

## Memory System Overview

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

Authors: Michael Antonov, Maxim Shemanarev, Boris Rayskiy  
Version: 3. 01  
Last Edited: July 9, 2010

# Copyright Notice

## Autodesk® Scaleform® 3

© 2011 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/or other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, Algor, Alias, Alias (swirl design/logo), AliasStudio, Alias|Wavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backburner, Backdraft, Beast, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DXF, Ecotect, Exposure, Extending the Design Team, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, Freewheel, GDX Driver, Green Building Studio, Heads-up Design, Heidi, HumanIK, IDEA Server, i-drop, Illuminate Labs AB (design/logo), ImageModeler, iMOUT, Incinerator, Inferno, Inventor, Inventor LT, Kynapse, Kynogon, LandXplorer, LiquidLight, LiquidLight (design/logo), Lustre, MatchMover, Maya, Mechanical Desktop, Moldflow, Moldflow Plastics Advisers, MPI, Moldflow Plastics Insight, Moldflow Plastics Xpert, Moondust, MotionBuilder, Movimento, MPA, MPA (design/logo), MPX, MPX (design/logo), Mudbox, Multi-Master Editing, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProMaterials, RasterDWG, RealDWG, Real-time Roto, Recognize, Render Queue, Retimer, Reveal, Revit, RiverCAD, Robot, Scaleform, Scaleform AMP, Scaleform CLIK, Scaleform GFx, Scaleform IME, Scaleform Video, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Softimage|XSI (design/logo), Sparks, SteeringWheels, Stitcher, Stone, StormNET, StudioTools, ToolClip, Topobase, Toxik, TrustedDWG, U-Vis, ViewCube, Visual, Visual LISP, Volo, 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.

## How to Contact Autodesk Scaleform:

---

Document	Memory System Overview
Address	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	<a href="http://www.scaleform.com">www.scaleform.com</a>
Email	<a href="mailto:info@scaleform.com">info@scaleform.com</a>
Direct	(301) 446-3200
Fax	(301) 446-3199

# Table of Contents

<b>1</b>	<b>GFx 3.x Memory System</b>	<b>1</b>
1.1	Key Memory Concepts	1
1.1.1	Overridable Allocator	1
1.1.2	Memory Heaps	2
1.1.3	Garbage Collection	2
1.1.4	Memory Reporting and Debugging	3
1.2	GFx 3.3 Memory API Changes	3
<b>2</b>	<b>Memory Allocation</b>	<b>5</b>
2.1	Overriding Allocation	5
2.1.1	Implementing a custom GSysAlloc	6
2.2	Using Fixed Memory Blocks for GFx	7
2.2.1	Memory Arenas	8
2.3	Delegating Allocation to the OS	10
2.3.1	Implementing GSysAllocPaged	10
2.4	Memory Heaps	13
2.4.1	Heap APIs	13
2.4.2	Auto Heaps	14
<b>3</b>	<b>Memory Reports</b>	<b>17</b>
<b>4</b>	<b>Garbage Collection</b>	<b>22</b>
4.1	Configuring the memory heap to be used with garbage collection	23
4.2	GFxMovieView::ForceCollectGarbage	25

# 1 GfX 3.x Memory System

Scaleform® GfX™ 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 GfX 3.2. With the new GfX 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 GfX 3.3 memory system, providing details on key areas that including:

- An overridable memory allocation interface.
- Heap-based memory management of MovieViews 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 GfX 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 GfX memory use.

### 1.1.1 Overridable Allocator

All external memory allocations made by GfX go through a central interface installed on GfXSystem 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 GSysAlloc 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 dmalloc.
2. GfX 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 GSysAllocWinAPI 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 GfX 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 MovieView 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.

GfX 3.3 addresses this challenge by providing a new GSysAlloc-based heap implementation that frees memory much more aggressively. GfX 3.3 also allows MovieViews 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 GfX 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 GfX, garbage collection is contained within the GfXMovieView object, which serves as an execution sandbox for the AS Virtual Machine. With GfX 3.3 it is possible to share MovieView heaps, which also shares and unifies the associated garbage collector. In most cases collection will be triggered automatically when MovieView heap grows due to ActionScript allocations; however, it is also possible to trigger it explicitly or instruct GfX 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 GfX 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 GfX 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 GMemoryHeap::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 GfX shuts down, a leaked memory report is generated in the Visual Studio output window or console. Since GfX doesn’t have any known internal memory leaks, any leaks detected are most likely caused by improper use of reference counting on GfX objects. The extra associated debug data are not allocated in Shipping configurations.

## 1.2 GfX 3.3 Memory API Changes

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

- GfXMovieView 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 GSysAlloc 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 GSysAlloc. This approach results in more memory returned eagerly to the application, improving efficiency and reuse.

Note that the original GSysAlloc implementation has now been renamed to GSysAllocPaged, and is still available. Developers overriding GSysAlloc can choose to either update their implementation, or

change the base class to `GSysAllocPaged`. Users relying on `GSysAllocStatic` and memory arenas are not affected, as that class is still available.



## 2 Memory Allocation

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

1. Overriding the GSysAlloc interface to delegate allocations to the application memory system,
2. Providing GfX with one or more fixed-size memory blocks, or
3. Instructing GfX 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 GSysAlloc, they delegate GfX allocations to their own memory system. When GfX needs memory for loading a new SWF file, for example, it requests it by calling the GSysAlloc::Alloc function; it calls GSysAlloc::Free when the content is unloaded. In this scenario, the best possible memory efficiency is achieved when all systems, including GfX, 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 GSysAlloc 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 GfX as well – a configuration that is described in Section 2.2. To make this scenario more practical, GfX 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 GfX allocator, developers may follow these two steps:

1. Create a custom implementation of the GSysAlloc interface, which defines the Alloc, Free, and Realloc methods.
2. Provide an instance of this allocator in the GfXSystem constructor during GfX initialization.

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

### 2.1.1 Implementing a custom GSysAlloc

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

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

    virtual void Free(void* ptr, UPInt size, UPInt align)
    {
        GUNUSED2(size, align);
        _aligned_free(ptr);
        return true;
    }

    virtual void* Realloc(void* oldPtr, UPInt oldSize,
                          UPInt newSize, UPInt align)
    {
        GUNUSED(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 GSysAlloc interface and implements three virtual functions: Alloc, Free, and Realloc. Although implementation of these functions must honor alignment, GFx 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 GfxSystem constructor during Gfx initialization:

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

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

## 2.2 Using Fixed Memory Blocks for Gfx

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

```
void*          pmemChunk = ...;  
GSysAllocStatic blockAlloc(pmemChunk, 6*1024*1024);  
GfxSystem      gfxSystem(&blockAlloc);  
...
```

The above example passes a six megabyte chunk of memory to Gfx 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 Gfx, making sure that no files are loaded, or movie instances created, that would exceed the specified memory amount. Once GSysAllocStatic fails, it will return 0 from its `Alloc` implementation, causing Gfx to either fail or crash. If needed, developers can use a simple GSysAllocPaged wrapper object to detect this critical condition.

## 2.2.1 Memory Arenas

GFx 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 GFx files can be loaded and GFxMovieViews created, such that these regions are fully released once their occupying GFx objects are destroyed. A memory arena can be destroyed without shutting down the rest of GFx, or unloading unrelated GFx files. Once an arena is destroyed, the application can reuse all of its memory for other non-GFx 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 GFx. When GFx allocates memory it is obtained from the GSysAlloc object specified on GFxSystem initialization. This memory is released as data is unloaded and GFx 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, GFx 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 GFx. 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 GFx and other systems. In earlier versions of the SDK, developers would have had to shut down GFx to ensure that buffer memory was completely released. With GFx 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, GMemory::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 GMemory::CreateArena and give it a non-zero integer identifier (zero means global or default arena, which always exists). A GSysAlloc interface is needed to access the memory. For a fixed-size memory buffer, GSysAllocStatic may be used.

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

2. Call `GFxLoader::CreateMovie` / `GFxMovieDef::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
GPtr<GFxMovieDef> pMovieDef =
    *Loader.CreateMovie(filenameStr, GFxLoader::LoadWaitFrame1, 1);
...

// use arena 1 for the instance
GPtr<GFxMovieView> pMovie =
    *pnewMovieDef->CreateInstance(GFxMovieDef::MemoryParams(1), false);
```

3. Use the resulting `GFxMovieDef`/`GFxMovieView` objects as long as needed.
4. Release all of the created movies and objects allocated within an arena. `GFxMovieDef::WaitForLoadFinish` should be called before `Release` if `GFxThreadedTaskManager` 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 `GPtr` 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 `GMemory::ArenalsEmpty` 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;".
GMemory::DestroyArena(1);
```

The `GSysAlloc` object may now go out of scope, or be destroyed, after which point ‘pbuf1’ 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 Gfx. These include:

- *GSysAllocWinAPI* – uses VirtualAlloc/VirtualFree APIs and is the best choice on Microsoft platforms due to good alignment and its ability to leverage paging; and
- *GSysAllocPS3* – 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 Gfx WinAPI and PS3 GSysAlloc 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 dmalloc, for example.

### 2.3.1 Implementing GSysAllocPaged

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

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

```
class MySysAlloc : public GSysAllocPaged
{
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 `GSysAllocPaged` 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 `GSysAlloc::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 [GFX 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 `GFX`, so it can affect alignment support requirements imposed on a `GSysAlloc` implementation, as well as `GFX` performance and memory use efficiency. The `GSysAlloc::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, `GFX` 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.

- `Granularity` – the suggested granularity for GFx allocations. GFx 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 `GSysAlloc` 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 GFx 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 `GFxSystem` constructor during GFx initialization:

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



## 2.4 Memory Heaps

As mentioned, GfX 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 GfX 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 GfX objects, without having to consider the heap in which allocations will take place:

```
GPTr<GfXFileOpener> opener = *new GfXFileOpener();  
loader.SetFileOpener(opener);
```

In this case, the operator 'new' is used without any heap specification, so the memory allocation will take place on the GfX global heap. Since the number of configuration objects is small, this approach works fine. The global heap is created during GfXSystem 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 GfXMeshCacheManager class creates an internal heap to hold and constrain all of the tessellated shape data, the GfXLoader::CreateMovie() function creates a heap to hold the data loaded from a particular SWF/GFX file, and the GfXMovieDef::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 GfX 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 GfX.

Memory heaps are represented by the GMemoryHeap class, with instances created for dedicated GfX sub-systems or data sets. Each child memory heap is created using GMemoryHeap::CreateHeap on the global heap, and is destroyed by calling GMemoryHeap::Release. During heap lifetime, memory can be allocated from the heap by calling GMemoryHeap::Alloc function and can be released by calling GMemoryHeap::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 GSysAlloc interface.

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

- GHEAP\_ALLOC(heap, size, id)
- GHEAP\_MEMALIGN(heap, size, alignment, id)
- GHEAP\_NEW(heap) ClassName
- GFREE(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 GfxSpriteDef object may be created in a custom heap 'pHeap' with the following call:

```
GPtr<GfxSpriteDef> def = *GHEAP_NEW(pHeap) GfxSpriteDef(pdataDef);
```

Here, GfxSpriteDef is an internal class used in Gfx. Since most objects in Gfx are reference counted, they are often held in smart pointers such as GPtr<>, 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 GNewOverrideBase, 'delete' operator is used directly. It is not necessary to specify the heap to free memory, as the delete operator and GFREE(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 GString, or GArray. Consider the following sample code that declares a class and creates an instance of it in the heap:

```
class GfxSprite : public GRefCountBase<GfxSprite>
{
    GString          Name;
    GArray<GPtr<GfxSprite> > Children;

    GfxSprite(GString& name, ...) { ... }
};
```

```
GPTr<GFxSprite> sprite = *GHEAP_NEW(pHeap) GFxSprite(name);
sprite->DoSomething();
```

Although the GFxSprite 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 GFxSprite 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, GFx core makes use of special “auto heap” allocation macros:

- GHEAP\_AUTO\_ALLOC(ptr, size)
- GHEAP\_AUTO\_NEW(ptr) ClassName

These macros behave much like their GHEAP\_ALLOC and GHEAP\_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:

```
GMemoryHeap*    pheap = ...;
UByte *         pBuffer = (UByte*)GHEAP_ALLOC(pheap, 100,
GFxStatMV_ActionScript_Mem);
GPTr<GFxSprite> sprite = *GHEAP_AUTO_NEW(pBuffer + 5) GFxSprite();
...
sprite->DoSomething();
...
// Release sprite and free buffer memory.
sprite = 0;
GFREE(pBuffer);
```

In this example a memory buffer is created from a specified heap, and a GFxSprite object is created in the same heap. The peculiar thing about this example is that it does not just pass the ‘pBuffer’ pointer into GHEAP\_AUTO\_NEW. Instead, it passed an address *within* the buffer allocation. This illustrates a novel feature of the GFx 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, we the GFxSprite class presented earlier can be rewritten as follows:

```
class GFxSprite : public GRefCountBase<GFxSprite>
```

```

{
    GStringLH                Name;
    GArrayLH<GPtr<GFxSprite> > Children;

    GFxSprite(GString& name, ...) { ... }
};
GPtr<GFxSprite> sprite = *GHEAP_NEW(pHeap) GFxSprite(name);
sprite->DoSomething();

```

Here, the GString and GArray classes have been replaced by their “local heap” equivalents, GStringLH and GArrayLH. 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 Gfx core, making sure that memory is allocated from the right heap.

### 3 Memory Reports

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

Although the HUD UI has been removed from GfXPlayer to make it more lightweight, a memory report may be generated and sent to the console by pressing Ctrl + F5 keys. This report is one of the possible formats generated by the GMemoryHeap::MemReport function:

```
void MemReport(GStringBuffer& buffer, MemReportType detailed);  
void MemReport(GFxLog* pLog, MemReportType detailed);;
```

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

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

```
Memory 9,428K / 9,544K  
System Summary  
    System Memory Footprint          10,551,296  
    System Memory Used Space        10,432,512  
    Page Mapping Footprint           32,768  
    Page Mapping Used Space          26,624  
    Bookkeeping Footprint            114,688  
    Bookkeeping Used Space           101,776  
    Debug Info Footprint              524,288  
    Debug Info Used Space             350,208  
    HUD Footprint                     253,952  
    HUD Used Space                    157,856
```

- GMemoryHeap::MemReportSummary – The memory consumed by a few major categories within GfX, as shown in the following example:

```
Summary  
    Image          7,564,784  
    Sound           0
```

Video	0
Movie View	494,384
Movie Data	240,320
Mesh Cache	179,232
Font Cache	186,736

- **GMemoryHeap::MemReportMedium** – In addition to all of the information given by **MemReportBrief** and **MemReportSummary**, this report type outputs a heap summary for the Global heap. The format of the additional information is illustrated in the following example:

[Heap] Global	9,523,200
Heap Summary	
Total Footprint	9,523,200
Local Footprint	327,680
Child Footprint	9,195,520
Child Heaps	8
Local Used Space	241,808
Bookkeeping	6,096
DebugInfo	61,440
Segments	8
Granularity	16,384
Dynamic Granularity	16,384
Reserve	16,384
Allocations Count	1,135

- **GMemoryHeap::MemReportFull** – This memory type is what the **GFxPlayer** outputs when pressing **Ctrl-F5**. It includes all the information from **MemReportMedium**, but it includes all heaps, not just the Global heap. In addition, within each heap segment is a breakdown of the memory allocated, by stat ID. An example of the information given by heap is as follows:

[Heap] MovieData "3dgenerator.swf"	7,843,840
Heap Summary	
Total Footprint	7,843,840
Local Footprint	245,760
Child Footprint	7,598,080
Child Heaps	1
Local Used Space	240,320
Bookkeeping	5,040
DebugInfo	30,720
Segments	14
Granularity	8,192
Dynamic Granularity	8,192

Used Memory	
General	172
String	2,263
MovieDef	
CharDefs	44,420
ShapeData	2,872
Tags	33,456
Fonts	137,120
MD_Other	17,349
Allocations Count	583

- GMemoryHeap::MemReportHeapsOnly – This memory type generates a breakdown of the memory footprint per heap. Note that the values for each heap include the sum of all child heaps, which are indented below the parent heap information. An example follows:

[Heap] Global	9,523,200
[Heap] _Font_Cache	204,800
[Heap] _Mesh_Cache	196,608
[Heap] _FMOD_Heap	409,600
[Heap] MovieData "3dgenerator.swf"	7,843,840
[Heap] _Images	7,598,080
[Heap] MovieDef "3dgenerator.swf"	4,096
[Heap] MovieView "3dgenerator.swf"	536,576

- GMemoryHeap::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.swf	
Used Memory	8,277,279
General	1,720
Image	7,564,784
String	2,790
MovieDef	235,537
CharDefs	44,420
ShapeData	2,872
Tags	33,456
Fonts	137,184
MD_Other	17,605
MovieView	449,696
MovieClip	62,796
ActionScript	364,700

Text	11,172
MV_Other	11,028
FontCache	22,752
Batches	22,752

- GMemoryHeap:: MemReportSimpleBrief – This memory report type outputs the total memory consumed by Gfx for each category (stat ID). In addition, it outputs the amount of memory consumed by overhead, and the total memory that is claimed by Gfx but not currently in use. An example of the format can be seen below:

Total Allocated Memory	9,773,056
Used Memory	9,229,758
Renderer	5,788
General	384,398
Image	7,695,904
Sound	172
String	8,393
Video	16
Debug Memory	8,192
StatBag	8,192
RenderGen	184,367
Vertices	8,596
MeshFill	21,072
MeshStroke	12,944
Tessellator	69,971
Rasterizer	7,096
Stroker	1,664
EdgeAA	36,380
StrokerAA	6,656
RG_Other	19,848
MovieDef	435,125
CharDefs	58,026
ShapeData	2,872
Tags	33,456
Fonts	269,424
Sounds	8
ActionOps	53,662
MD_Other	17,677
MovieView	449,718
MovieClip	62,796
ActionScript	364,700
Text	11,162



MV_Other	11,060
IME	6,023
FontCache	51,662
Batches	24,532
GlyphCache	16,431
FC_Other	3,936
Overhead	91,952
Unused Memory	143,98

- GMemoryHeap:: MemReportSimple – This memory report type has the same structure as MemReportSimpleBrief , but includes a heap breakdown for each stat ID, overhead, and unused memory. An example of a portion of this report type is as follows:

Total Allocated Memory	9,773,056
Used Memory	9,228,500
Renderer	5,788
[Heap] Global	5,788
General	382,504
[Heap] Global	18,944
[Heap] _Mesh_Cache	1,896
[Heap] _FMOD_Heap	359,944
[Heap] MovieData "3dgenerator.swf"	172
[Heap] MovieDef "3dgenerator.swf"	1,448
[Heap] MovieView "3dgenerator.swf"	100
Image	7,695,904
[Heap] _Font_Cache	131,120
[Heap] _ImagesMovieData "3dgenerator.swf"	7,564,784
Sound	172
[Heap] Global	172
String	8,393
[Heap] Global	5,604
[Heap] MovieData "3dgenerator.swf"	2,263
[Heap] MovieView "3dgenerator.swf"	526
...	

As mentioned above, the Gfx 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 GfxFMovieView 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 or 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 tessellator and EdgeAA. This category grows as new shapes are tessellated, up to a limit.
- **FontCache** – contains cached font data.

## 4 Garbage Collection

GfX 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, GfX 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 GFx 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 GFx source code can do this, since it requires rebuilding GFx.

To disable garbage collection, open the file *Include/GConfig.h* and uncomment the line with `GFC_NO_GC` macro, which is commented out by default.

```
// Disable garbage collection
#define GFC_NO_GC
```

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

## 4.1 Configuring the memory heap to be used with garbage collection

GFx 3.0 and higher allows configuring of the memory heap that hosts the garbage collector. GFx 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:

```
GFxMovieView* GFxMovieDef::CreateInstance(const MemoryParams& memParams =
                                          GFxMovieDef::MemoryParams(),
                                          bool initFirstFrame = true)
```

The `MemoryParams` structure that is expected as the first argument is defined as a nested class inside the `GFxMovieDef` class as follows:

```
struct MemoryParams
{
    GMemoryHeap::HeapDesc    Desc;
    Float                   HeapLimitMultiplier;
    UInt                    MaxCollectionRoots;
    UInt                    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 GFx 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 `MovieView` heap may be created in two steps:

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

GFxMovieView* GFxMovieDef::CreateInstance(GFxMovieDef::MemoryContext* memContext,
    bool initFirstFrame = true)
```

The `MemoryContext` object encapsulates the `MovieView` heap, as well as other heap-specific objects, such as the garbage collector. This second approach of creating the `MovieView` 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 `MovieViews` 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, GfX 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 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 GfXMovieView::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.