Autodesk®
Scaleform®

Memory System Overview

This document explains how to configure, optimize, and manage memory in Scaleform 3.3 and higher.

Authors: Michael Antonov, Maxim Shemanarev, Alex Mantzaris
Version: 4.02
Last Edited: July 1, 2011

Autodesk®
GAMEWARE

Copyright Notice

How to Contact Autodesk Scaleform:

Document	Memory System Overview
Address	Autodesk Scaleform Corporation
	6305 Ivy Lane, Suite 310
	Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Scaleform 3.x Memory System

Autodesk® Scaleform® 3.0 transitioned memory allocation to a heap-based strategy and introduced detailed memory reporting for allocations and heaps. Memory reports are available programmatically or through the Analyzer for Memory and Performance (AMP) tool, which is shipped as a stand-alone application starting with Scaleform 3.2. With the Scaleform 3.3 release, the memory interface has been updated to address inefficiencies of the original heap implementation; these changes are covered in Section 1.2.

The rest of this document describes the Scaleform 3.3 and higher memory system, providing details on key areas that including:

- An overridable memory allocation interface.
- Heap-based memory management of Movies and GFx subsystems.
- ActionScript Garbage Collection.
- The available memory reporting functionality.

1.1   Key Memory Concepts

This section provides a high-level overview of the key Scaleform memory-management concepts, their interaction and effect on memory use. It is recommended that developers understand these concepts before customizing the allocator and/or profiling Scaleform memory use.

1.1.1 Overridable Allocator

All external memory allocations made by Scaleform go through a central interface installed on GFx::System during startup, allowing developers to plug in their own memory management system. Memory management can be customized by following several different approaches:

1. Developers can install their own Scaleform::SysAlloc interface, implementing its Alloc/Free/Realloc functions to delegate to memory allocator. This is most appropriate if the application uses a centralized allocator implementation, such as dlmalloc.
2. Scaleform can be given one or more large, fixed-sized memory blocks, pre-allocated at startup. Additional blocks may be reserved for temporary screens such as "Pause Menu" and fully released through the use of memory arenas.
3. Scaleform-provided system allocator implementation such as Scaleform::SysAllocWinAPI can be used to take advantage of hardware paging.

The approach chosen depends on the memory strategy adopted by the application. Case (2) is common on consoles with fixed memory budgets for sub-systems. The different implementations are described in detail in Section 2: Memory Allocation.

1.1.2 Memory Heaps

Regardless of the memory management approach developers choose externally, all internal Scaleform allocations are organized into heaps, broken down by file and by purpose. Developers will most likely first encounter these while looking at AMP or MemReport output. The most common heaps are:

- Global Heap – holds all shared allocations and configuration objects.
- MovieData Heap – stores read-only data loaded from a particular SWF/GFx file.
- MovieView Heap – represents a single GFx::Movie ActionScript sandbox, containing its timeline and instance data. This heap is subject to internal garbage collection.
- MeshCache – Tessellated shape triangle data are allocated here.

Memory heap implementation works by servicing allocations of the given file sub-system from dedicated memory blocks we refer to as "pages". This has a number of benefits:

- It reduces the number of external system allocations and thus improving performance.
- It eliminates thread-synchronization within a heap accessed by only one thread.
- It reduces external fragmentation by freeing related data as a unit.
- It enforces logical structure on memory reports.

However, heaps can also have a significant disadvantage – they are subject to internal fragmentation. Internal fragmentation occurs when small memory blocks freed within the heap don't get released to the rest of the application, because all of their associated pages are not free, resulting in effectively "unused" memory being held by the heap.

Scaleform 3.3 addressed this challenge by providing a new Scaleform::SysAlloc-based heap implementation that frees memory much more aggressively. Scaleform 3.3 and higher also allows GFx::Movie objects located on the same thread to share a single memory heap, which can result in less memory held by partially-filled blocks.

1.1.3 Garbage Collection

Scaleform 3.0 introduced ActionScript (AS) garbage collection, eliminating memory leaks caused by circular references within AS data structures. Proper collection is critical for operation of CLIK and any involved ActionScript program.

In Scaleform, garbage collection is contained within the GFx::Movie object, which serves as an execution sandbox for the AS Virtual Machine. With Scaleform 3.3 and higher, it is possible to share Movie heaps, which also shares and unifies the associated garbage collector. In most cases collection will be triggered automatically when the Movie heap grows due to ActionScript allocations; however, it is also possible to trigger it explicitly or instruct Scaleform to collect as explicit frame-based intervals. The details of collection and its configuration options are described in Section 4: Garbage Collection.

1.1.4 Memory Reporting and Debugging

The Scaleform memory system has the ability to mark allocations with "stat ID" tags that describe the purpose of those allocations. Allocated memory can then be reported by heap, broken down into stat ID categories, or by stat ID, broken down into heaps. This memory reporting functionality is exposed primarily by the Analyzer for Memory and Performance (AMP), which is shipped as a stand-alone profiling application starting with Scaleform 3.2. For details on using AMP, please refer to [AMP User Guide](#). Memory reports can also be obtained programmatically in string format by calling the Scaleform::MemoryHeap::MemReport function.

The stat ID tags described above are stored separately from the actual memory allocations, as debug data. Debug data are also kept for memory leak detection. When a debug version of Scaleform shuts down, a leaked memory report is generated in the Visual Studio output window or console. Since Scaleform doesn't have any known internal memory leaks, any leaks detected are most likely caused by improper use of reference counting on Scaleform objects. The extra associated debug data are not allocated in Shipping configurations.

1.2   Scaleform 3.3 Memory API Changes

As was mentioned earlier, Scaleform 3.3 includes two significant updates to improve memory use:

- GFx::Movie memory context sharing allows independent movie view instances to share heaps and garbage collection, improving memory reuse. Internal string tables are also shared, resulting in additional savings. Details of this sharing are covered in Section 4.
- The SysAlloc interface has been updated to be more "malloc-friendly", removing the 4K page-alignment requirement on external allocations and optimizing heaps to route all allocations larger than 512 bytes directly to SysAlloc. This approach results in more memory returned eagerly to the application, improving efficiency and reuse.

Note that the original Scaleform::SysAlloc implementation has now been renamed to Scaleform::SysAllocPaged, and is still available. Developers overriding SysAlloc can choose to either update their implementation, or change the base class to SysAllocPaged. Users relying on SysAllocStatic and memory arenas are not affected, as that class is still available.

2 Memory Allocation

As discussed in Section 1.1.1, Scaleform allows developers to customize memory allocation by choosing one of the three approaches:

1. Overriding the SysAlloc interface to delegate allocations to the application memory system,
2. Providing Scaleform with one or more fixed-size memory blocks, or
3. Instructing Scaleform to allocate memory directly from the operating system.

For best memory efficiency, it is important that developers select an approach that matches their application. The differences between approaches (1) and (2), which are the most commonly used, are discussed below.

When developers override SysAlloc, they delegate Scaleform allocations to their own memory system. When Scaleform needs memory for loading a new SWF file, for example, it requests it by calling the Scaleform::SysAlloc::Alloc function; it calls Scaleform::SysAlloc::Free when the content is unloaded. In this scenario, the best possible memory efficiency is achieved when all systems, including Scaleform, use a single shared global allocator, as any freed memory block becomes available for reuse by any system that may need it. Any compartmentalization of allocations reduces the overall sharing efficiency, thus increasing the total memory footprint. The details of overriding SysAlloc are provided in section 2.1 below.

The greatest problem with the global allocation approach is fragmentation, which is the reason why many console developers prefer to use a predetermined, fixed-memory layout instead. With predetermined memory layout, fixed-size memory regions are allocated for each system, with the size of each of these regions determined at startup, or during level loading. Overall, this approach trades the efficiency of a global system for a more predictable memory layout.

If an application uses this fixed-memory approach, developers should use a fixed-sized memory block allocator for Scaleform as well – a configuration that is described in Section 2.2. To make this scenario more practical, Scaleform also allows creation of secondary memory arenas, or regions of memory which are used for limited durations of time, such as when a "Pause Menu" is displayed.

2.1 Overriding Allocation

To replace the external Scaleform allocator, developers may follow these two steps:

1. Create a custom implementation of the SysAlloc interface, which defines the Alloc, Free, and Realloc methods.
2. Provide an instance of this allocator in the GFx::System constructor during Scaleform initialization.

A default Scaleform::SysAllocMalloc implementation, which relies on the standard malloc/free and their system-specific alternatives with alignment support, is provided by Scaleform and may be used directly, or as reference.

2.1.1 Implementing a custom SysAlloc

The simplest way to implement a custom memory heap is to copy the Scaleform SysAllocMalloc implementation, modifying it to make calls to another memory allocator. A slightly-modified Windows-specific SysAllocMalloc implementation is included below as reference:

```cpp
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);
    }
};
```

As can be seen, the implementation of the allocator is relatively simple. The MySysAlloc class derives from a base SysAlloc interface and implements three virtual functions: `Alloc`, `Free`, and `Realloc`. Although implementation of these functions must honor alignment, Scaleform will typically request only small alignments, such as 16 bytes or less. The oldSize and align arguments are passed into Free/Realloc interface to simplify implementation; they are helpful when implementing Realloc as a wrapper for a pair of Alloc/Free calls, for example.

Once developers have created an instance of their own allocator, it can be passed to the GFx::System constructor during Scaleform initialization:

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

In Scaleform 3.0, the GFx::System object needs to be created before any other Scaleform object and destroyed after all Scaleform objects are released. This is typically best done as part of the allocation initialization function, which calls code that uses Scaleform. However, it can also be part of another allocated object whose lifetime exceeds that of Scaleform. GFx::System should NOT be globally declared. If such use is not convenient, the GFx::System::Init() and GFxSystem::Destroy() static functions may be called instead, without instantiating an object. Similar to the GFx::System constructor, GFx::System::Init() takes a SysAlloc pointer argument.

2.2   Using Fixed Memory Blocks for Scaleform

As discussed in the introduction, console developers may choose to reserve one or more memory blocks up front, and pass them to Scaleform instead of overriding SysAlloc. This is done by instantiating SysAllocStatic as follows:

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

The above example passes a six megabyte chunk of memory to Scaleform to use for its allocations. Of course, the memory block cannot be reused or released until both the gfxSystem and blockAlloc objects go out of scope. It is possible, however, to add extra "releasable" memory blocks for temporary purposes, such as for displaying a pause menu. Such blocks, which are supported through the use of memory arenas, are discussed below.

When using a static allocator, developers need to be particularly mindful of the amount of memory used by Scaleform, making sure that no files are loaded, or movie instances created, that would exceed the specified memory amount. Once SysAllocStatic fails, it will return 0 from its Alloc implementation, causing Scaleform to either fail or crash. If needed, developers can use a simple SysAllocPaged wrapper object to detect this critical condition.

2.2.1 Memory Arenas

Scaleform 3.1 introduces support for Memory Arenas, which are user-specified allocator regions guaranteed to be fully released at specified points in program execution. Specifically, memory arenas define memory regions into which Scaleform files can be loaded and GFx::Movies created, such that these regions are fully released once their occupying Scaleform objects are destroyed. A memory arena can be destroyed without shutting down the rest of Scaleform, or unloading unrelated Scaleform files. Once an arena is destroyed, the application can reuse all of its memory for other non-Scaleform data. As one possible use case, a memory arena can be defined for loading a "Pause Menu" screen of a game, which requires memory temporarily, but must be released and reused once game-play resumes.

2.2.1.1 Background

In general, developers do not need to define memory arenas to reuse memory occupied by Scaleform. When Scaleform allocates memory it is obtained from the SysAlloc object specified on GFx::System initialization. This memory is released as data is unloaded and Scaleform objects are destroyed. For a number of reasons, however, the released memory pattern may not always match that of allocations. As an example, if CreateMovie is called, the movie is used, and then released, it is possible that while most memory blocks allocated during the Create call are released, a few of them are held for a longer period of time. This may happen for a number of reasons including fragmentation, dynamic data structures, multi-threading, Scaleform resource sharing, and caching.

In most cases, having a few temporarily unreleased memory blocks is not a problem, since such blocks do not accumulate, and are reused during the lifetime of Scaleform. However, the unpredictable nature of allocations may pose a problem if an application reserves fixed-sized memory buffers that it needs to share with both Scaleform and other systems. In earlier versions of the SDK, developers would have had to shut down Scaleform to ensure that buffer memory was completely released. With Scaleform 3.1, memory arenas for such buffers may be defined, and SWF/GFX files may be designated for loading into specific memory arenas. Once these files are unloaded, Scaleform::Memory::DestroyArena may be called, and all of the arena memory may be safely reused.

2.2.1.2 Using Memory Arenas

To use memory arenas, developers need to follow several steps:

1. Create an arena by calling Scaleform::Memory::CreateArena and give it a non-zero integer identifier (zero means global or default arena, which always exists). A SysAlloc interface is needed to access the memory. For a fixed-size memory buffer, SysAllocStatic may be used.

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

2. Call GFx::Loader::CreateMovie / GFx::MovieDef::CreateInstance, specifying the arena id to use. Heaps needed for those movie objects will be created within the specified arena, so most of the memory required will come from it. Note that some "shared" global memory may still be allocated, so developers need to make sure there is enough reserve available globally as well.

```
// use arena 1 for the movie
Ptr<GFx::MovieDef> pMovieDef =
          *Loader.CreateMovie(filenameStr, Loader::LoadWaitFrame1, 1);
...

// use arena 1 for the instance
Ptr<GFx::Movie> pMovie =
          *pnewMovieDef->CreateInstance(MemoryParams(1), false);
```

3. Use the resulting GFx::MovieDef/GFx::Movie objects as long as needed.

4. Release all of the created movies and objects allocated within an arena. GFx::MovieDef::WaitForLoadFinish should be called before Release if GFx::ThreadedTaskManager was used for threaded background loading, as it forces background threads to finish loading and to release their movie references.

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

or (if Scaleform::Ptr is not used):
```
pMovie->Release();
pMovieDef->Release();
```

5. Call DestroyArena. After this is done, all of the arena memory can be safely reused until CreateArena is called again. The function Scaleform::Memory::ArenaIsEmpty can be used to check if memory arena is empty and is ready to be destroyed.

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

The SysAlloc object may now go out of scope, or be destroyed, after which point 'pbuff1' is ready for reuse. Memory arenas may be recreated when necessary.

2.3 Delegating Allocation to the OS

Instead of overriding the allocator, developers can choose to use one of the OS-direct allocators provided with Scaleform. These include:

- *Scaleform::SysAllocWinAPI* – uses VirtualAlloc/VirtualFree APIs and is the best choice on Microsoft platforms due to good alignment and its ability to leverage paging; and
- *Scaleform::SysAllocPS3* – uses sys_memory_allocate/sys_memory_free calls for efficient page management and alignment on PS3.

The main benefit of using a system allocator comes from HW paging. Paging is the ability of the OS to remap physical memory to the virtual address space in blocks of page size. If used intelligently by an allocation system, paging can combat fragmentation on large memory blocks, by making linear memory ranges available out of smaller free memory blocks that are scattered throughout the address space.
The Scaleform WinAPI and PS3 SysAlloc implementations take advantage of this ability by mapping and un-mapping system memory in blocks of 64K, or smaller, granularity. This granularity is much more efficient than the 2MB blocks used by dlmalloc, for example.

2.3.1 Implementing SysAllocPaged

With the introduction of an updated SysAlloc interface in Scaleform 3.3, the original SysAlloc implementation was renamed to SysAllocPaged. An allocator implementation that makes use of it is still available in Scaleform, and is used for the SysAllocStatic and OS-specific allocator implementations. These are included in the library, but will not be linked if not used. For compatibility purposes, SysAllocPaged can be implemented instead of SysAlloc as described below.

A slightly modified SysAllocPagedMalloc allocator code is included below for reference:

```
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;
        }
    };
```

As can be seen, MySysAlloc class derives from a base SysAllocPaged interface and implements three virtual functions: `GetInfo`, `Alloc`, and `Free`. There is also a `ReallocInPlace` function that may be implemented as an optimization, but is omitted here.

- `GetInfo()` – Returns allocator alignment support capabilities and granularity by filling in the Scaleform::SysAllocPaged::Info structure members.

- `Alloc` – Allocates memory of specified size. The function takes a size and alignment arguments that need to be honored by the Alloc implementation. The passed align value will never be greater than MaxAlign returned from GetInfo. Since in the above case the MaxAlign value is set to 1 byte, the second argument may be safely ignored.

- `Free` – Free memory earlier allocated with Alloc. The Free function receives the extra size and align arguments that tell it the size of the original allocation. Since they are not needed for this free-based implementation, they can be safely ignored.

- `ReallocInPlace` – Attempt to reallocate memory size without moving it to a different location, returning false if that is not possible. Many users will not need to override this function; please refer to [Scaleform Reference Documentation](#) for details.

Since the behavior of `Alloc` and `Free` functions is fairly standard, the only function of interest is `GetInfo`. This function reports the capabilities of an allocator to Scaleform, so it can affect alignment support requirements imposed on a SysAllocPaged implementation, as well as Scaleform performance and memory use efficiency. The `SysAllocPaged::Info` structure contains six values:

- `MinAlign` – the minimum alignment that the allocator will apply to ALL allocations.
- `MaxAlign` – the maximum alignment that the allocator can support. If this value is set to 0, Scaleform will assume that any requested alignment is supported. If the reported value is 1, no alignment support is assumed, so the align argument can be ignored in the Alloc implementation.

11

- `Granularity` – the suggested granularity for Scaleform allocations. Scaleform will try to use at least this size for allocation requests. If the allocator is not alignment-friendly, as is the case with malloc, a value of at least 64K is recommended.
- `HasRealloc` – a boolean flag that specifies whether the implementation supports `ReallocInPlace`. In this case, false is returned.
- `SysDirectThreshold` - defines the global size threshold, when not null. If the allocation size is greater or equal to SysDirectThreshold, the request is redirected to the system, ignoring the granulator layer.
- `MaxHeapGranularity`- if not null, MaxHeapGranularity restricts the maximum possible heap granularity. In most cases, MaxHeapGranularity can reduce the system memory footprint for the price of more frequent segment alloc/free operations, which slow down the allocator. MaxHeapGranularity must be at least 4096, and a multiple of 4096.

As can be seen from the `MaxAlign` argument description, the SysAlloc interface is easy to implement for an allocator that doesn't support alignment, but can also take advantage of it, if it does. In practice, there are three possible scenarios worth considering:

1. *The allocator implementation doesn't support alignment*, or handles it inefficiently. If this is the case, simply set the `MaxAlign` value to 1 and let Scaleform do all of the needed alignment work internally.
2. *The allocator handles alignment efficiently.* If this is the case, set `MaxAlign` to 0, or the largest alignment size you can support, and implement the `Alloc` function to properly handle alignment.
3. *The allocator is an interface to the system,* with page size that is always enforced and aligned. To handle this efficiently, simply set both `MinAlign` and `MaxAlign` to the system page size, and pass all the allocation sizes to the OS.

Once an instance of the custom allocator has been created, it can be passed to the GFx::System constructor during Scaleform initialization:

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

2.4 Memory Heaps

As mentioned, Scaleform maintains memory within memory pools, called heaps. Every heap is created for a particular purpose, such as for holding data loaded from specific file subsystem data like the mesh cache. This section outlines the APIs used to manage heap memory within the Scaleform core.

2.4.1 Heap APIs

To make setup convenient for end users, most of the memory heap management details are hidden. Developers can use the 'new' operator directly on all external Scaleform objects, without having to consider the heap in which allocations will take place:

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

In this case, the operator 'new' is used without any heap specification, so the memory allocation will take place on the Scaleform global heap. Since the number of configuration objects is small, this approach works fine. The global heap is created during GFx::System initialization, and remains active until shutdown. It always uses memory arena 0.

Additional memory heaps are used internally to control fragmentation and track per-system memory use. For example, the GFx::MeshCacheManager class creates an internal heap to hold and constrain all of the tessellated shape data, the `GFx::Loader::CreateMovie()` function creates a heap to hold the data loaded from a particular SWF/GFX file, and the `GFx::MovieDef::CreateInstance` function creates a heap to hold the timeline and ActionScript runtime state. All these details are implemented behind the scenes and should not concern most users, at least until they plan to modify, extend, or debug Scaleform code.

The rest of this section describes the details of the memory heap system, and provides specific information on a number of the classes and macros involved. Knowledge of this material is not critical to using Scaleform.

Memory heaps are represented by the Scaleform::MemoryHeap class, with instances created for dedicated Scaleform sub-systems or data sets. Each child memory heap is created using `Scaleform::MemoryHeap::CreateHeap` on the global heap, and is destroyed by calling `Scaleform::MemoryHeap::Release`. During heap lifetime, memory can be allocated from the heap by calling `Scaleform::MemoryHeap::Alloc` function and can be released by calling `Sclaeform::MemoryHeap::Free`. When a heap is destroyed, all of its internal memory is released. For heaps of significant size, this memory will typically be immediately released to the application through the SysAlloc interface.

To ensure that memory is allocated from the right heap, most of the allocations in Scaleform use special macros that take a Scaleform:MemoryHeap pointer argument. These include:

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

The use of macros ensures that the appropriate line and filename information is passed for the allocation in debug builds. The 'id' is used to tag the purpose of the allocation, which is tracked and grouped in the Scaleform AMP memory statistics system.

Given the memory allocation macros, a new GFx::SpriteDef object may be created in a custom heap 'pHeap' with the following call:

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

Here, GFx::SpriteDef is an internal class used in Scaleform. Since most objects in Scaleform are reference counted, they are often held in smart pointers such as Ptr<>, and are freed when such pointers go out of scope by calling the internal Release function. For non-reference counted objects, such as the ones that derive from Scaleform::NewOverrideBase, 'delete' operator is used directly. It is not necessary to specify the heap to free memory, as the delete operator and SF_FREE(address) macro can automatically determine the heap.

2.4.2 Auto Heaps

Although the memory allocation macros provide all the necessary functionality for heap support, they can sometimes be cumbersome because they require a heap pointer to be passed throughout the program code. This pointer passing can become particularly painful when applied to aggregate container objects that perform their own memory allocation, such as Scaleform::String, or Scaleform::Array. Consider the following sample code that declares a class and creates an instance of it in the heap:

```
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();
```

Although the GFx::Sprite object is created in the specified heap, it contains a string and an array of objects that need to be allocated dynamically. If no special action is taken, these objects would be allocated on the global heap, which is not likely to be the intended scenario. To address this issue, a heap pointer could, in theory, be passed into the GFx::Sprite object constructor and then into the string and array constructors. This argument passing would make programming significantly more tedious, and would potentially require many simple objects, such as arrays and strings, to maintain pointers to the heaps, consuming extra memory. To simplify the programming of such situations, Scaleform core makes use of special "auto heap" allocation macros:

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

These macros behave much like their SF_HEAP_ALLOC and SF_HEAP_NEW siblings, but with one distinction: They take a pointer to a memory location instead of the pointer to a heap. When called, these macros automatically identify the heap based on the provided memory address, and make an allocation from the same heap. Consider this example:

```
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);
```

In this example a memory buffer is created from a specified heap, and a GFx::Sprite object is created in the same heap. The peculiar thing about this example is that it does not just pass the 'pbuffer' pointer into SF_HEAP_AUTO_NEW. Instead, it passed an address *within* the buffer allocation. This illustrates a novel feature of the Scaleform memory heap system, whereby it has its ability to efficiently identify a memory heap based on any address within a memory allocation. This unique feature is the key to an elegant solution to the heap-passing problem discussed in the beginning of this section. Armed with automatic heap identification, a programmer can create containers that automatically allocate memory from the correct heap based on their own location. In other words, the GFx::Sprite class presented earlier can be rewritten as follows:

```
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();
```

Here, the Scaleform::String and Scaleform::Array classes have be replaced by their "local heap" equivalents, Scaleform::StringLH and Scaleform::ArrayLH. These local heap containers allocate memory from the same heap as the object in which they are contained. This is accomplished by using the "auto heap" technique described above, without the extra argument passing overhead. These and similar containers are used throughout the Scaleform core, making sure that memory is allocated from the right heap.

2.4.3 Heap Tracer

Users can trace the essential memory allocations by enabling the SF_MEMORY_TRACE_ALL define in GFxConfig.h. Memory operations such as CreateHeap, DestroyHeap, Alloc, Free etc can be traced by calling the SetTracer method in the Scaleform::Memory class.

A sample code on how to use the tracer in case of SysAllocWinAPI is provided below:

```
    #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 Memory Reports

Although Scaleform 3.0 originally introduced detailed memory reporting as part of Scaleform Player AMP HUD, profiling functionality has now been moved into a stand-alone AMP application, which can remotely connect to both PC and console Scaleform applications. For details on using AMP, please refer to AMP User Guide.

The HUD UI has been removed from Scaleform Player to make it more lightweight, but a memory report may still be generated and sent to the console by pressing (Ctrl + F5) keys. This report is one of the possible formats generated by the Scaleform::MemoryHeap::MemReport function:

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

MemReport generates a memory report, possibly formatted in XML, written to a supplied string buffer. This string can then be written to the output console, the screen, or a file for debugging purposes. The MemReportType argument can be one of the following:

- MemoryHeap::MemReportBrief – A summary of the total memory consumed by the Scaleform system, as shown in the following example:

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

- MemoryHeap::MemReportFull – This memory type is what the Scaleform Player outputs when pressing (Ctrl-F6). It includes a system memory summary and individual heap memory summaries, and if SF_MEMORY_ENABLE_DEBUG_INFO is defined, it also includes a breakdown of the memory allocated, by stat ID. An example of the information given by heap is as follows:

```
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 – This memory type generates the memory consumed by file. The report sums the values per stat ID for the MovieData, MovieDef, and all MovieViews for each SWF. The format can be seen below:

```
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 – This memory type is what AMP displays in the memory tab, and is the most detailed report, showing how much memory is used by each heap, how much is used for debug information, and how much is claimed by Scaleform but is currently unused. For example:

```
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
```

20

```
            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
```

As mentioned above, when Scaleform is compiled with SF_MEMORY_ENABLE_DEBUG_INFO, the memory system marks allocations with "stat ID" tags that describe the purpose of those allocations. A brief description of some stat ID categories follows:

- MovieDef – represents loaded file data. This memory should not increase after the file is completely loaded.

- MovieView - contains GFx::Movie instance data. The size of this category will increase if ActionScript is allocating a lot of objects, or if there are a lot of movie clips of other objects alive on stage.

- CharDefs – definitions of individual movie clips and other flash objects.
- ShapeData – vector representation of complex shapes and fonts.
- Tags – animation keyframe data.
- ActionOps – ActionScript bytecode.
- MeshCache – contains the cached shape mesh data generated by vector tesselator and EdgeAA. This category grows as new shapes are tessellated, up to a limit.
- FontCache – contains cached font data.

4 Garbage Collection

Scaleform versions 2.2 and below used a simple reference counting mechanism for ActionScript objects. In most cases this is good enough. However, ActionScript allows you to create circular reference situations, where two or more objects have references to each other. This could result in memory leaks, affecting performance. Consider an example of code that will produce a leak unless one of the object references is explicitly disconnected:

Code:

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

Theoretically, it is possible to rework such code to avoid circular references, or to have a cleanup function that would disconnect objects in the reference cycle. In most situations, clean up functions do not work well, and the issue becomes even worse if ActionScript's classes and components are in use. Common cases that could result in memory leaks include the use of singletons, as well as the use of standard Flash UI Components.

To resolve the reference cycles, Scaleform 3.0 introduced garbage collection, a highly-optimized cleanup mechanism based on reference counting. If there are no circular references, this mechanism works as a regular reference counting system. However, in the case when circular references are created, the collector frees otherwise unreferenced objects with circular references. For most Flash files there will be a very little, if any, performance impact.

As mentioned above, memory leaks that were caused by circular references prior to Scaleform 3.0 will be eliminated with garbage collection, which is enabled by default. If not required, the garbage collection functionality may be disabled. However, only customers with access to the Scaleform source code can do this, since it requires rebuilding Scaleform.

To disable garbage collection, open the file *Include/GFxConfig.h* and comment the line with GFX_AS_ENABLE_GC macro, which is un-commented out by default.

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

After commenting the macro, it is necessary to perform a complete Scaleform library rebuild.

4.1 Configuring the memory heap to be used with garbage collection

Scaleform 3.0 and higher allows configuring of the memory heap that hosts the garbage collector. Scaleform 3.3 and higher allows heap sharing, and thus garbage collector sharing, among multiple MovieViews.

To create a new heap for a given MovieDef, the following function may be used:

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

The `MemoryParams` structure that is expected as the first argument is defined as a structure as follows:

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

- `Desc` – A descriptor for the heap that will be created for the movie. It is possible to specify heap alignment (`MinAlign`), granularity (`Granularity`), limit (`Limit`)and flags (`Flags`). If heap limit is specified, then Scaleform will try to maintain heap footprint at or below this limit.
- `HeapLimitMultiplier` – A floating point heap limit multiplier (0…1) which is used to determine how the heap limit grows if it is exceeded. See below for details.
- `MaxCollectionRoots` – Number of roots before garbage collection is executed. This is an initial value of max roots cap; it might be increased during the execution. See below for details.
- `FramesBetweenCollections` – Number of frames after which collection is forced, even if the max roots cap is not reached. It often beneficial to perform intermediate collections when the max roots cap is high, because it reduces the collection cost when that occurs. See below for details.

Equivalently, a movie view heap may be created in two steps:

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

```
Movie* MovieDef::CreateInstance(MemoryContext* memContext,
```

```
                        bool initFirstFrame = true)
```

The MemoryContext object encapsulates the movie view heap, as well as other heap-specific objects, such as the garbage collector. This second approach of creating the movie view and its heap through a memory context has the added flexibility of specifying the heap name displayed by AMP, whether the heap is to be thread-safe or not, and whether the heap is to be marked as debug and therefore excluded from AMP reports. More importantly, it allows multiple movie views on the same thread to share a single heap, garbage collector, string manager, and text allocator, thus reducing overhead.

As indicated above, `MemoryParams::Desc` is used to specify general properties of the memory heap that is used for the movie instance specific allocations (for example, ActionScript allocations). To control the heap footprint it is possible to set two parameters: `Desc.Limit` and `HeapLimitMultiplier`.

The heap has initial pre-set limit 128K (the so called "dynamic" limit). When this limit is exceeded, a special internal handler is called. This handler has logic to determine whether to try to free up space, or to expand the heap. The heuristic used to make this decision is taken from the Boehm-Demers-Weiser (BDW) garbage collector and memory allocator.

The BDW algorithm is as follows (pseudo-code):

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

The "collect" stage includes invocation of the ActionScript garbage collector plus some other actions to free memory, such as flushing internal caches.

The default value for `HeapLimitMultiplier` is 0.25. Thus, Scaleform will perform memory freeing only if footprint of allocated since the last memory collection is greater than 25% (the default value of HeapLimitMultiplier) of the current heap footprint. Otherwise, it will expand the limit up to the requested size plus 25% of the heap footprint.

If the user has specified `Desc.Limit`, then the above algorithm works the same way up to that specified limit. If the heap limit exceeds the `Desc.Limit` value, then collection is invoked regardless of the number of allocations since the last collection. The dynamic heap limit is set to the heap footprint after collection plus any extra memory that is required to fulfill the requested allocation.

The second way to control the garbage collector behavior is to specify the `MaxCollectionRoots/ FramesBetweenCollections` pair. `MaxCollectionRoots` specifies the number of roots that causes a garbage collector invocation when exceeded. A "root" is any ActionScript object that potentially may form

circular references with other ActionScript objects. In general, an ActionScript object is added to the roots array if it was referenced (touched) by ActionScript (for example, "obj.member = 1" touches the object "obj"). The garbage collector is invoked when the number of touched objects exceeds the value specified in `MaxCollectionRoots`. By default, this parameter is set to 0, indicating that this mechanism is disabled.

`FramesBetweenCollections` specifies the number of frames after which the garbage collection is forced to be invoked. The term "frame" here means a single Flash frame. This value may help to avoid performance spikes, which may occur if the garbage collector needs to go through a lot of objects. For example, `FramesBetweenCollections` may be set to 1800 and the garbage collector will be invoked every 60 seconds with a 30 fps Flash frame rate. The `FramesBetweenCollections` parameter may also be also used to avoid excessive ActionScript memory heap growth, in the case where `Desc.Limit` is not set. By default, this parameter is set to 0, indicating that this mechanism is disabled.

4.2   GFx::Movie::ForceCollectGarbage

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

This method can be used to force garbage collector execution by the application. It does nothing if garbage collection is off.