メモリレポート機能は、Scaleform 3.2 以降でスタンドアロンのプロファイリングアプリケーションとして配布される「Analyzer for Memory and Performance」（AMP）で提供されています。AMP の使用方法については「AMP User Guide」を参照してください。なおメモリレポートは、Scaleform::MemoryHeap::MemReport 関数をコールすることで、プログラム的にストリング形式で取得することもできます。

上述の stat ID タグは、デバッグデータとして、実際のメモリアロケーションとは別に格納されます。また、デバッグデータはメモリリークの検出を目的として保持されます。Scaleform のデバッグバージョンがシャットダウンしたとき、Visual Studio 出力ウィンドウまたはコンソール内にリークしたメモリレポートが生成されます。Scaleform では現在のところ内部メモリリークの発生は認められていないため、検出されたリークはすべて Scaleform の参照カウントの不適切な使用によるものと考えられます。追加の関連デバッグデータは Shipping コンフィギュレーション内にはアロケートされません。

1.2 Scaleform 3.3 メモリ API の変更

Scaleform 3.3 では、前述のとおり、メモリ利用を改善する目的でふたつの大きなアップデートが適用されています。

- GFx::Movie メモリコンテクストシェアリングをアップデートすることによって、個々のムービービューインスタンスがヒープとガーベージコレクションを共有できるようになり、メモリの再利用性が改善されました。内部ストリングテーブルも共有するとさらなる節約が得られます。これら共有の詳細についてはセクション 4 で詳しく説明します。
- SysAlloc インタフェースをアップデートして「malloc との親和性」をより高めました。外部アロケーションでの 4K ページ境界要件を廃止するとともに、ヒープを最適化して 512 バイトよりも大きいすべてのアロケーションを SysAlloc に直接ルートするようにしています。こによってアプリケーションにメモリが返される頻度が高くなり、利用効率と再利用性を高めています。

なお、もともとの Scaleform::SysAlloc インプリメンテーションは現在では Scaleform::SysAllocPaged に名称が変更されているだけで引き続き利用することができます。開発者は、開発者自身のインプリメンテーションで SysAlloc をオーバーライドするか、SysAllocPaged に対するベースクラスを変更することができます。SysAllocStatic とメモリアリーナに依存しているユーザーは、そのクラスは依然として利用可能ですので影響を受けません。

2 メモリアロケーション

セクション 1.1.1 で述べたように、Scaleform では、以下の 3 通りの方法で開発者によるメモリアロケーションのカスタマイズが可能です。

1. SysAlloc インタフェースをオーバーライドして、アプリケーションメモリシステムへのアロケーションを委任する。または、
2. Scaleform にひとつ以上の固定サイズメモリブロックを与える。または、
3. メモリをオペレーティングシステムから直接アロケートするように Scaleform に指示する。

メモリの利用効率を高めるには開発したアプリケーションに適したアプローチを選択する必要があります。アプローチ(1)と比較的使われることの多いアプローチ(2)の違いは以下で説明します。

開発者が SysAlloc をオーバーライドした場合、Scaleform アロケーションは開発したメモリシステムに委任されます。Scaleform はたとえば新しい SWF ファイルのロードのためにメモリを必要としたとき、Scaleform::SysAlloc::Alloc 関数をコールしてメモリを要求します。コンテンツのアンロードで Scaleform は Scaleform::SysAlloc::Free をコールします。このシナリオの場合、Scaleform を含むすべてのシステムが単一の共有グローバルアロケータを使用したときにメモリの利用効率がもっとも高くなります。その理由は、フリーとなっているメモリブロックはいずれも、フリーメモリを必要とするかもしれないすべてのシステムに対して再利用可能な状態になっているためです。アロケーションのいかなるコンパートメント化も共有効率全体を低下させ、結果として総メモリフットプリントは増加します。SysAlloc のオーバーライドの詳細はセクション 2.1 で説明します。

グローバルアロケーションアプローチを使った最大の課題はフラグメンテーションの発生で、多くのコンソール開発者が固定メモリレイアウトを好む理由にもなっています。固定メモリレイアウト方法では、スタートアップ時あるいはレベルロード中に決まるサイズで、メモリ領域が各システムにアロケートされます。このアプローチは、グローバルシステムが持つメモリ利用効率と、予測可能なメモリレイアウトのいずれかを選択するかというトレードオフの問題ともいえます。

アプリケーションが固定メモリアプローチを使用する場合、開発者は、セクション 2.2 で説明する Scaleform の固定サイズメモリブロックアロケータを同様に使用しなければなりません。このシナリオをより現実的なものにするために、Scaleform は、二次メモリアリーナの作成、または「Pause Menu」を表示するときのような限られた時間だけ使われるメモリ領域の作成を許しています。

2.1 アロケーションをオーバーライドする

外部 Scaleform アロケータを置き換えるには、次のふたつのステップに従ってください。

1. 独自の SysAlloc インターフェイス インプリメンテーションを作成します。これは Alloc、Free、Realloc メソッドを定義します。

2. Scaleform の初期化中に GFx::System コンストラクタにこのアロケータのインスタンスを提供します。

デフォルトの Scaleform::SysAllocMalloc は、標準の malloc/free に依存するともに、アライメントをサポートしたシステム固有の malloc/free の代替に依存しています。デフォルトの SysAllocMalloc は Scaleform によって提供され、直接使用されるかリファレンスとして使用されます。

2.1.1独自の SysAlloc を実装する

独自のメモリ ヒープを実装する最も簡単な方法は、弊社の GFx SysAllocMalloc インプリメンテーションをコピーして、自分のメモリ アロケータまたはヒープを呼び出すように変更することです。わずかな変更が入った Windows 特有の SysAllocMalloc の実装は下記の事項を含んでおります。

```
class MySysAlloc : public SysAlloc
{
  public:
    virtual void* Alloc(UPInt size, UPInt align)
    {
        return _aligned_malloc(size, align);
    }

    virtual void  Free(void* ptr, UPInt size, UPInt align)
    {
        SF_UNUSED2(size, align);
        _aligned_free(ptr);
        return true;
    }
    virtual void* Realloc(void* oldPtr, UPInt oldSize,
                          UPInt newSize, UPInt align)
    {
        SF_UNUSED(oldSize);
        return _aligned_realloc(oldPtr, newSize, align);
    }
};
```

ご覧のように、アロケータの実装は非常に簡単です。MySysAlloc クラスはベースの SysAlloc インターフェイスから作成され、3 つの仮想関数、Alloc、Free 、Realloc を実装しています。これら関数のインプリメンテーションではアライメントを遵守しなければなりませんが、Scaleform は一般に 16 バイト以下程度の小さいアライメントのみを要求します。oldSize 引数と align 引数はインプリメンテーションを単純化するために Free/Realloc インタフェースに渡されます。これらの引数はたとえば Realloc をインプリメンテーションするときの Alloc/Free コールのラッパとして有用です。

デベロッパーは自分のアロケータのインスタンスをいったん作成すると、Scaleform の初期化中に GFx::System コンストラクタに渡すことができます。

```
MySysAlloc myAlloc;
System gfxSystem(&myAlloc);
```

Scaleform 3.0 以降では GFx::System オブジェクトを他のすべての Scaleform オブジェクトより前に作成し、Scaleform オブジェクトがすべて解放された後で破棄する必要があります。これは通常、Scaleform を使用するコードを呼び出す割り当て初期化関数の一部として行うのが最善の方法ですが、有効期間が Scaleform よりも長い、別の割り当てられたオブジェクトの一部とすることもできます (GFx::System はグローバルに宣言するべきではありません)。このような使用法が不便だと思う場合、そのオブジェクトを作成せずに、GFx::System::Init() と GFx::System::Destoy() 静的関数を代わりに使用することもできます。GFx::System コンストラクタと同様に、GFx::System::Init() は SysAlloc ポインタ引数を取ります。

2.2 Scaleform の固定メモリブロックの使用

導入部で説明したように、ひとつ以上のメモリブロックを予約しておき、SysAlloc をオーバーライドする代わりにそれらメモリブロックを Scaleform に渡す方法があります。この処理を行うには SysAllocStatic を次のようにインスタンス化します。

```
void*            pmemChunk = …;
GSysAllocStatic  blockAlloc(pmemChunk, 6*1024*1024);
GFxSystem        gfxSystem(&blockAlloc);
…
```

上記の例は 6 メガバイトのメモリ チャンクを Scaleform に渡して、アロケーションに使用します。このメモリ ブロックは `gfxSystem` と `blockAlloc` オブジェクトの両方がスコープから離れるまで、再利用も解放も、もちろんできません。ただし、ポーズメニューを表示するといった一時的な目的で「解放可能」なメモリブロックを追加することができます。メモリアリーナの使用を通じてサポートされているそのようなブロックについては以下で説明します。

静的アロケータを使用する場合、開発者は Scaleform が使用するメモリ サイズに特に気を配る必要があります。指定されたメモリ サイズを超えてしまうような、ファイルのロードや、ムービー インスタンスの作成が行われたりしないことを確認します。一度 SysAllocStatic がエラーを起こすと、その Alloc インプリメンテーションから 0 が返され、Scaleform がエラーになる、またはクラッシュする原因となります。必要であれば、開発者は簡単な SysAllocPaged ラッパー オブジェクトを使って、この重要な条件を検出することもできます。

2.2.1メモリ アリーナ

Scaleform 3.1 は「メモリ アリーナ」のサポートを導入しました。これはユーザー指定のアロケータ領域で、プログラムの実行時に、指定された時点で完全に解放されることが保証されています。具体的に言うと、メモリ アリーナは Scaleform ファイルがロードされ、GFx::Movies が作成されるメモリ領域を定義します。このような領域は、その領域を占有している Scaleform オブジェクトが破棄されると、完全に解放されます。メモリ アリーナは、残りの Scaleform をシャットダウンする、または無関係の Scaleform ファイルをアンロードせずに、破棄することができます。アリーナが破棄されると、アプリケーションは他の Scaleform 以外のデータに、そのメモリをすべて再利用することができます。使用例として、メモリ アリーナはゲームの「ポーズ メニュー」画面のロードに定義できます。この画面は一時的にメモリを必要としますが、ゲームが再開したら解放されて、再利用される必要があります。

2.2.1.1 背景

一般的なケースとして、ほとんどの開発者は、Scaleform に占有されているメモリを再利用するように、メモリ アリーナを定義する必要はありません。Scaleform がメモリを割り当てるとき、メモリは GFx::System で指定された SysAlloc オブジェクトから取得されます。このメモリは、データをアンロードして Scaleform オブジェクトを破棄すると、解放されます。ただし、多くの理由で、解放されたメモリ パターンが常に、アロケーションと合致するとは限りません。例えば、CreateMovie を呼び出して使用し、その後解放した場合、呼び出しの間に割り当てられたほとんどのメモリ ブロックは解放されますが、一部のブロックはもっと長い間保有される可能性もあります。これは、断片化、ダイナミック データ構造、マルチスレッド、Scaleform リソースの共有とキャッシングなどの多くの理由で起きることがあります。

ほとんどの場合、一時的に解放されないままのメモリ ブロックがわずかにあっても、問題にはなりません。このようなブロックは蓄積されず、Scaleform の有効期間中に再利用されるからです。しかし、アロケーションの予測できない特性のため、使用しているゲーム エンジンが、Scaleform と他のゲーム エンジン データの両方で共有する固定サイズのメモリ バッファを予約している場合は、問題が起きることもあります。以前のバージョンの SDK では、バッファ メモリを完全に解放するため、開発者は Scaleform をすべてシャットダウンしなければなりませんでした。ただ、Scaleform 3.1 の場合、ユーザーはこのようなバッファにメモリ アリーナを定義して、SWF/GFX ファイルを特定のメモリ アリーナにロードするように指定することができます。このようなファイルがアンロードされると、Scaleform::Memory::DestroyArena を呼び出して、すべてのアリーナ メモリを安全に再利用することができます。

2.2.1.2 メモリ アリーナを使用する

メモリ アリーナを使用するには、以下の手順に従う必要があります:

1. Scaleform::Memory::CreateArena を呼び出し、ゼロ以外の整数の識別子を付けてアリーナを作成します (ゼロはグローバル、またはデフォルトのアリーナを意味していて、そのようなアリーナは常に存在しています)。SysAlloc インターフェイスを提供しメモリにアクセスする必要があります。固定サイズのメモリ バッファの場合、SysAllocStatic を使用することができます。

```
// pbuf1が10,000,000バイトのバッファを指すと仮定する。
SysAllocStatic  sysAlloc(pbuf1, 10000000);
Memory::CreateArena(1, &sysAlloc);
```

2. GFx::Loader::CreateMovie / GFx::MovieDef::CreateInstance を呼び出します。その一方で使用するアリーナ ID を指定します。そのようなムービー オブジェクトに必要なヒープは、指定したアリーナ内に作成されるので、必要なメモリのほとんどは、そこから取得されます。一部の「共有」グローバル メモリは、そのまま割り当てられる可能性があるので、グローバルに使用できるメモリを十分に確保しておくことも必要です。

```
// ムービーにアリーナ 1を使用する。
Ptr<MovieDef> pMovieDef =
        *Loader.CreateMovie(filenameStr, Loader::LoadWaitFrame1, 1);
...
// インスタンスにアリーナ 1を使用する。
Ptr<Movie> pMovie =
        *pnewMovieDef->CreateInstance(MovieDef::MemoryParams(1), false);
```

3. 結果として作成された GFx::MovieDef/GFx::Movie オブジェクトを、必要な限り使用します。

4. アリーナ内に存在する、作成したムービーとオブジェクトをすべて解放します。GFx::ThreadedTaskManager が、スレッドのバックグラウンド読み込みに使用されていた場合、解放する前に GFx::MovieDef::WaitForLoadFinish を呼び出す必要があります。GFx::ThreadedTaskManager は、バックグラウンドのスレッドに読み込みを終了し、自分のムービー リファレンスを解放するように強制するためです。

```
pMovieDef->WaitForLoadFinish(true);
pMovie    = 0;
pMovieDef = 0;
```

   または (Scaleform::Ptr が使用されていない場合):

```
Movie->Release();
pMovieDef->Release();
```

5. DestroyArena を呼び出します。その後、すべてのアリーナ メモリは、CreateArena が再度呼び出されるまで、安全に再利用できます。Scaleform::Memory::ArenaIsEmpty 関数を使って、メモリ アリーナが空か、また、破棄してもよいかどうかを確認することができます。

```
// DestroyArenaは、メモリが呼び出しの前に適切に解放されなかった場合、アサートする。
```

```
// Releaseでは "return *(int*)0;" でクラッシュする。
Memory::DestroyArena(1);
```

SysAlloc オブジェクトは、これでスコープから離れる、または破棄することができます。それ以降は、‘pbuff1’ を再利用することができます。メモリ アリーナは必要なときに、もう一度、再作成することができます。

2.3 OS へのアロケーションの委任

アロケータをオーバーライドする代わりに、Scaleform が提供する OS ダイレクトアロケーションのひとつを使用することができます。次のようなアロケーションが用意されています。

- *Scaleform::SysAllocWinAPI* – VirtualAlloc/VirtualFree API を使用し、優れたアラインメントと、そのページング活用能力のために、Microsoft プラットフォームで最善の選択肢になります。
- *Scaleform::SysAllocPS3* – sys_memory_allocate/sys_memory_free 呼び出しを PS3 上の効果的なページ管理と配置のために使用します。

システムアロケーターを使う主な目的は、HW ページングから来ています。ページングとは、物理メモリをページ サイズのブロックで仮想アドレス空間に再マップする OS の機能です。アロケーション システムに賢明なかたちで使用された場合、ページングはそのアドレス空間中に散在している小さな空きメモリ ブロックから、直線的なメモリ範囲を使用できるようにすることで、大きなメモリ ブロック上の断片化に対抗することができます。

弊社の Scaleform WinAPI と PS3 GSysAlloc インプリメンテーションは、粒度のブロックでシステムメモリのマッピング/マッピング解除を行うことによって、この機能を活用しています。 この粒度は、たとえば dlmalloc で使われている 2MB ブロックよりも効率面で優れています。

2.3.1 SysAllocPaged のインプリメンテーション

Scaleform 3.3 で SysAlloc インタフェースがアップデートされたことに伴い、もともとの SysAlloc インプリメンテーションは SysAllocPaged に名称が変更になりました。SysAllocPaged を使ったアロケータ インプリメンテーションは Scaleform で引き続き利用可能なほか、GSysAllocStatic および OS 固有のアロケータ インプリメンテーションで使用されます。SysAllocPaged はライブラリには含まれていますが、使用しない場合はリンクされません。次に示すように、互換性を目的として、SysAlloc の代わりに SysAllocPaged を使用することができます。

参考までに、わずかに変更を加えた SysAllocPagedMalloc アロケータ コードを以下に掲載しています:

```
class MySysAlloc : public SysAllocPaged
{
  public:
    virtual void  GetInfo(Info* i) const
    {
        i->MinAlign     = 1;
        i->MaxAlign     = 1;
        i->Granularity  = 128*1024;
        i->HasRealloc   = false;
    }

    virtual void* Alloc(UPInt size, UPInt align)
    {
        // Ignore 'align' since reported MaxAlign is 1.
        return malloc(size);
    }

    virtual bool  Free(void* ptr, UPInt size, UPInt align)
    {
        // free() doesn't need size or alignment of the memory block, but
        // you can use it in your implementation if it makes things easier.
        free(ptr);
        return true;
    }
};
```

ご覧のように、MySysAlloc クラスはベースの GSysAlloc インターフェイスから作成され、3 つの仮想関数、`GetInfo`、`Alloc` 、`Free` を実装しています。また、最適化として実装できる `ReallocInPlace` という関数もありますが、ここでは省略しています。

- `GetInfo()` – アロケータ アラインメントのサポート機能と粒度を、`Scaleform::SysAllocPaged::Info` ストラクチャ メンバを記述することで返します。
- `Alloc` – 指定されたサイズのメモリを割り当てます。この関数は、Alloc インプリメンテーションで受け入れられるサイズとアラインメントの引数を取ります。渡されるアラインメントの値は、`GetInfo` が返す `MaxAlign` を超えることはありません。今回は MaxAlign 値を 1 バイトに設定したので、2 つ目の引数は無視しても問題ありません。
- `Free` – `Alloc` で以前割り当てられた空きメモリ。この `Free` 関数は元の割り当てサイズを告げる追加サイズと配置の引数を受け取ります。これらの引数はこの `Free` ベースのインプリメンテーションには必要ないので、無視してもかまいません。
- `ReallocInPlace` – 別の場所に移動せずに、メモリ サイズの再割り当てを試みます。不可能な場合は `false` を返します。多くのユーザーはこの関数をオーバーライドする必要はありません。詳細は「Scaleform Reference Documentation」を参照してください。

Alloc 関数と Free 関数の動作はかなり標準的です。したがって特に興味深い関数は GetInfo だけになります。この関数は使用しているアロケータの性能を Scaleform に報告するので、自分の SysAllocPaged インプリメンテーションに課せられたアラインメント サポートの必要条件、さらに Scaleform のパフォーマンスとメモリ使用の効率性に影響する場合があります。
SysAllocPaged::Info ストラクチャには以下の 6 つの値が含まれています:

- MinAlign – アロケータが「すべての」アロケーションに適用する最低限のアラインメントです。
- MaxAlign – アロケータがサポートできる最大限のアラインメントです。この値が 0 の場合、Scaleform はリクエストされたどのようなアラインメントでもサポートされると想定します。報告された値が 1 であれば、アラインメントはサポートされないと見なされるので、使用している Alloc インプリメンテーションでアラインメント引数を無視してもかまいません。
- Granularity – Scaleform アロケーションの推奨粒度です。Scaleform は割り当てのリクエストに、最低でもこのサイズを使おうとします。malloc の場合と同様に、使用しているアロケータがアラインメントに適していない場合、最低でも 64K という値を使用することをお勧めします。
- HasRealloc – 使用しているインプリメンテーションが ReallocInPlace をサポートしているかどうかを指定するブール型フラッグです。今回の場合は false を返します。
- SysDirectThreshold – グローバルサイズの閾値が、空値でないとき、これを定義します。アロケーション サイズが SysDirectThreshold と同じ、あるいは大きいとき、システムにリダイレクトされます。この時、グラニュレータ レイヤーは無視されます。
- MaxHeapGranularity – この MaxHeapGranularity は、空値でないとき、ヒープの最大可能粒度を制限します。頻発するセグメントの取得や解放の操作負荷によるシステムメモリのフットプリントは、アロケータの動作を遅くしてしまいますが、多くの場合、MaxHeapGranularity によって、このフットプリントを削減することができます。MaxHeapGranularity は、最小 4096 で、4096 の倍数とします。

MaxAlign 引数の説明で分かるように、アロケータがアラインメントをサポートしていないときには、SysAlloc インターフェイスは容易に実装できますが、サポートしていれば、それをうまく活用することもできるのです。実際に、考慮する価値のある可能なシナリオは 3 つあります:

1. 使用するアロケータ インプリメンテーションはアラインメントをサポートしない、または効率的に処理しない。この場合、MaxAlign 値を 1 に設定し、必要なアラインメント作業を Scaleform に内部ですべて行ってもらいます。
2. アロケータはアラインメントを効率的に処理する。この場合 MaxAlign を 0 に設定するか、サポートできる最大配置サイズに設定し、アラインメントを適切に処理するように Alloc 関数を実装します。
3. アロケータはシステムに対するインターフェイスであり、常に強制されアラインメントされるページ サイズを伴う。これを効率的に処理するには、MinAlign と MaxAlign の両方をシステムのページ サイズに設定し、すべての割り当てサイズを OS に渡すだけです。

自分のアロケータのインスタンスをいったん作成すると、Scaleform の初期化中に GFx::System コンストラクタに渡すことができます。

```
MySysAlloc myAlloc;
```

```
GFx::System gfxSystem(&myAlloc);
```

2.4 メモリ ヒープ

前述のとおり Scaleform はヒープと呼ぶメモリプール内にメモリを維持します。それぞれのヒープは、メッシュキャッシュのような特定のファイルサブシステムデータからロードされたデータを保持するといった特定の目的に従って作成されます。このセクションでは Scaleform コア内でヒープメモリを管理する API について説明します。

2.4.1ヒープ API

エンド ユーザーが使いやすいように設定するため、ほとんどのメモリ ヒープ管理の詳細は、内部のコードの中に隠されています。開発者は 'new' 演算子を外部のすべての Scaleform オブジェクト上で直接使用することができます。どのヒープ アロケーションを行うか悩む必要はありません。

```
Ptr<FileOpener> opener = *new FileOpener();
loader.SetFileOpener(opener);
```

ご覧のように、演算子'new'はいっさいヒープを指定せずに使用されています。したがって、このメモリ アロケーションは、Scaleform グローバル ヒープの内部で行われます。 グローバルヒープは GFx::System の初期化中に作成され、シャットダウンされるまで有効な状態を維持します。グローバルヒープは常にメモリアリーナ 0 を使用します。

ただし、メモリ ヒープは内部的に使用され断片化とメモリー使用のシステム単位のトラッキングを制御します。たとえば、GFx::MeshCacheManager クラスは内部のヒープを作成して、テッセレートされた図形データをすべて保有し拘束します。`GFx::Loader::CreateMovie()` はヒープを作成して、特定の SWF/GFX ファイルからロードされたデータを保持します。`GFx::MovieDef::CreateInstance` はヒープを作成して、タイムラインと ActionScript のランタイムステートを保有します。これらはすべて裏で行われので、ほとんどのユーザーに影響しないはずです。少なくとも Scaleform コードの変更や拡張、またはデバッグを企てるまでは影響されません。これらはすべて裏で行われるので、ほとんどのユーザーに影響しないはずです。少なくとも Scaleform コードの変更や拡張、またはデバッグを企てるまでは影響されません。

この章の残りの部分ではメモリ ヒープ システムの詳細について説明し、関係する多くのクラスやマクロの特定の情報を提供しています。この資料は Scaleform の使用に不可欠というわけではないので、時間がない場合は省略してもかまいません。

以前説明したように、メモリ ヒープは Scaleform::MemoryHeap クラスで表され、専用の Scaleform サブシステム、またはデータ セットのために作成されたインスタンスを伴います。子メモリ ヒープはそれぞれグローバル ヒープ上で `Scaleform::MemoryHeap::CreateHeap` を使って作成され、`Scaleform::MemoryHeap::Release` を呼び出すことで破棄されます。ヒープが有効な間は、メモ

リは Scaleform::MemoryHeap::Alloc 関数を呼び出してそのヒープから割り当て、Scaleform::MemoryHeap::Free を呼び出して解放することができます。ヒープが破棄されると、その内部メモリはすべて解放されます。サイズの大きなヒープの場合、このメモリは通常、SysAlloc インターフェイスを通じて直ちにゲームに解放されます。

メモリが正しいヒープから割り当てられていることを確認するため、Scaleform のほとんどのアロケーションは Scaleform::MemoryHeap ポインタを取る特別なマクロを使用します。以下が含まれます:

- SF_HEAP_ALLOC(heap, size, id)
- SF_HEAP_MEMALIGN(heap, size, alignment, id)
- SF_HEAP_NEW(heap) ClassName
- SF_FREE(ptr)

マクロを使うことで、適切なラインとファイル名情報が、デバッグ ビルドでのアロケーションのために確実に渡されます。を使ってアロケーションの目的にタグを付けます。Scaleform AMP メモリ統計システムで、それをトラックしてグループ化します。

メモリ アロケーション マクロがあるので、新規の GFx::SpriteDef オブジェクトを以下の呼び出しでカスタム ヒープ 'pHeap' に作成することができます:

```
Ptr<SpriteDef> def = *SF_HEAP_NEW(pHeap) SpriteDef(pdataDef);
```

ここでは、GFx::SpriteDef は Scaleform で使用される内部のクラスです。Scaleform のほとんどのオブジェクトは参照カウントされるので、多くの場合 Ptr<>などのスマート ポインタに保有され、そのようなポインタが内部の Release 関数を呼び出すことでスコープから離れるときに解放されます。Scaleform::NewOverrideBase から派生したものなど、非参照カウント オブジェクトの場合、'delete' 演算子が直接使用されます。ヒープを指定してメモリを解放する必要はありません。この delete 演算子と SF_FREE(address) マクロはそのヒープを自動的に見つけ出すことができるからです。

2.4.2自動ヒープ

メモリ割り当てのマクロは必要な機能をすべて提供してヒープのサポートを有効にしますが、プログラム コード全体でヒープ ポインタを渡すことを要求するので、場合によっては使いづらいことがあります。このポインタ渡しは、Scaleform::String や Scaleform::Array<>などのメモリ アロケーションを行うことが必要なコンテナ オブジェクトの集合に適用される場合は、特に手間がかかることがあります。クラスを宣言し、ヒープにそのインスタンスを作成する以下のサンプル コードについて考えます:

```
class Sprite : public RefCountBase<Sprite>
{
    String                  Name;
    Array<Ptr<Sprite> > Children;
```

```
    Sprite(String& name, …) { … }
};
Ptr<Sprite> sprite = *SF_HEAP_NEW(pHeap) Sprite(name);
sprite->DoSomething();
```

GFx::Sprite オブジェクト自体は指定されたヒープに作成されますが、このオブジェクトは、ダイナミックな割り当てが必要な文字列オブジェクトと配列オブジェクトを含んでいます。特に配慮しない限り、これらのオブジェクトはグローバル ヒープに割り当てられることになり、これでは、意図していたこととは異なります。この問題を解決するためには、ヒープ ポインタが、GFx::Sprite オブジェクト コンストラクタに渡され、次に文字列と配列コンストラクタへと、理論的には渡されればよいことになります。この引数渡しはプログラミングを著しく難しいなものにしてしまうばかりか、ポインタをヒープに維持するために、配列や文字列などの簡単なオブジェクトが多数必要になる可能性があり、どこか他の場所で使ったほうが良かったはずのメモリを消費してしまいます。

このような状況でプログラミングを簡素化するため、Scaleform コアは特別な「自動ヒープ」割り当てマクロを使用します:

- SF_HEAP_AUTO_ALLOC(ptr, size)
- SF_HEAP_AUTO_NEW(ptr) ClassName

このようなマクロはその兄弟である SF_HEAP_ALLOC や SF_HEAP_NEW とほぼ同様に動作しますが、1 つ違いがあります。それは、これらのマクロがヒープへのポインタではなく、メモリ ロケーションへのポインタを取ることです。呼び出されると、これらのマクロは自動的に与えられたメモリ アドレスに基づいてヒープを識別し、同じヒープからアロケーションを行います。以下の例について考えてみましょう:

```
MemoryHeap*    pheap = …;
UByte *         pbuffer = (UByte*)SF_HEAP_ALLOC(pheap, 100,
StatMV_ActionScript_Mem);
Ptr<Sprite> sprite = *SF_HEAP_AUTO_NEW(pbuffer + 5) GFxSprite();
…
sprite->DoSomething();
...
// Release sprite and free buffer memory.
sprite = 0;
SF_FREE(pbuffer);
```

この例では、指定されたヒープからメモリ バッファを作成して、次にそのメモリ バッファと同じヒープに GFx::Sprite オブジェクトを作成します。この例で特異な点は、'pbuffer' ポインタを SF_HEAP_AUTO_NEW にただ渡すだけではなく、バッファ アロケーション「の中で」アドレスを渡すということです。このコードが間違っているわけではありません。むしろ、Scaleform メモリ ヒープ システムの革新的な機能を表しているのです。それは、メモリ割り当て内の任意のアドレスに基づいてメモリ ヒープを識別する機能です (しかも、非常に能率的に行います)。この独特の機能は、この章の冒頭で説明したヒープ渡し問題の優れた解決策の鍵となります。自動ヒープ識別で強化されているので、その位置に基づいた正確なヒープから自動的にメモリを割り当てるコンテナを作成すること

ができます。言い換えると、以前提示した GFx::Sprite クラスを以下のように書き直すことができます:

```
class Sprite : public RefCountBase<Sprite>
{
    StringLH                  Name;
    ArrayLH<Ptr<Sprite> > Children;

    Sprite(String& name, …) { … }
};
Ptr<Sprite> sprite = *SF_HEAP_NEW(pHeap) Sprite(name);
sprite->DoSomething();
```

この例では、Scaleform::String クラスと Scaleform::Array<>クラスを、同等の「ローカル ヒープ」である Scaleform::StringLH と Scaleform::ArrayLH<>に置き換えました。これらのローカル ヒープ コンテナは自らが含まれているオブジェクトと同じヒープから、上記の「自動ヒープ」技術を使ってメモリを割り当てます。すべて、それ以外では必要となる追加の引数渡しオーバーヘッドなしで行います。メモリが最後に必ず正しいヒープに行くために必要な、このようなコンテナや同様のコンテナを Scaleform コア全体で目にするでしょう。

2.4.3ヒープトレーサー

GFxConfig.h で定義されている SF_MEMORY_TRACE_ALL をイネーブルすると必須のメモリー割り当てをトレースできます。CreateHeap、DestroyHeap、Alloc、Free などのメモリー操作は、Scaleform::Memory クラスで SetTracer 方式を呼び出してトレースできます。

SysAllocWinAPI の場合にトレーサーをどのように使えば良いかは、次のコードで示します。

```
#include "Kernel/HeapPT/HeapPT_SysAllocWinAPI.h"

class MyTracer : public Scaleform::MemoryHeap::HeapTracer
{
 public:
    virtual void OnCreateHeap(const MemoryHeap* heap)
    {
        //...
    }

    virtual void OnDestroyHeap(const MemoryHeap* heap)
    {
        //...
    }

    virtual void OnAlloc(const MemoryHeap* heap, UPInt size, UPInt align,
                                unsigned sid, const void* ptr)
    {
```

```
        //...
    }

    virtual void OnRealloc(const MemoryHeap* heap, const void* oldPtr,
                           UPInt  newSize, const void* newPtr)
    {
        //...
    }

    virtual void OnFree(const MemoryHeap* heap, const void* ptr)
    {
        //...
    }
};

static MyTracer tracer;
static SysAllocWinAPI sysAlloc;
static GFx::System* gfxSystem;


class MySystem : public Scaleform::System
{
 public:

    MySystem() : Scaleform::System(&sysAlloc)
    {
        Scaleform::Memory::GetGlobalHeap()->SetTracer(&tracer);
    }
    ~MySystem() { }
};


SF_PLATFORM_SYSTEM_APP(FxPlayer, MySystem, FxPlayerApp)
```

3 メモリレポート

以前 Scaleform 3.0 で Scaleform Player AMP HUD の一部として導入した詳細なメモリレポート機能は、現在ではプロファイリング機能として、パソコンとコンソール Scaleform アプリケーションをリモートで接続するスタンドアロンの AMP アプリケーションに移動しています。AMP の使用方法については「AMP User Guide」を参照してください。

軽量化のために Scaleform Player から HUD UI を移動していますが、Ctrl＋F5 キーを押すことでメモリレポートが生成されコンソールに送られます。このレポートのフォーマットは、Scaleform::MemoryHeap::MemReport 関数が生成するフォーマットのひとつと同じです。

```
void MemReport (class StringBuffer& buffer, MemReportType detailed,
                    bool xmlFormat = false);
void MemReport (struct MemItem* rootItem, MemReportType detailed);
```

MemReport はメモリ レポートを生成し、与えられたストリング バッファに、おそらくは XML 形式で書き込みます。続いてこのストリングは、出力コンソールまたはスクリーンに、デバッグを目的として書き込まれます。MemReportType 引数は次のいずれかです。

- MemoryHeap::MemReportBrief – Scaleform システムが消費した全メモリのサマリです。以下に例を示します。

```
Memory 4,680K / 4,737K
    Image                                       1,052
    Sound                                       136
    Movie View                                  1,739,784
    Movie Data                                  300,156
```

- MemoryHeap::MemReportFull – Ctrl＋F6 キーを押下したときに Scaleform Player が出力するレポートと同じです。システムメモリ サマリと個々のヒープメモリ サマリ、および、SF_MEMORY_ENABLE_DEBUG_INFO が定義されていれば、stat ID ごとの割り当てメモリの内訳で構成されます。ヒープによって与えられる情報の例は次のとおりです。

```
Memory 4,651K / 4,707K
    System Summary
        System Memory FootPrint                 4,832,440
        System Memory Used Space                4,775,684
        Debug Heaps Footprint                   12,884
        Debug Heaps Used Space                  5,392
    Summary
        Image                                   1,052
        Sound                                   136
        Movie View                              1,715,128
```

```
    Movie Data                                  300,156
[Heap] Global                                   4,848,864
    Heap Summary
        Total Footprint                         4,848,864
        Local Footprint                         2,427,860
        Child Footprint                         2,421,004
        Child Heaps                             4
        Local Used Space                        2,417,196
        DebugInfo                               120,832
    Memory
        MovieDef
            Sounds                              8
            ActionOps                           80,476
            MD_Other                            200
        MovieDef                                36
        MovieView
            MV_Other                            32
        General                                 91,486
        Image                                   92
        Sound                                   136
        String                                  31,425
        Debug Memory
            StatBag                             16,384
        Renderer
            Buffers                             2,097,152
            RenderBatch                         5,696
            Primitive                           1,512
            Fill                                900
            Mesh                                4,884
            MeshBatch                           2,200
            Context                             15,184
            NodeData                            15,160
            TreeCache                           13,220
            TextureManager                      2,336
            MatrixPool                          4,096
            MatrixPoolHandles                   2,032
            Text                                1,232
        Renderer                                13,488
    Allocations Count                           1,822
```

- MemoryHeap:: MemReportFileSummary – ファイルが消費するメモリを出力します。それぞれの SWF に対して、MovieData、MovieDef、および各 MovieView の値を statID 別に表示します。フォーマットは次のとおりです。

```
Movie File 3DGenerator_AS3.swf
    Memory                                      1,956,832
```

```
        MovieDef                        152,664
            CharDefs                    39,848
            ShapeData                   1,716
            Tags                        73,656
            MD_Other                    37,444
        MovieView                       196,249
            MovieClip                   120
            ActionScript                181,917
            MV_Other                    14,212
        General                         1,596,712
        Image                           960
        String                          2,783
        Renderer                        7,464
            Text                        7,464
```

- MemoryHeap::MemReportHeapDetailed – AMP が Memory タブに表示するメモリタイプです。ヒープごとの使用メモリ量、デバッグ情報の使用メモリ量、Scaleform によって要求されていて現在使用されていないメモリ量を表示するもっとも詳細なレポートです。たとえば、

```
Total Footprint                         4,858,148
    Used Space                          4,793,972
       Global Heap                      2,405,492
           MovieDef                     80,720
               Sounds                   8
               ActionOps                80,476
               MD_Other                 200
           MovieView                    32
               MV_Other                 32
           General                      92,566
           Image                        92
           Sound                        136
           String                       31,276
           Debug Memory                 8,192
               StatBag                  8,192
           Renderer                     2,182,960
               Buffers                  2,097,152
               RenderBatch              5,696
               Primitive                1,512
               Fill                     900
               Mesh                     4,884
               MeshBatch                2,200
               Context                  15,184
               NodeData                 15,016
               TreeCache                13,136
               TextureManager           2,336
               MatrixPool               8,192
               MatrixPoolHandles        2,032
```

```
            Text                            1,232
        Movie Data Heaps                    300,156
            MovieData "3DGenerator_AS3.swf" 300,156
                MovieDef                    152,664
                    CharDefs                39,848
                    ShapeData               1,716
                    Tags                    73,656
                    MD_Other                37,444
                General                     141,332
                Image                       960
                String                      2,312
        Movie View Heaps                    1,727,936
            MovieView "3DGenerator_AS3.swf" 1,727,936
                MovieView                   195,977
                    MovieClip               120
                    ActionScript            181,645
                    MV_Other                14,212
                General                     1,467,668
                String                      471
                Renderer                    7,464
                    Text                    7,464
        Other Heaps                         360,580
            _FMOD_Heap                      360,580
                General                     359,944
    Debug Data                              12,884
    Unused Space                            51,292
        Global                              51,292
            _FMOD_Heap                      4,384
            MovieData "3DGenerator_AS3.swf" 5,356
            MovieView "3DGenerator_AS3.swf" 33,032
```

上述のとおり、Scaleform を SF_MEMORY_ENABLE_DEBUG_INFO を使ってコンパイルした場合、メモリシステムは、アロケーションの目的を示す「stat ID」タグを使って、アロケーションをマークします。

- MovieDef – ロードされたファイルデータを表します。ファイルが完全にロードされたあとは、このメモリが増加することはありません。
- MovieView – GFx::Movie インスタンスデータで構成されます。ActionScript が多くのオブジェクトをアロケート中のとき、または、ActionScript がアロケートしていないオブジェクトのムービークリップが多数ステージ上で動作しているとき、このカテゴリのサイズは増加します。
- CharDefs – 個々のムービークリップおよび他のフラッシュオブジェクトを定義します。
- ShapeData – コンプレックスシェープとフォントのベクター表現です。
- Tags – アニメーションキーフレームデータ。
- ActionOps – ActionScript バイトコード。
- MeshCache – ベクターテッセレータと EdgeAA が生成した、キャッシュ済みのシェープメッシュデータで構成されます。

- FontCache – キャッシュ済みのフォントデータで構成されます。

4 ガーベージ コレクション

Scaleform バージョン 2.2 以前は、ActionScript オブジェクトに対して、簡単な参照カウント メカニズムを使用していました。ほとんどの場合はこれで十分なのです。ただし、ActionScript を使うと、2 つ以上のオブジェクトがお互いの参照を持つ場合に、いわゆる「循環参照」を作成することができてしまいます。この循環参照はシステムのパフォーマンスに影響するメモリ リークを起こします。
。このようなオブジェクト参照の 1 つが明確に解除される場合を除き、リークを起こすコードの例について考えてみます。

Code:

```
var o1 = new Object;
var o2 = new Object;
o1.a = o2;
o2.a = o1;
```

循環参照を避けるように、あるいは参照のサイクルでオブジェクトの接続を解除するクリーンアップ関数を持つように、このようなコードを書き直すことが、理論的には可能です。しかし、ほとんどの状況ではクリーンアップ関数はうまく動作せず、ActionScript のクラスやコンポーネントが使用中の場合は、問題がさらに悪化します。メモリ リークが起きる可能性のある一般的なケースには、シングルトンの使用や標準の Flash UI コンポーネントの使用も含まれます。

循環参照を解消するため、Scaleform 3.0 からは「ガーベージ コレクション」という高度に最適化された参照カウント ベースのクリーンアップ メカニズムを導入しています。Scaleform のガーベージ コレクションは一般的なガーベージ コレクションに似ていますが、参照カウントと併用するように最適化されています。ユーザーの ActionScript が「循環参照」を作成しない場合は、このメカニズムは標準の参照カウント システムとして動作します。循環参照が作成された場合、コレクタが実行され、必要なときにこのようなオブジェクトを解放します。ほとんどの Flash ファイルの場合、あったとしてもパフォーマンスへの影響は非常に小さなものになります。

デフォルトでは、Scaleform 3.0 以降のバージョンで循環コレクションはオンになっています。循環参照によって発生していた以前のメモリ リークはすべて、ガーベージ コレクションを使用すると解消されます。必要ない場合、ガーベージ コレクションの機能をオフにすることができます。そのためには Scaleform をリビルドしなければならないため、これができるのはソース コードにアクセスできるユーザーだけです。

ガーベージコレクションをオフにするには、*Include/GFxConfig.h* ファイルを開き、GFX_AS_ENABLE_GC マクロの行をコメントする必要があります (デフォルトではコメントアウトされていません)。

```
// Enable garbage collection
```

```
#define GFX_AS_ENABLE_GC
```

このマクロをコメント アウトした後で、完全な Scaleform ライブラリの再構築を行う必要があります。

4.1 ガーベージ コレクションを使えるようにメモリ ヒープを構成する

Scaleform 3.x では、ガーベージ コレクションをホストするメモリ ヒープの構成を行えるようになっています。Scaleform 3.3 以上ではヒープを共有できるため、複数の MovieView 間でガーベージコレクタを共有することが可能です。

所定の MovieDef に対して新しいヒープを生成するには以下の関数を使用します。

```
Movie* MovieDef::CreateInstance(const MemoryParams& memParams,
                                        bool initFirstFrame = true)
```

最初の引数となる MemoryParam 構造は、以下のような構造として定義されます：

```
struct MemoryParams
{
      MemoryHeap::HeapDesc   Desc;
      float                   HeapLimitMultiplier;
      unsigned                MaxCollectionRoots;
      unsigned                FramesBetweenCollections;
};
```

- `Desc` – ムービー用に作成されるヒープのための記述子です。これによって、ヒープアラインメント（`MinAlign`）、粒度（`Granularity`）、リミット（`Limit`）、フラグ（`Flag`）を指定できます。ヒープ リミットが指定されると、Scaleform はこのリミット以下に、ヒープのフットプリントを抑えようとします。
- `HeapLimitMultiplier` – 浮動小数点のヒープ リミット乗数です（0…1）。リミットを超えてしまった場合に、ヒープ リミットがどのように増加するかを決定します。詳細は以下をご覧ください。
- `MaxCollectionRoots` – ガーベージ コレクションが実行される以前のルートの数。最大ルートの上限数の初期の値です。ガーベージ コレクションが実行中は、増大することになります。詳細は、以下をご覧ください。
- `FramesBetweenCollections` – 最大ルート上限に到達していないときでもガーベージ コレクションが強制的に実行された後のフレーム数。最大ルート上限を高めに設定してあるときに、中間的なガーベージ コレクションを実行すると、そのコレクション実行時のコストを低く抑えることができます。詳細は以下をご覧ください。

同様に、movie view ヒープはふたつのステップで生成します。

```
MemoryContext* MovieDef::CreateMemoryContext(
```

```
                                                const char* heapName,
                                                const MemoryParams& memParams,
                                                bool debugHeap )

Movie* MovieDef::CreateInstance(MemoryContext* memContext,
                                            bool initFirstFrame = true)
```

MemoryContext オブジェクトは、movie view ヒープと、ガーベージコレクタのようなヒープ固有の他のオブジェクトをカプセル化します。メモリコンテクストを通じて movie view とそのヒープを生成するこの後者のアプローチには、そのヒープがスレッドセーフかどうかに関係なく、かつ、そのヒープがデバッグとしてマークされていて AMP レポートから除外されるかどうかに関係なく、AMP で表示されるヒープ名を指定できるという自由度があります、より重要な点は、同一スレッド上の複数の movie view に対して、単一ヒープ、ガーベージコレクタ、ストリングマネージャ、およびテキストアロケータの共有を許可していることであり、オーバーヘッドの抑制につながります。

`MemoryParams::Desc` は、例えば ActionScript のアロケーションなどの、ムービー インスタンスに使われるメモリ ヒープの一般的なプロパティを指定するために使用します。`Desc.Limit` と `HeapLimitMultiplier` という二つのパラメータを設定してヒープのフットプリントの大きさを管理することができます。

あるヒープが初期設定されたリミットを 128K としているとします（これを "ダイナミック" リミットと呼びます）。　このリミットを超えてしまった場合、特別な内部ハンドラが呼び出されます。このハンドラは：　（メモリ）スペースを解放するか、あるいはヒープを拡大するか決定するロジックを持っています。（メモリを）解放するのか、（ヒープを）拡大するか決定するための経験則（ヒューリスティック）は、Boehm-Demers-Weiser(BDW)のガーベージコレクターとメモリ アロケータから取り込んできています。

BDW アルゴリズムは、以下のようなものです（疑似コードです）：

```
if (allocs since collect >= heap footprint * HeapLimitMultiplier)
    collect
else
    expand(heap footprint + overlimit + heap footprint *
           HeapLimitMultiplier)
```

“collect”のステージは、ActionScript ガーベージ コレクタの起動と、さらに他のメモリを解放してやることも含んでいます。例えば、内部キャッシュをフラッシュすることなどです。
`HeapLimitMultiplier` のデフォルト値は、0.25 です。よって、最後のメモリのガーベージコレクション以降の取得フットプリントが、現在のヒープのフットプリントの 25%（これは、HeapLimitMultiplier のデフォルト値です）を超えたときに初めて Scaleform はメモリ解放を実行します。または、Scaleform は、要求されたサイズにヒープ フットプリントの 25%を加えたところまでリミットを拡大します。

ユーザーが指定の `Desc.Limit` を保持している場合には、上記のアルゴリズムは、上記同様に、そこのリミットに到達するまで働きます。その `Desc.Limit` をヒープ リミットが超えた時には、最後のコ

レクション作業以降に取得されたアロケーション数に関わりなく、ガーベージ コレクションが起動されます。ダイナミック ヒープ リミットは、コレクション作業後のヒープのフットプリントサイズに、要求されたアロケーションに必要な差分を加えたサイズに設定されます（アロケーションに必要なものより小さなメモリがコレクション作業で解放されている場合です）。
ガーベージ コレクタを管理する次の方法は、対となっている `MaxCollectionRoots/FramesBetweenCollections` を指定するものです。

`MaxCollectionRoots` は、そこを超えるとガーベージ コレクションの起動が引きおこされる指定のルート数を決定します。ここで、「ルート」という意味は、他の ActionScript オブジェクトとの間で循環参照を形成してしまう可能性を持った ActionScript を言います。普通、一個の ActionScript オブジェクトは、ActionScript にタッチされるとルートの配列に付け加えられます。（つまり、ActionScript は、そのオブジェクトを参照することになるのです。例えば "obj.member=1" という ActionScript は、"obj" というオブジェクトにタッチしています。）基本的に、私たちは、タッチされた ActionScript オブジェクトは、全てそれぞれ一個のルートだと考えてもよいでしょう。そこで、MaxCollectionRoots は、そのときに ActionScript が働きかけているオブジェクトの概算数を特定するわけです。こうして、MaxCollectionRoots が設定した値を、タッチされたオブジェクトの数が超えてしまうとガーベージ コレクタが起動されます。デフォルトでは、このパラメータは 0 に設定されています。つまり、このメカニズムはデフォルトではオフになっています。

`FrameBetweenCollections` は、強制的にガーベージ コレクションが起動されるまでのフレーム数を設定します。ここで、"フレーム" という用語は、単一の Flash フレームを指します。ある SWF ファイルのフレーム レートが 30fps だとしましょう。このとき、1 秒は 30 フレームに等しく、2 秒は 60 フレームということになります。この値は、Advance のパフォーマンスに大きなスパイクが起こってしまうことを防止することもできます。そういった瞬時の過渡現象は、ガーベージ コレクタがたくさんのオブジェクトを精査するのに忙しいような場合に発生してしまう可能性があります。こういった時、この値は、1800 程度に設定するのが良いでしょう。Flash のフレーム レートを 30fps として、ガーベージ コレクタは 60 秒毎に起動されることになります。`FrameBetweenCollections` のパラメータは、`Desc.Limit` が設定されていない場合に ActionScript のメモリヒープが過度に増大してしまうことを防ぐためにも使うことができます。デフォルトでは、このパラメータは 0 です。このメカニズムはデフォルトではオフになっています。

4.2 GFx::Movie:: ForceCollectGarbage

```
virtual void  ForceCollectGarbage() = 0;
```

このメソッドを使って、アプリケーションからガーベージ コレクタを強制的に実行させることができます。ガーベージ コレクションがオフの場合は何も起きません。