Autodesk®
Scaleform®

메모리 시스템 개요

이 문서는 Scaleform 3.3 에서 메모리를 설정, 최적화, 관리하는 방법을 설명한다.

집필: Michael Antonov, Maxim Shemanarev

버전: 4.01

최종 편집: 2011 년 2 월 17 일

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.

연락처:

문서	메모리 시스템 개요
주소	Autodesk Scaleform Corporation
	6305 Ivy Lane, Suite 310
	Greenbelt, MD 20770, USA
웹사이트	www.scaleform.com
이메일	info@scaleform.com
직통전화	(301) 446-3200
팩스	(301) 446-3199

목차

1 Scaleform 3.x 메모리 시스템

Autodesk® Scaleform® 3.0 에서는 메모리 할당이 힙(heap) 기반 전략으로 구현 되었으며, 할당 및 힙에 대한 상세 메모리 보고가 도입되었습니다. 메모리 보고는 프로그램적으로나 Scaleform 3.2 와 함께 제공되는 단일 실행 응용 프로그램인 AMP(Analyzer for Memory and Performance) 도구를 통해서 사용할 수 있습니다. 새로 배포된 Scaleform 3.3 에서는 메모리 인터페이스가 업데이트되어 비효율적이었던 이전 버전의 힙 구현이 개선되었습니다. 이러한 변경 사항에 대해서는 섹션 1.2 를 참고하십시오.

이 문서는 Scaleform 3.3 메모리 시스템에 대해 기술하며, 다음의 핵심 영역에 대한 상세 정보를 제공합니다.

- 오버라이드 메모리 할당 인터페이스
- Movies 와 Scaleform 하위 시스템의 힙 기반 메모리 관리
- ActionScript 조각모음(Garbage Collection)
- 사용 가능한 메모리 보고 기능

1.1 중요한 메모리 개념

이 섹션은 Scaleform 메모리 관리 개념 및 개념 간의 상호 작용, 메모리 사용에 미치는 영향에 대해 심도 있게 다룹니다. 할당자 커스터마이징 및/또는 Scaleform 메모리 사용 프로파일링을 하기 전에 이러한 개념을 이해해둘 필요가 있습니다.

1.1.1 오버라이드 할당자

Scaleform 가 생성하는 모든 외부 메모리 할당은 시작되는 동안 GFx::System 에 설치된 중앙 인터페이스를 통과하여 개발자 고유의 메모리 관리 시스템을 플러그인할 수 있도록 해줍니다. 메모리 관리는 다음 몇 가지 다른 접근을 통해 커스터마이징할 수 있습니다.

1. 개발자들은 Scaleform::SysAlloc 인터페이스를 설치하고 해당 인터페이스의 Alloc/Free/Realloc 기능을 구현하여 메모리 할당자로 위임할 수 있습니다. 이것은 응용 프로그램이 dlmalloc 처럼 중앙화된 할당자 구현을 사용할 때 매우 적절한 방법입니다.
2. Scaleform 는 시작 시 사전 할당된, 하나 이상의 커다란 고정 크기 메모리 블록으로 전달될 수 있습니다. 추가 블록은 "일시중지 메뉴"와 같은 일시적인 화면을 위해 예약되어 있으며 메모리 영역 사용을 통해 완전히 해제됩니다.
3. Scaleform 이 제공한 시스템 할당자 구현(예: Scaleform::SysAllocWinAPI)은 하드웨어 페이징을 이용하기 위해 사용될 수 있습니다.

응용 프로그램에서는 메모리 전략에 따라 선택된 접근을 채용합니다. 사례 (2)는 하위 시스템을 위한 고정적 메모리 예약 방식이 적용된 콘솔에서는 일반적입니다. 다른 구현에 대해서는 섹션 2 '메모리 할당'에서 자세히 다루도록 합니다.

1.1.2 메모리 힙

개발자들이 외부적으로 메모리 관리 접근을 선택하는 것과는 별개로, 모든 내부 Scaleform 할당은 파일 및 목적에 따라 쪼개진 힙으로 편성되며, 개발자들은 AMP 또는 MemReport 출력을 관찰하는 동안 이러한 점을 발견할 수 있습니다. 가장 일반적인 힙은 다음과 같습니다.

- Global 힙 - 모든 공유 할당과 구성 오브젝트를 보유합니다.
- MovieData 힙 - 개별 SWF/GFx 파일에서 불러온 읽기 전용 데이터를 저장합니다.
- MovieView 힙 - 타임라인과 인스턴스 데이터를 포함한 단일 GFx::Movie ActionScript 샌드박스를 대표합니다. 이 힙은 내부 조각모음에 영향을 받습니다.
- MeshCashe - 출력되는 shape triangle data 는 이곳에 할당됩니다.

메모리 힙 구현은 보통 "페이지"라고 부르는, 주어진 파일 하위 시스템 전용 메모리 블록의 할당을 제공함으로써 작동합니다. 이것은 여러 가지 장점이 있습니다.

- 외부 시스템 할당의 수를 경감시켜 성능이 향상됩니다.
- 오직 하나의 쓰레드만 액세스한 힙 내부의 쓰레드 동기화를 제거합니다.
- 관련된 데이터를 하나의 개체로 해방하여 외부 단편화를 줄입니다.

- 메모리 보고의 논리적 구조를 강화합니다.

하지만, 힙이 내부 단편화에 영향을 받는다는 점에서 이것은 큰 단점이 될 수 있습니다. 내부 단편화는 힙 내부에서 해방된 소형 메모리 블록이 다른 응용 프로그램으로 해제되지 않을 때 발생하는데, 이것은 연관된 모든 페이지가 해방되지 않아 힙이 비효율적으로 "사용하지 않는" 메모리를 보유하는 결과를 가져왔기 때문입니다.

Scaleform 3.3 은 보다 적극적으로 다량의 메모리를 해방하는 새 Scaleform::SysAlloc 기반 힙 구현을 통해 이 문제를 해결합니다. 또한, Scaleform 3.3 은 같은 쓰레드 상에 GFx::Movie 를 위치하게 함으로써 단일 메모리 메모리를 공유하게 하는데, 이를 통해 부분적으로 채워진 블록에 메모리가 걸리는 현상을 줄일 수 있습니다.

1.1.3 조각모음

Scaleform 3.0 는 AS(ActionScript) 조각모음을 도입하여 AS 데이터 구조 내의 선형 참조로 인해 일어나는 메모리 누수를 제거합니다. 적절한 조각모음은 CLIK 및 관련된 ActionScript 프로그램의 연산에 매우 중요합니다.

Scaleform 에서 조각모음은 GFx::Movie 오브젝트에 포함되며, AS 가상머신을 위한 실행 샌드박스 역할을 합니다. Scaleform 3.3 을 이용하면 Movie 힙을 나누는 것은 물론, 연관된 가비지 콜렉터(garbage collector)를 나누고 통합할 수 있습니다. 보통 조각모음은 ActionScript 할당에 의해 Movie 힙이 커졌을 때 자동으로 실행되지만, 명시적 프레임 기반 간격에 따라 수집하도록 실행시키거나 Scaleform 에 명령하는 것도 가능합니다. 조각모음에 대한 상세 내용 및 구성 옵션은 섹션 4: 조각 모음에 기술되어 있습니다.

1.1.4 메모리 보고 및 디버깅

Scaleform 메모리 시스템은 할당에 "stat ID" 태그를 붙일 수 있는데, 이 태그에는 각 할당에 대한 목적이 기술되어 있습니다. 할당된 메모리는 stat ID 카테고리로 또는 stat ID 에 따라 쪼개진 힙에 의해 보고될 수 있으며 힙으로 쪼개집니다. 이 메모리 보고 기능은 Scaleform 3.2 와 함께 제공되는 단일

실행 프로파일링 응용 프로그램인 AMP(Analyzer for Memory and Performance)에 의해 첫 번째로 노출됩니다. AMP 의 사용에 대한 상세 정보는 AMP 사용자 설명서를 참조하십시오. 메모리 보고는 프로그램적으로도 얻을 수 있으며, Scaleform::MemoryHeap::MemReport 함수를 호출하여 문자열 형식으로 얻습니다.

위에서 언급한 stat ID 태그는 실제 메모리 할당과 별도로 디버그 데이터로서 저장됩니다. 또한, 디버그 데이터는 메모리 누수 탐지를 위해서 보관됩니다. Scaleform 디버그 버전이 셧다운되면 누수 메모리 보고가 Visual Studio 결과 창 또는 콘솔로 생성됩니다. Scaleform 에서는 어떠한 알려진 내부 메모리 누수도 발생하지 않으므로, 탐지된 모든 누수는 Scaleform 오브젝트에 의지하는 참조의 부적절한 사용에 의해 발생했을 가능성이 큽니다. 그 밖의 관련된 디버그 데이터는 Shipping 구성에 할당되지 않습니다.

1.2 Scaleform 3.3 메모리 API 변경점

앞서 언급한 것처럼 Scaleform 3.3 에는 메모리 사용 향상을 위한 두 가지 중대한 업데이트가 포함되어 있습니다.

- GFx::Movie 메모리 컨텍스트 쉐어링은 독립적인 movie view 인스턴스가 힙과 조각모음을 공유하도록 하여 메모리 재사용을 개선합니다. 내부 문자열 테이블 역시 공유되어 추가적인 절약이 가능합니다. 이러한 공유에 대해서는 섹션 4 에서 상세하게 다룹니다.
- SysAlloc 인터페이스는 훨씬 "malloc 친화적으로" 업데이트되었는데, 외부 할당에 대한 4K 페이지 정렬 요건을 제거하고 힙을 최적화하여 512 바이트보다 큰 모든 할당을 직접 SysAlloc 으로 보냅니다. 이러한 접근은 더 많은 메모리를 적극적으로 응용 프로그램에 반환하여 효율과 재사용을 증대합니다.

기존의 Scaleform::SysAlloc 구현은 Scaleform::SysAllocPaged 로 명칭이 변경되었으며 여전히 사용 가능합니다. SysAlloc 을 오버라이딩할 경우 해당 구현을 업데이트하거나 기본 클래스를 SysAllocPaged 로 변경하면 됩니다. SysAllocStatic 와 메모리 영역 클래스는 그대로 사용할 수 있으므로, 해당 클래스를 사용하는 사용자는 영향을 받지 않습니다.

2 메모리 할당

섹션 1.1.1 에서 다룬 것처럼, Scaleform 에서는 세 가지 방법 중 하나를 선택하여 메모리 할당을 커스터마이징할 수 있습니다.

1. 응용 프로그램 메모리 시스템으로 할당을 위임하는 SysAlloc 인터페이스를 오버라이딩
2. Scaleform 와 함께 하나 이상의 고정 크기 메모리 블록을 제공
3. Scaleform 가 운영 체제에서 곧바로 메모리를 할당하도록 명령

메모리 효율성을 최대로 높이려면 개발 중인 응용 프로그램에 알맞은 접근법을 선택해야 합니다. 일반적으로 사용되는 접근법 (1)과 (2)의 차이점에 대해서는 아래에서 다룹니다.

개발자들은 SysAlloc 을 오버라이드할 때 자신의 메모리 시스템에 Scaleform 할당을 위임합니다. 예를 들어, 새 SWF 파일을 불러올 메모리가 필요할 때 Scaleform 는 Scaleform::SysAlloc::Alloc 함수를 호출하여 불러오기를 요청합니다. 해당 콘텐츠가 내려졌을 때는 Scaleform::SysAlloc::Free 함수를 호출합니다. 이 시나리오에서 제일 가능한 메모리 효율은 Scaleform 를 포함한 모든 시스템이 단일 공유 전역 할당자(single shared global allocator)를 사용하고 해제된 메모리 블록이 그것을 필요로 하는 어떤 시스템에 의해 재사용되도록 하는 것입니다. 할당의 구역화는 어떤 것이든 전체적인 공유 효율성을 떨어뜨리고, 그러므로써 총 메모리 사용량이 증가하게 됩니다. SysAlloc 오버라이딩에 대해서는 다음 섹션 2.1 에서 자세하게 다룹니다.

전역 할당 접근의 가장 큰 문제점은 단편화이며, 이것은 많은 콘솔 개발자들이 사전에 결정된 고정 메모리 레이아웃을 대안으로 선호하는 이유입니다. 사전에 결정된 메모리 레이아웃과 함께 고정 크기 메모리 지역은 각 시스템에 할당되며, 이러한 지역의 개별 크기는 실행 시 또는 레벨을 불러오는 중에 결정됩니다. 결국, 이러한 접근은 더 예측 가능한 메모리 레이아웃을 전역 시스템의 효율성과 맞교환 하는 것입니다.

응용 프로그램이 이러한 고정 메모리 접근을 사용할 경우, 개발자는 섹션 2.2 에 기술된 구성뿐 아니라 Scaleform 에도 반드시 고정 크기 메모리 블록 할당자를 사용해야 합니다. 이 시나리오를 좀 더 실용적으로 만들기 위해서 Scaleform 는 2 차 메모리 영역 또는 "일시중지 메뉴"가 표시될 때처럼 제한된 시간 동안 사용되는 메모리 지역의 생성을 허용합니다.

2.1 할당 오버라이딩

외부 Scaleform 할당자를 대체하려면 다음의 두 가지 단계를 따릅니다.

1) 자신만의 SysAlloc 을 구현한다. Alloc, Free, Realloc 메소드 포함
2) Scaleform 초기화 중에 GFx::System 생성자에 이 할당자의 인스턴스 넘기기

정렬 지원과 함께 제공되는 특정 시스템의 대안 및 표준 malloc/free 에 의존하는 기본 Scaleform::SysAllocMalloc 구현은 Scaleform 에서 제공되며 직접적으로 또는 참고사항으로 사용될 수 있습니다.

2.1.1 자신만의 SysAlloc 구현

자신만의 메모리 힙을 구현하는 가장 쉬운 방법은 Scaleform 의 SysAllocMalloc 을 복사해서 다른 메모리 할당자를 호출하도록 수정하는 것이다. 약간 수정된 특정 윈도우에 대한 SysAllocMalloc 구현의 예는 다음과 같다.

```
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);
    }
};
```

보이는 것처럼 할당자 구현은 매우 간단한다. MySysAlloc 클래스는 SysAlloc 를 기반으로 상속받아서 3 개의 가상함수를 구현한다(`Alloc, Free, Realloc`). 이러한 함수의 구현은 반드시 정렬을 수행해야 하지만, Scaleform 는 대체로 16 바이트 이하의 작은 정렬에만 요청합니다. oldSize 와 align 전달 인자는 Free/Realloc 인터페이스로 넘어가 구현을 단순화합니다. 이들은 예를 들어, Alloc/Free 를 호출하기 위한 wrapper 로 Realloc 을 구현할 때 도움이 됩니다.

개발자들이 자신의 할당자의 인스턴스를 한번 생성하면, 그것은 Scaleform 의 초기화 과정에서 GFx::System 의 생성자에 전달 될 수 있다.

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

Scaleform 3.0 에서 GFx::System 객체는 어떤 Scaleform 객체들 보다 먼저 생성되어야 하며, 모든 Scaleform 객체들이 해제된 다음에 삭제되어야 한다. Scaleform 를 사용하는 코드를 사용하는 함수에서 할당을 초기화 할때는 이렇게 하는 것이 최선이다. 하지만, Scaleform 보다 더 오랫동안 살아남아야 하는 할당 객체가 있을 수도 있다(GFx::System 은 전역으로 선언하지 **않는** 것이 좋다). 만약 이렇게 사용하는 것이 불편하다면 객체 생성없이 GFx::System::Init()이나 GFx::System::Destroy()같은 정적 함수를 대신 사용할 수 있다. GFx::System 생성자와 비슷하게 GFx::System::Init()함수도 SysAlloc 포인터를 전달인자로 갖는다.

2.2 Scaleform 에 고정 메모리 블록 사용하기

도입부에서 논의한 것처럼 콘솔 개발자들은 선행적으로 1 개 이상의 메모리 블록을 예약하는 방법을 택하여, SysAlloc 을 오버라이딩하는 대신 Scaleform 에 해당 메모리 블록을 보낼 수 있습니다. 이것은 다음과 같이 SysAllocStatic 을 인스턴트화하여 실행합니다.

```
void*            pmemChunk = …;
SysAllocStatic   blockAlloc(pmemChunk, 6*1024*1024);
System           gfxSystem(&blockAlloc);
…
```

위의 예제는 6 메가 메모리 청크를 Scaleform 가 할당에 사용하도록 넘기고 있다. 물론, 이 메모리 블록은 `gfxSystem` 과 `blockAlloc` 객체가 스코프를 벗어날 때 까지는 재사용되거나 해제될 수 없다. 하지만, 일시중지 메뉴를 표시하는 등의 일시적 목적으로 여분의 "해제 가능" 메모리 블록을 추가하는 것도 가능합니다. 이러한 블록은 메모리 영역 사용을 통해 지원되는데, 자세한 내용은 아래에서 다룹니다.

정적 할당자를 사용할때는 개발자가 항상 Scaleform 에 의해서 사용되는 메모리 양에 대해서 주의해야 한다. 당연히 파일을 로드하거나 무비 인스턴스가 생성될 때 지정된 메모리 크기를 넘어가지 않도록 해야한다. SysAllocStatic 이 실패하면 `Alloc` 구현함수가 0 을 반환 할 것이며, Scaleform 도 크래쉬가 발생할 것이다. 필요하다면 간단한 SysAlloc Paged 래퍼 객체를 사용해서 이러한 심각한 상황을 탐지할 수도 있다.

2.2.1 메모리 아레나 (Memory Arenas)

Scaleform 3.1 에서는 프로그램 실행 중 특정 포인트에서 완전히 해제될 수 있게 해주는 사용자 정의 할당 영역인 메모리 아레나라는 지원 기능을 추가했다. 조금 더 자세히 말하면 메모리 아레나는 Scaleform 파일의 로딩 및 GFx::Movie 가 생성 될 수 있는 메모리 영역을 정의함으로써 이러한 영역들을 차지하고 있던 Scaleform 객체를 소멸하여 완전히 해제될 수 있게 해준다. 메모리 아레나는 Scaleform 의 나머지 부분을 종료하거나 관련 없는 Scaleform 파일을 언로딩 하지 않고서도 소멸 시킬 수 있다. 아레나가 소멸되면 어플리케이션은 기타 비 Scaleform 데이터를 위해 메모리를 재활용할 수 있다. 사용 사례의 하나로 메모리 아레나는 메모리를 잠시 요구하는 게임의 "일시 정지"스크린을 로딩하도록 정의할 수 있으나 게임 플레이가 다시 시작되면 반드시 해제된 후 재사용 되어야 한다.

2.2.1.1 배경 지식

대부분의 개발자들은 일반적인 케이스에서 Scaleform 에 의해 사용되고 있는 메모리를 재사용하기 위해 메모리 아레나를 정의할 필요가 없을 것이다. Scaleform 가 메모리를 할당할 때 GFx::System 에서 정의한 SysAlloc 객체로부터 얻게 되고 이 메모리는 데이터를 언로딩 하면서 해제하고 Scaleform 객체를 소멸할 것이다. 하지만 몇 가지의 이유에서 해제된 메모리 패턴은 할당되었던 것과 항상 일치하지 않는다. 예를 들면, 만약 CreateMovie 를 호출하여 사용한 후 해제한다면 Create 콜이 해제될

때 대부분의 메모리 블록들이 할당되면서 몇 개는 조금 더 오랜 시간 동안 잡혀있는 경우가 발생할 수도 있다. 이런 경우는 단편화, 동적 데이터 구조, 멀티 스레딩, 공유 및 캐싱하고 있는 Scaleform 리소스를 포함한 몇 가지의 이유로 발생할 수 있다. 대부분 몇 개의 메모리 블록들이 다른 블록보다 잠시 동안 해제되지 않는 문제는 이 블록들이 계속 쌓이지 않고 Scaleform 수명 동안 재사용이 되기 때문에 큰 문제를 발생시키지는 않는다. 하지만 이렇게 예상할 수 없는 할당은 어플리케이션이 고정된 크기의 메모리 버퍼를 가지고 있고 Scaleform 와 다른 시스템들이 데이터를 공유해야 하는 경우에 문제가 될 수 있다. SDK 의 초기 버전들에서는 개발자들이 버퍼 메모리를 완전히 해제시키기 위해 Scaleform 를 완전히 정지시켜야 했다. 하지만 Scaleform 3.1 에서는 사용자가 직접 이러한 버퍼용 메모리 아레나를 정의하고 특정 메모리 아레나에 지정된 SWF/GFX 파일을 로딩하게 할 수 있다. 이 파일들이 언로딩이 되면 Scaleform::Memory::DestroyArena 를 호출하여 안전하게 모든 아레나 메모리를 재사용할 수 있다.

2.2.1.2 메모리 아레나의 사용

메모리 아레나를 사용하기 위해서 개발자들은 다음의 단계를 따라야 한다.

1. Scaleform::Memory::CreateArena 를 호출하여 0(0 이란 항상 존재하는 전역 l 혹은 디폴트 아레나를 뜻한다)이 아닌 정수 식별자를 부여하여 아레나를 생성한다. 이 메모리에 접근하려면 SysAlloc 인터페이스가 필요한데 고정 크기의 메모리 버퍼의 경우 SysAllocStatic 을 사용할 수 있다.

   ```
   // Assume pbuf1 points to a buffer of 10,000,000 bytes.
    SysAllocStatic  sysAlloc(pbuf1, 10000000);
    Memory::CreateArena(1, &sysAlloc);
   ```

2. 사용할 아레나 ID 를 정의하는 동안 GFx::Loader::CreateMovie/GFx::MovieDef::CreateInstance 를 호출한다. 이러한 movie 객체를 위해 필요한 힙은 특정 아레나에 생성되어 필요한 대부분의 메모리가 여기서부터 올 수 있도록 한다. 여기서 주의할 것은 특정 "공유" 전역 메모리는 지속적으로 할당될 수 있으므로 전역적으로 충분한 비축 메모리를 만들어놔야 한다.

   ```
    // use arena 1 for the movie
    Ptr<MovieDef> pMovieDef =
   ```

```
            *Loader.CreateMovie(filenameStr, Loader::LoadWaitFrame1, 1);
...
// use arena 1 for the instance
Ptr<Movie> pMovie =
        *pnewMovieDef->CreateInstance(MovieDef::MemoryParams(1), false);
```

3. 필요한 기간만큼 GFx::MovieDef/GFx::Movie 객체의 결과를 사용한다.
4. 아레나 내부의 생성된 무비들과 객체들을 모두 해제한다. GFx::MovieDef::WaitForLoadFinish 는 반드시 해제 이전에 호출되어야 한다. 만약 GFx::ThreadedTaskManager 가 백그라운드 로딩 스레드에 사용되었다면, 백그라운드 스레드는 강제로 로딩이 끝나며 그것들의 무비 참조들도 해제된다.

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

또는 (만약 Scaleform::Ptr 가 사용되지 않았다면):

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

5. DestroyArena 를 호출한다. 이 작업 후, 아래나의 모든 메모리는 CreateArena 가 다시 호출될 때까지 안전하게 재 사용될 수 있다. 그 Scaleform::Memory::ArenaIsEmpty 함수는 메모리 아레나가 비어있는지, 또는 소멸될 준비가 되었는지 확인할 때 사용할 수 있다.

```
// DestroyArena will ASSERT if memory was not properly released before
// the call. In Release it will crash at "return *(int*)0;".
Memory::DestroyArena(1);
```

SysAllocobject 는 'pbuff1' 포인터의 재사용이 준비 된 이후에 범위(scope)를 벗어나거나 소멸될 수 있다. 메모리 아레나는 필요할 때 재 생성 될 수 있다.

2.3 *OS 로 할당 위임하기*

할당을 오버라이딩하는 대신 Scaleform 에서 제공하는 OS 직접 할당자를 사용하는 방법을 택할 수 있습니다. 여기에는 다음이 포함됩니다.

- *Scaleform::SysAllocWinAPI* –VirtualAlloc()/VirtualFree() API 를 사용함. MS 플랫폼에서는 메모리 정렬이나 페이징 감소 등의 이유로 최선의 선택임.

- *Scaleform::SysAllocPS3* –PS3 에서 sys_memory_allocate/sys_memory_free 를 사용해서 효율적인 페이지 관리가능.

시스템 할당자를 사용함으로써 얻어지는 주요 이점은 하드웨어 페이징이다. 페이징이란 OS 가 물리적 메모리를 페이지 크기에 맞춰서 가상 주소 공간에 재맵핑하는 것을 말한다. 할당 시스템이 충분히 똘똘하다면 커다란 메모리 블록에서 단편화를 줄일 수 있는데, 주소 공간 여기저기에 흩어져있는 작은 메모리 블록을 연속적인 메모리 형태로 만드는 것이다.

Scaleform 의 WinAPI 및 PS3 의 SysAlloc 은 64K 또는 그보다 더 작은 블록상에서 맵핑과 언매핑에 의한 시스템 메모리를 사용할 수 있다는 이점이 있다. 이러한 입상은, 예를 들어 dlmalloc 에 의해 사용된 2MB 블록보다 훨씬 효율적입니다.

2.3.1 SysAllocPaged 구현하기

Scaleform 3.3 에서 업데이트된 SysAlloc 인터페이스를 도입함에 따라, 구 SysAlloc 구현은 SysAllocPaged 으로 명칭이 변경되었습니다. SysAllocPaged 을 사용하는 할당자 구현은 여전히 Scaleform 에서 사용 가능하며, SysAllocStatic 과 OS 특정 할당자 실행을 위해 사용됩니다. 이들은 라이브러리에 포함되어 있으나, 사용하지 않으면 연결되지 않습니다. 아래에 기술한 것처럼, 호환성을 목적으로 SysAlloc 대신 SysAllocPaged 이 구현될 수 있습니다.

살짝 변경된 SysAllocPagedMalloc 할당자 코드예시를 보면 다음과 같다.

```
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;
    }
};
```

보이는 것처럼 할당자 구현은 매우 간단한다. MySysAlloc 클래스는 SysAllocPaged 를 기반으로 상속받아서 3 개의 가상함수를 구현한다(`GetInfo, Alloc, Free`). 최적화를 하려면 ReallocInPlace 도 구현할 수 있지만, 여기서는 생략하였다.

- `GetInfo()` –Scaleform::SysAllocPaged::Info 구조체에 값을 넣어서 할당자의 정렬 지원이나 크기를 반환한다.
- `Alloc` –지정된 크기의 메모리를 할당한다. 이 함수는 크기와 정렬에 관련된 전달인자가 필요하다. 넘겨진 정렬값은 GetInfo 의 MaxAlign 보다 절대로 클 수 없다. MaxAlign 값을 1 로 설정했으므로 두번째 전달인자는 무시할 수 있다.
- `Free` –Alloc 으로 할당한 메모리를 해제한다. free 함수는 추가적으로 최초 크기와 정렬 전달인자를 받는다. 여기서는 필요없으므로 무시했다.
- `ReallocInPlace` –위치를 이동시키지 않고 다시 메모리 재할당을 시도한다. 불가능하면 false 를 반환한다. 대부분의 사용자는 이 함수를 오버라이드할 필요가 없을 것이다. 상세한 내용은 Scaleform Reference Documentation 을 참고하기 바란다.

보면 알겠지만 `Alloc` 과 `Free` 함수는 표준과 거의 똑같다. 따라서 관심이 가는 함수는 오직 `GetInfo` 뿐일 것이다. 이 함수는 당신의 할당자가 어떤 능력이 있는지를 Scaleform 에 알려준다. 이렇게 해서 SysAllocPaged 구현에서 정렬에 관련된 부분에 영향을 미친다. 또한, Scaleform 가 메모리를 좀더 효율적으로 사용할 수 있게 해준다. `SysAllocPaged::Info` 구조체에는 4 가지 값이 있다.

- `MinAlign` –할당자가 모든 할당에 적용할 최소 정렬

- `MaxAlign` –할당자가 지원하는 최대 정렬. 이 값이 0 이면 Scaleform 는 모든 정렬이 지원된다고 가정함. 이 값이 1 일 경우에는 정렬이 지원되지 않는 다는 것이므로 Alloc 구현시에 정렬 전달인자를 무시해도 됨.
- `Granularity` –Scaleform 할당시에 추천하는 크기. Scaleform 는 적어도 이 크기로 할당을 요청함. 사용자의 할당자가 (앞서의 malloc 처럼) 정렬에 친화적이 아니라면 적어도 64K 정도를 추천함
- `HasRealloc` –`ReallocInPlace` 를 지원할 것인지에 대한 플래그. 우리가 구현한 경우는 false
- `SysDirectThreshold` - 는 null 이 아닐 때 전체 크기의 임계 값을 정의한다. 만약 할당된 크기가 SysDirectThreshold 보다 크거나 같다면, 시스템이 리디렉션 되고 세분화 계층은 무시된다.
- `MaxHeapGranularity` - 가 만약 null이 아니라면, MaxHeapGranularity 는 최대 가능한 세분성을 제한한다. 대부분의 경우, 자주 사용되는 할당연산의 alloc/free 연산의 비용을 절감하기 위해 MaxHeapGranularity 는 시스템 메모리 공간을 줄일 수 있다. MaxHeapGranularity 는 최소 4096 이거나, 4096의 배수이어야 한다.

`MaxAlign` 설명에서 보이는 것처럼 정렬을 지원하지 않는 SysAlloc 인터페이스를 구현하는 것은 쉽다. 하지만 정렬을 지원한다면 그 만큼 혜택이 있다. 실제로 다음과 같은 3 가지 시나리오를 고려해볼 필요가 있다.

1. 그 할당자의 구현이 정렬을 지원하지 않거나 비효율적으로 관리하도록 한다. 이러한 경우 `MaxAlign` 값을 1 로 설정하고 Scaleform 가 내부적으로 필요한 정렬 처리를 하도록 한다.
2. 그 할당자가 정렬을 효율적으로 관리하게 한다. 이러한 경우 `MaxAlign` 을 0 으로 설정하거나 사용자가 지원하는 가장 큰 정렬 크기를 설정한다. 그리고 `Alloc` 함수를 적절히 구현한다.
3. 그 할장자가 시스템 인터페이스이며 항상 강제적으로 정렬하며, 페이지 크기가 정해져 있다. 이를 효율적으로 다루기 위해서는 `MinAlign` 및 `MaxAlign` 을 시스템 페이지 크기로 설정하고 모든 할당 크기를 OS 에 넘긴다.

일단 사용자의 할당자의 인스턴스가 생성되었으면 Scaleform 의 초기화 과정에서 GFx::System 생성자에 전달 될 수 있다.

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

2.4 *메모리 힙*

언급한 것처럼 Scaleform 는 힙이라 불리는 메모리풀 내부에서 메모리를 유지합니다. 모든 힙은 mesh cache 등의 특정한 파일 하위 시스템 데이터에서 불러온 데이터를 보유하는 등의 개별적인 목적을 위해서 생성됩니다. 이 섹션에서는 Scaleform 코어에서의 힙 메모리 관리를 위해 사용되는 API 를 간략히 서술합니다.

2.4.1 힙 API

최종 사용자가 사용하기 쉽도록 대부분의 메모리 힙 관리자는 내부코드에 숨겨져 있다. 개발자는 모든 외부 Scaleform 객체에 대해서 어떤 힙 할당이 발생할지 고문하지 않고 'new'연산자를 사용할 수 있다.

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

이러한 경우'new'연산자는 힙을 지정하지 않고 사용되었고, 메모리 할당은 Scaleform 전역 힙 내부에서 발생할 것이다. 설정 객체의 개수가 적기 때문에 이러한 접근은훌륭하다. 전역(global) 힙은 GFx::System 초기화 중에 생성되며 종료될 때까지 활성 상태를 유지하며, 항상 메모리 영역 0 을 사용합니다.

추가 메모리 힙은 내부적인 단편화를 제어하고 시스템 당 메모리 사용을 추적한다. 예를 들어서, GFx::MeshCacheManager 클래스는 내부 힙을 생성해서 모든 shape 데이터의 삼각화 된 데이터를 가지고 있게된다. `GFx::Loader::CreateMovie()`는 특정 SWF/GFX 파일에서 로드된 데이터를 보관할 힙을 생성한다. `GFx::MovieDef::CreateInstance` 는 타임라인과 액션스크립트 실행상태를 보관할

힙을 생성한다. 이러한 모든 상세 사항들은 씬(scene)의 뒤에서 관리되므로 적어도 Scaleform 코드를 디버그 하거나 수정, 확장하려는 것이 아니라면 대부분의 사용자들은 신경 쓸 필요가 없다.

이번 장의 나머지 부분은 메모리 힙 시스템을 자세하기 위해서 몇 개의 클래스와 매크로에 대하여 언급할 것이다. 이 내용들은 Scaleform 를 사용하는데 있어서 크리티컬하지는 않기 때문에 시간이 없는 독자라면 넘어가도 좋다.

메모리 힙은 전용 Scaleform 하위 시스템 혹은 데이터 셋를 위해 생성된 인스턴스와 함께 Scaleform::MemoryHeap 클래스에 의해 나타내어진다. `Scaleform::MemoryHeap::CreateHeap` 을 사용하면 글로벌 힙에서 자식 힙을 생성할 수 있으며 `Scaleform::MemoryHeap::Release` 를 사용해서 삭제할 수 있다. 힙이 살아있는 동안에는 `Scaleform::MemoryHeap::Alloc` 을 사용해서 힙에서 메모리를 할당할 수 있으며 `Scaleform::MemoryHeap::Free` 를 사용해서 해제할 수 있다. 힙이 삭제되었을 때는 모든 내부 메모리가 해제된다. 큰 크기의 힙은 SysAlloc 인터페이스를 통해서 즉시 삭제된다.

메모리가 정확한 힙에 할당되도록 하기 위해서 Scaleform 대부분의 할당은 Scaleform::MemoryHeap 포인터에서 필요한 특별한 매크를 사용한다.

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

디버그 빌드에서 이러한 매크로를 사용하면 적절한 행번호와 파일명이 프로그램에게 넘겨진다. 'id'는 Scaleform AMP 메모리 통계시스템에서 그룹화하여 추적할 수 있는 할당용 태그로 사용된다.

주어진 메모리 매크로를 사용해서 GFx::SpriteDef 오브젝트를 pHeap 이라는 커스텀 힙에 생성할 수 있다.

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

여기서 GFx::SpriteDef 은 Scaleform 에서 사용되는 내부 클래스다. 대부분의 Scaleform 오브젝트는 참조카운트를 가지고 있기 때문에 Ptr<>같은 스마트 포인터가 사용되며 스코프를 벗어날 때 자동적으로 내부 해제함수에 의해서 해제된다. Scaleform::NewOverrideBase 처럼 참조카운트가 없은 객채에 대해서는 'delete'연산자를 직접 사용해야 한다. delete 연산자와 SF_FREE(address) 매크로는 자동으로 힙을 구분한다.

2.4.2 자동 힙

힙 할당 매크로가 모든 필요한 힙관련 기능을 제공하고는 있지만, 항상 힙 포인터를 넘기는 것이 귀찮을 때가 있다. 이러한 포인터 전달은 Scaleform::String/Scaleform::Array<>같은 메모리 할당이 필요한 컨테이너 객체에 적용될 때 특히 짜증난다. 다음 코드를 보고, 이 클래스가 선언되어 힙에 인스턴스가 생성된다고 생각해보자.

```
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 가 지정된 힙에서 생성되고는 있지만, 동적 할당 되어야하는 하나의 문자열과 배열 객체도 포함하고 있다. 특별한 작업을 하지 않으면, 의도치않게 이 객체들이 전역 힙에서 할당될 것이다. 이 문제를 해결하려면 이론상 힙 포인터가 GFx::Sprite 객체 생성자로 전달된 다음, 문자열과 배열 생성자에게도 전달되어야 한다. 이러한 인자를 전달하는 행위는 프로그램을 복잡하게 하며, 배열,

문자열, 힙포인터를 유지하기 위한 엄청난 추가메모리와 같이 간단하지만 많은 객체들이 필요하게 된다. 이러한 상황을 쉽게 해결하기 위해서 Scaleform 코어는 특별히 "자동힙" 할당 매크로를 사용한다.

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

이 매크로는 SF_HEAP_ALLOC 과 SF_HEAP_NEW 형제들과 닮아있다. 이들은 한가지만 차이가 있는데, 이들은 힙 포인터가 아니라 메모리 위치 포인터를 받는다는 것이다. 일단 호출이 되면, 이들 매크로는 주어진 메모리 주소에 따라서 힙을 파악하고, 동일한 힙에 메모리를 할당한다. 다음 예제를 보자.

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

이 예제에서 특정 힙에서 메모리 버퍼를 생성하고, GFx::Sprite 객채가 동일한 힙에서 생성되었다. 이 예제의 특이한 점은 'pbuffer' 포인터를 SF_HEAP_AUTO_NEW 에 단순히 전달한 것이 아니라, 버퍼 할당 공간 범위내의 주소를 전달 했다는 것이다. 이 코드는 실수로 이렇게 만들어진 것이 아니라, Scaleform 메모리 힙 시스템의 새로운 기능을 보여주는 것이다. 이 새로운 기능이란 메모리가 할당된 공간의 어떤 주소에 근거해서도 매우 효율적으로 메모리 힙을 판별해 내는 능력을 의미한다. 이러한 독특한 기능은 이번 장의 서두에서 언급했던 힙 전달 문제에 대한 좋은 해결 방법에 대한 열쇠가 된다. 자동 힙 판별를 사용한다면, 프로그래머는 위치에 따라 정확한 힙의 위치에 자동으로 메모리를 할당하는 컨테이너를 생성할 수 있을 것이다. 다시 말해서, 앞서 설명한 GFx::Sprite 를 다음과 같이 재코딩 할 수 있다.

```
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();
```

여기서 우리는 Scaleform::String 과 Scaleform::Array<>클래스를 "지역 힙(Local Heap)"과 동일한 Scaleform::StringLH 과 Scaleform::ArrayLH<>로 바꿨다. 이러한 지역 힙 컨테이너는 메모리 할당을 앞서 설명한 "자동 힙" 테크닉을 사용해서 객체힙과 동일한 힙에서 할당한다. 이때 추가적인 인자전달과 같은 오버헤드도 없다. 이것과 비슷한 컨테이너들이 Scaleform 코어 전체에 걸쳐서 사용되고 있다는 것을 알게 되었을 것이다.

2.4.3 힙 추적기(Heap Tracer)

GFxConfig.h 에 정의된 SF_MEMORY_TRACE_ALL 을 활성화하여 핵심 메모리 할당을 추적할 수 있으며 Scaleform::Memory 클래스의 SetTracer 메서드를 사용하면 CreateHeap, DestroyHeap, Alloc, Free 등과 같은 메모리 연산을 추적할 수 있습니다.

다음 예제 코드는 SysAllocWinAPI 를 사용하는 경우 추적기를 사용하는 방법을 보여줍니다.

```
#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 메모리 보고

상세 메모리 보고는 원래 Scaleform Player AMP HUD 의 일부분으로 Scaleform 3.0 에 도입되었으나, 이번에 프로파일링 기능이 PC 와 콘솔 Scaleform 응용 프로그램 간 원격 연결이 가능한 단일 실행형 AMP 응용 프로그램으로 이동되었습니다. AMP 의 사용에 대한 상세 정보는 AMP 사용자 설명서를 참조하십시오.

Scaleform Player 경량화를 위해 HUD UI 가 제거되었지만, (Ctrl + F5) 키를 누르면 메모리 보고가 생성되어 콘솔로 전달됩니다. 이 보고는 다음의 Scaleform::MemoryHeap::MemReport 함수에서 생성 가능한 형식 중 하나입니다.

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

MemReport 는 메모리 보고를 생성하며, 이러한 보고는 제공된 스트링 버퍼에 XML 형식으로 기록됩니다. 이러한 스트링은 디버깅 목적으로 출력 콘솔 또는 화면에 표시될 수 있습니다. 다음 중 하나가 MemReportType 전달 인자가 됩니다.

- MemoryHeap::MemReportBrief – Scaleform 시스템의 총 메모리 사용에 대한 요약입니다. 예는 다음과 같습니다.

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

- MemoryHeap::MemReportFull – Ctrl+F6 입력시 Scaleform Player 에서 출력하는 메모리 형식입니다. 여기에는 시스템 메모리 요약 및 개별 힙 메모리 요약이 포함되며 SF_MEMORY_ENABLE_DEBUG_INFO 가 정의된 경우 stat ID 별로 할당된 memory breakdown 도 포함됩니다. 힙별로 제공된 정보 예는 다음과 같습니다.

```
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 – 파일에서 사용한 메모리를 생성하는 메모리 형식입니다. 각각의 SWF 에 대한 MovieData, MovieDef 및 모든 MovieViews 의 stat ID 별 값의 합계를 보고합니다. 예는 다음과 같습니다.

```
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 가 메모리 탭에서 표시되는 메모리 형식이며 가장 자세한 보고로써, 각 힙에서 사용한 메모리 크기, 디버그 정보에 대해 사용된 크기 및 Scaleform 가 점유했지만 현재 사용하지 않는 크기를 표시합니다. 예제:

```
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
```

앞서 언급한 것처럼 Scaleform 가 SF_MEMORY_ENABLE_DEBUG_INFO 를 통해 컴파일되면 메모리 시스템은 할당에 해당 할당의 목적을 기술한 "stat ID" 태그를 붙입니다.

- MovieDef – 불러온 파일 데이터를 나타냅니다. 이 메모리는 파일을 완전히 불러온 후에는 증가하지 않습니다.
- MovieView – GFx::Movie 인스턴스 데이터를 포함합니다. 이 카테고리의 크기는 ActionScript 가 많운 오브젝트를 할당 중일 때 또는 화면에 없은 다른 오브젝트의 movie clip 수가 많을 때 증가합니다.
- CharDefs – 개별적인 movie clip 및 기타 flash 오브젝트의 정의입니다.
- ShapeData – 복잡한 셰이프 및 폰트의 벡터 표현입니다.
- Tags – 애니메이션 keyframe 데이터입니다.
- ActionOps – ActionScript 바이트코드입니다.
- MeshCache – 벡터 tesselator 와 EdgeAA 에서 생성된 캐쉬된 셰이프 메쉬 데이터를 포함합니다. 새 셰이프가 한계치까지 바둑판 모양으로 변할수록 이 카테고리는 증가합니다.
- FontCache – 캐쉬된 폰트 데이터를 포함합니다.

4 가베지 콜렉션

Scaleform 2.0 이하에서는 액션스크립트 객체에 간단한 참조 카운팅을 사용했다. 대부분의 경우에는 충분했다. 하지만, 액스스크립트에서는 2 개 이상이 객체가 서로를 참조하는 소위 순환 참조(circular references)를 허용한다. 이것은 메모리 누수를 일으켜서 시스템 성능에 영향을 준다. 다음 코드는 둘중의 하나가 명시적으로 연결 해제되지 않으면 메모리 누수를 일으킨다.

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

이론적으로, 순환참조를 피하기 위해서 재작업 하거나 청소 함수를 두어서 참조 시에 상호 객체연결을 절단하도록 하는 것도 가능하나. 하지만 대부분의 경우에는 청소 함수가 제대로 작동하지 않고 오히려 상황을 악화시키게 되는데, 특히 액션스크립트의 클래스와 컴포넌트들이 사용되면 그렇다.

순환참조를 해결하기 위해서 Scaleform 3.0 은 고수준으로 최적화된 참조 카운팅 기반 해제 메커니즘의 가베지 컬랙션을 제공한다. 순환 참조를 만들지 않는다면, 이 메커니즘은 일반적인 참조 카운팅 시스템처럼 동작 한다. 그러나 순환 참조가 발생하면 콜랙터가 참조되지 않은 객체과 순환 참조들을 해제한다. 대부분의 플래시 파일들에서 이것은 매우 적은 실행효율 저하를 나타낸다.

위에서 언급했던 바와 같이 Scaleform 3.0 의 가베지 컬랙션은 기본적으로 활성화 되어있기 때문에 순환참조로 인해 발생된 모든 메모리 누수는 이 가베지 컬랙션 의해 제거 된다. 만약 기 기능이 불필요한 경우에 이러한 기능을 꺼놓을 수도 있다. 단, 이 기능은 Scaleform 를 리빌드 해야하기 때문에 소크 코드를 가지고 있는 개발자만이 가능하다.

가베지 콜렉션을 비활성화하기 위해서는 Include/GFxConfig.h 를 열고 디폴트로는 주석 처리가 되어 있지 않은 GFX_AS_ENABLE_GC 매크로 라인을 주석 처리한다.

```
// Enable garbage collection
#define GFX_AS_ENABLE_GC
```

매크로의 주석을 제고 했으면 Scaleform 라이브러리를 완전히 새로 재빌드 해야한다.

4.1 *가베지 컬렉션과 함께 사용되는 메모리 힙의 설정*

Scaleform 3.0 은 가베지 컬렉터에와 연동되는 메모리 힙의 설정을 따른다. Scaleform 3.3 및 상위 버전에서는 힙 공유를 허용하며, 따라서 복수의 MovieView 간에 가비지 콜렉터 공유를 허용합니다.

다음 함수를 사용하면 주어진 MovieDef 에 맞는 새 힙을 생성할 수 있습니다.

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

첫번째 매개변수인 `MemoryParams` 구조체의 첫번째 인자는 다음의 구조체로 정의된다.

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

- `Desc` – 무비를 위해 생성되는 힙의 식별자. 힙의 정렬(`MinAlign`), 모양(`Granularity`), 제한(`Limit`), 플래그(`Flags`)를 명시한다. 만약 힙의 제한(`Limit`)을 지정한 경우, Scaleform는 힙의 사용량을 지속적으로 제한 값 보다 낮게 유지하려 할 것이다.
- `HeapLimitMultiplier` – 부동소수점 힙 제한 연산자(0~1). 이 값은 힙이 한도를 넘었을 때 어떻게 제한할 것인지 결정하는데 사용된다. 아래의 세부사항을 참조하라.
- `MaxCollectionRoots` – 가베지 컬렉터의 이 실행되기 이전의 roots의 수, 이 값은 root의 최대 초기 값이다. 이것은 실행 중에 늘릴 수 있다. 아래의 세부사항을 참조하라.
- `FramesBetweenCollections` – 최대 roots 값에 도달하지 않았더라도 컬렉션이 강제된 이후의 프래임의 수이다. 현재의 최대 root 값이 높을 때 즉각적으로 컬렉션을 수행하는데 유용하며, 이러한 즉각적인 컬렉션의 수행은 비용이 적다. 자세한 내용은 다음을 보라.

마찬가지로 movie view 힙은 두 가지 단계로 생성됩니다.

```
MemoryContext* MovieDef::CreateMemoryContext(
                                        const char* heapName,
                                        const MemoryParams& memParams,
```

```
                                        bool debugHeap)

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

MemoryContext 오브젝트는 movie view 는 물론 가비지 콜렉터와 같은 다른 힙 특정 오브젝트를 캡슐화합니다. movie view 및 메모리 컨텍스트를 통해 movie view 의 힙을 생성하는 두 번째 접근은, 힙의 쓰레드-안전(thread-safe) 여부 및 힙이 디버그로 표시되어 AMP 보고에서 제외되는지와 상관없이 AMP 에 의해 표시되는 힙 이름을 지정하는 데 훨씬 융통성이 있습니다. 더욱 중요한 것은 같은 쓰레드 상에 복수의 movie view 를 허용하여 단일 힙, 가비지 콜렉터, 문자열 관리자 및 텍스트 할당자를 공유함으로써 오버헤드를 줄이는 것입니다.

`MemoryParams::Desc` 는 무비 인스턴스 할당을 정의하는 메모리 힙의 일반적인 속성들을 정의하는데 사용된다.(예:ActionScript 할당) 두개의 인자로(`Desc.Limit`, `HeapLimitMultiplier`) 힙의 사용량을 제어하는 것이 가능하다. 그 힙은 128K 로 선 초기화 된다.(그래서 동적 제한이라고 불린다.) 이 제한 값을 넘을 때, 특별한 내부 핸들러가 호출된다. 이 핸들러는 공간에 대한 해제를 시도할 것인지, 힙을 확장할 것인지 결정할 수 있다. Boehm-Demers-Weiser(BDW) 가베지 컬렉터의 와 메모리 할당자로 해제할 것인지 확장할 것인지 판단하는데 사용되는 추론이다.
BDW 알고리즘은 다음과 같다.(의사 코드)

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

"collect" 단계에는 액션스크립트 가베지 컬렉터의 호출과 내부 캐시들을 해제하는 것과 같은 다른 메모리 해제 기능을 포함한다.

`HeapLimitMultiplier` 의 기본값은 0.25 이다. Scaleform는 마지막 메모리 컬렉션이 현재의 힙 사용량의 25%(HeapLimitMultiplier의 기본값)보다 커질 때 메모리를 해제 할 것이다. 그렇지 않으면 요청한 값보다 힙 사용량의 25%만큼 더 확장할 것이다.

만약 사용자가 `Desc.Limit`를 정의하면 위의 알고리즘은 정의된 제한과 동일하게 작동하도록 한다. 만약 힙이 `Desc.Limit` 값의 제한을 넘어선다면 컬렉션은 최종 컬렉션 이후의 할당들의 수에 관계 없이

호출될 것이다. 동적 힙 제한은 요청된 할당을 위해 컬렉션 + Delta 가 이루어진 후 힙 사용 공간에 설정될 것이다 (컬렉션이 할당에 필요한 메모리 보다 더 적은 메모리를 해제했을 경우에 한함).

두 번째 방법은 다음의 두 가지 가베지 컬렉터의 동작을 제어하는 것이다. `MaxCollectionRoots/ FramesBetweenCollections`. `MaxCollectionRoots` 는 가베지 컬렉터의 호출 때문에 초과되는 root의 수를 나타낸다. "root"는 잠재적으로 다른 액션스크립트 객체들과 순환 참조되는 형식의 액션스크립트 객체를 의미한다. 일반적으로 액션스크립트 객체는 액션스크립트로 참조(접근)되는 경우에 root 배열에 추가된다. (예: "obj.member=1"에서 접근되는 객체는 "obj"임.) MaxCollectionRoots에 정의된 값보다 참조되는 객체들의 수가 초과하면 가베지 컬렉터가 수행된다. 기본적으로 이 파라미터는 0으로 설정되어있고 이 기능을 사용하지 않도록 되어 있다.

`FramesBetweenCollections` 는 가베지 컬렉션이 강제로 수행된 이후의 프래임의 수를 나타낸다. "frame"이라는 용어는 하나의 플래시 프래임을 의미한다. 이 값은 가베지 컬렉터가 많은 수의 객체들이 수용될 때 급작스러운 성능 저하를 방지하는데 도움이 수도 있다. 예를들어 `FramesBetweenCollections` 가 1800 으로 설정되었다면 가베지 컬렉터가 30fps 플래시 프래임 비율인 60 초마다 수행될 것이다. `DescLimit` 값이 설정되지 않았을 경우, `FramesBetweenCollections` 인자는 과도한 액션스크립트 메모리 힙 증가를 막아준다. 기본적으로 이 인자는 0 으로 설정되며, 이 기능을 사용하지 않도록 설정되어 있다.

4.2 GFx::Movie:: ForceCollectGarbage

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

이 메소드는 프로그램에서 강제 가베지 콜렉터 실행을 하게 해준다. 만약 가베지 콜렉션이 꺼져있다면 아무 것도 하지 않는다.