Scaleform GFx

GFx Version Migration Guide

This document is a guide to upgrading from earlier versions of GFx (GFx 2.2 onwards) to the latest versions.

Authors:	Ankur Mohan, Michael Antonov
Version:	2.07
Last Edited:	September 7, 2010

Scaleform®

Copyright Notice

How to Contact Autodesk Scaleform:

Document	Scaleform GFx Version Migration Guide
Address	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. GFx 2.2–3.0 Changes

Scaleform® GFx™ 3.0 introduced several new features including heap based memory system, garbage collection, video support, and the common lightweight interface kit (CLIK™). To support the new features and for optimization, changes have been made to the renderer and new configurations have been added. This document summarizes the transition changes from GFx 2.2 to 3.0 and higher versions.

1.1. Include Directories

- **Eval/Lib customers:** Public GFx headers are located at *$(GFxSDK)/Include*
  GRenderer files are located at *Src/GRenderer*. This is unchanged from GFx 2.2
- **Src Customers:** Now the public GFx headers are only in */Include* instead of */Src/…* This may require you to update your additional include directories settings.

1.2. Third Party Libs

For all systems other than Windows and Xbox 360, third-party libraries are built automatically. On Windows and Xbox, all GFx 3.x solutions now link to release third-party libraries. This is because users typically do not debug libjpeg and zlib. To rebuild our third-party libraries, on Windows, users must load the *Projects/Win32/Msvcxx/SDK/GFx 3rd Party.sln* solution. This is different from 2.2, where we included the third-party projects in the SDK solutions.

1.2.1. New Configurations

In GFx 3.x, GFx lib is provided in three configurations-

- **Debug –** Debug build with Debug info and Memory Tracking. Compiler optimizations are disabled.
- **DebugOpt** – The DebugOpt configuration stands for an "optimized debug" library intended for use in development. This library is built with compiler optimization enabled, /Ob1 option for inline function expansion and GFC_BUILD_DEBUG defined; it includes debug symbol information and supports memory leak detection and statistics. Unless you are debugging the internals of GFx (in which case Debug build is the best choice), DebugOpt library should improve performance of your debug builds, while still retaining the benefits of debug error checking and memory reports.
- **Release** – Compiler optimization enabled, no debug information, no statistics tracking.

On Windows and Xbox 360:

- C++ exceptions and RTTI are always disabled.
- Debug – Default inline function expansion (/Ob0) is used for this build.
- DebugOpt - /Ob1 option for inline function expansion.

On Windows only:

In GFx 2.2, we had four configurations for the GFx lib: debug, debug static, release and release static.

- Debug lib was built with the Multi-threaded debug (/MTd) option (links with LIBCMTD.lib).
- Debug Static lib was built with the Multi-threaded debug (/MDd) option (links with MSVCRTD.lib).
- Release lib was built with the Multi-threaded debug (/MD) option (links with MSVCRT.lib).
- Release Static lib was built with the Multi-threaded debug (/MT) option (links with LIBCMT.lib).

These configurations were provided so that while building their executable, customers could decide whether they wanted to statically link with CRT libraries (using the static libs) or dynamically link (using non-static libs) to keep the executable size small.

Now we use the compiler switch /Zl to omit writing default lib entries to the GFx.lib file allowing the same version of the GFx lib to be linked whether the customer wants to dynamically or statically link CRT libs while building their applications.

On all other systems:

By default, C++ exceptions and RTTI are enabled. Each configuration can have "+NoRTTI" added (for example, "Debug+NoRTTI") to disable C++ exceptions and RTTI. Libs for each configuration are included. The configurations for Wii is slightly different; see readme_wii.txt. See readme_make.txt for more information on the GFx build system.

1.3. Library Name Changes

The names of the GFx libs have now been changed across all platforms supported by GFx.

- GFx lib renamed to libgfx lib
- GFxIME renamed to libgfx_ime

Other important libs include libgrenderer_d3d9 and libgfx_expat.

1.4. Multiple PlayStation 3 Versions

We no longer post multiple PLAYSTATION® versions, as the SDK is now stable enough that any version 250 or higher can work with what we post.

1.5. FxDelegate Sample

A new sample executable demonstrating a more robust communication mechanism between Adobe® Flash® content and C++ has been added (applicable for only GFx 3.0). The FxDelegate sample utilizes ExternalInterface in favor of fscommand to take advantage of the extended capabilities offered by FxDelegate. The SWF required to run this sample was built using Scaleform CLIK™ (see below). The source code for this executable can be found under *$(GFxSDK)/Apps/Samples/GameDelegate,* and the projects can be found under *$(GFxSDK)/Projects/.../Samples* on Windows platform*.*

1.6. Scaleform CLIK

Scaleform GFx 3.0 introduces the Common Lightweight Interface Kit (CLIK), which is a new UI component architecture built with GFx as the primary target. CLIK was designed from the ground up to be lightweight in terms of memory and performance, and provides a robust prototyping workflow to transition assets from design to shipping quickly, using its skinning capabilities. Note: CLIK is NOT compatible with GFx versions 2.2 and below. It requires the garbage collection system introduced in GFx 3.0, as well as several new extensions.

1.7. Memory Changes

The debug, block and standard allocator interfaces have been replaced by a single GSysAlloc interface, installable by the end user. Several default implementations of this interface are provided, including:

- *GSysAllocMalloc* – relies on malloc() and free() calls;
- *GSysAllocStatic* – allows developers to specify a single large memory block for GFx use;
- *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.

Internally, GFx relies on the GMemoryHeap class to organize heaps and make allocations. In GFx 3.0 Alpha and Beta 1, developers had to override GMemoryHeap implementation to substitute the allocator; this is, however, no longer possible as GMemoryHeap interface is now statically linked.

Developer who adopted the early version of GFx 3.0 will need to transition to the new GSysAlloc API, which should be easy to do as described below.

If you need to replace the system allocator you must follow two steps:

1) Create your own implementation of GSysAlloc interface, which defines GetInfo, Alloc, and Free methods.

2) Provide an instance of this allocator in GFxSystem constructor during GFx initialization.

1.7.1. Implementing your own GSysAlloc

The simplest way to implement your own memory heap is to copy our GSysAllocMalloc implementation and then modify it to make calls to your own memory allocator or heap. Slightly modified GSysAllocMalloc allocator code is included below for reference:

```
class MySysAlloc : public GSysAlloc
{
  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, the implementation of the allocator is relatively simple. MySysAlloc class derives from a base GSysAlloc interface and implements three virtual functions: `GetInfo`, `Alloc` and `Free`. There is also a `ReallocInPlace` function that can be implemented as an optimization, but is omitted here.

- `GetInfo()` – Returns allocator alignment support capabilities and granularity by filling in 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 our case we've set MaxAlign value to 1 byte, we can safely ignore the second argument.

- `Free` – Free memory earlier allocated with Alloc. The free function receive extra size and align arguments that tell it the size of the original allocation; since they are not needed for our 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.

As can be seen, the behavior of `Alloc` and `Free` functions is fairly standard, so the only function of particular interest is `GetInfo`. This function reports the capabilities of your allocator to GFx, so it can affect alignment support requirements imposed on your GSysAlloc implementation, as well as GFx performance and memory use efficiency. `GSysAlloc::Info` structure contains six values:

- `MinAlign` – minimum alignment that your allocator will apply to ALL allocations.
- `MaxAlign` – the maximum alignment that your allocator can support. If this value is set to 0, GFx will assume any requested alignment is supported. If the reported value 1, no alignment support is assumed, so align argument can be ignored in your Alloc implementation.
- `Granularity` – suggested granularity for GFx allocations; GFx will try to use at least this size for allocation requests. If your allocator is not alignment friendly, as is the case with malloc, we recommend using a value of at least 64K here.
- `HasRealloc` – a boolean flag that specifies whether your implementation supports `ReallocInPlace`; in our case we return false.
- `SysDirectThreshold`-defines the global size threshold, when not null. If the allocation size is greater or equal to SysDirectThreshold, it 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 slows down the allocator. MaxHeapGranularity must be at least 4096 and a multiple of 4096.

As can be seen from the `MaxAlign` argument description, GSysAlloc interface is easy to implement if your 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. *Your allocator implementation doesn't support alignment*, or handles it inefficiently. If this is the case, simply set `MaxAlign` value to 1 and let GFx do all of the needed alignment work internally.
2. *Your allocator handles alignment efficiently.* If this is the case, set `MaxAlign` to 0, or the largest alignment size you can support, and implement `Alloc` function to properly handle alignment.
3. *Your 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 of the allocation sizes to the OS.

In GFx 3.0 and higher, 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 a part of the allocation initialization function which calls code that uses GFx; however, it can also be a part of another allocated object whose lifetime exceeds that of GFx (GFxSystem should NOT be globally declared). If you find such use inconvenient, you can use GFxSystem::Init() and GFxSystem::Destroy() static functions instead, without creating the object. Similar to the GFxSystem constructor, GFxSystem::Init() takes a GSysAlloc pointer argument.

1.8. Renderer Changes

Several new features were added to the GRenderer interface for 3.0, mostly to support video playback. This information is relevant only if you decide to override the default GRenderer implementation provided by GFx.

If your application does not use the video player, the new functions except for GetStats will not be called (Bind and IsYUVTexture are called internally by our renderers).

GTexture:

```
virtual bool   InitMappableTexture(int width, int height,
                                    GImage::ImageFormat format,
                                    int mipmaps, int targetWidth = 0,
                                    int targetHeight = 0);
```

Purpose: Initializes a mappable texture (one that can have Map and Unmap called on it). Contents after initialization are undefined.

In current version, this is only used by the video player, format is always Image_RGBA_8888 (except for YUV; see below), and mipmaps is 0.

Parameters: The same as for `InitTexture.`

```
virtual int    Map(int level, int n, MapRect* maps, int flags);
```

Purpose: Causes the specified mipmap level of the texture to be accessible in system memory. Fills in the array of MapRect structs describing the region of memory where the texture contents can be accessed. Most textures will be accessed with one MapRect. This is analogous to the "Lock" function on Direct3D9.

Return Value: Returns the number of MapRects actually needed or 0 for unsupported or error.

Parameters:
`MapRect* maps`**:** Array of MapRect structs describing the region of memory where the texture contents can be accessed.
`n`**:** Size of the maps array.
`Flags`**:** Can be Map_KeepOld; if set, requires that the original texture contents be preserved after Map and Unmap; otherwise the caller intends to overwrite the entire texture. Requires that the texture be initialized with InitMappableTexture.

Comments:
The texture cannot be used for rendering while mapped. Only one outstanding map per texture is supported.

The order of color components and memory layout used is platform specific and was chosen to match the video decoder's format; see the sample renderers.

```
virtual bool     Unmap(int level, int n, MapRect* maps, int flags);
```

Purpose: Completes access to a mapped texture level. Pass the maps array and count returned by Map. Flags are ignored. This is analogous to the "Unlock" function on Direct3D9.

Return Value: Returns a value of 1 if the map/write/unmap operation sequence was successful; or 0 if not (meaning that the texture contents were lost and became undefined).

```
virtual int         IsYUVTexture()
```

Purpose: Indicates whether this is a YUV or YUV+A texture.

Return Value:

2: YUV+A

1: YUV
0: All others

Comments: This function is only called by the renderer internally, so a custom renderer does not have to use it.

```
 virtual void  Bind(int stage, GRenderer::BitmapWrapMode WrapMode,
               GRenderer::BitmapSampleMode SampleMode, bool useMipmaps);
```

Comments: This function is only called by the renderer internally, so a custom renderer does not have to use it. It is used to set states for sampling from the texture, including binding the texture object to a shader parameter or texture unit.

GRenderer:

```
virtual GTexture*   CreateTextureYUV()
```

Purpose: Creates a YUV(+A) texture, if supported by renderer and/or detected hardware. These textures are only used by the video player, and are only rendered with DrawBitmaps (so your renderer does not have to support YUV in the possible combinations of FillGouraudTex shaders). The video player will first try to use a YUV texture by calling CreateTextureYUV; it will fall back to RGBA textures if NULL is returned. On consoles with pixel shaders, video playback with RGBA textures may result in unusable performance.

YUV Textures

Several GTexture functions behave differently on YUV(+A) textures.

YUV(+A) textures are only initialized with InitMappableTexture, and only accessed through Map and Unmap. The format parameter of InitMappableTexture has the following meaning:

Image_RGB_888 → YUV
Image_RGBA_8888 → YUV+A

All other formats are invalid. Each YUV(+A) texture is internally stored as three or four separate single component textures; U and V are quarter size (Each dimension is half of the original size; for example, if 512x512 is specified for size, then the U and V channels will be 256x256). When a YUV texture is mapped, it will use 3 (4 for YUV+A) MapRects, one per channel. The video player expects this layout, so it is not possible to use any other texture representation.

To render the texture, a pixel shader that converts YUV to RGB is necessary; see the sample renderers.

```
virtual void    GetStats(GStatBag* pbag, bool reset)
```

Purpose: Gets renderer statistics (primitives drawn, texture memory usage, etc.). See GRendererD3DxImpl.cpp for the stat definitions.

1.9. *Miscellaneous Information*

- GFX and USM files are associated with the FxMediaPlayerAMP.exe application under */Bin/* on Windows platform (refer to the Getting Started with Video Guide).
- Our installers now install four system fonts that can be found in */3rdParty/MonotypeFonts*
    - EurostileNextLTPro-BoldExt.ttf
    - IT808___.ttf
    - slate_mo.ttf
    - slate_mo_bd.ttf

- For the GFxPlayer, we now have 6 configurations

  On Windows, for each DirectX version and GL:
    - Debug
    - Debug_Static
    - DebugOpt
    - DebugOpt_Static
    - Release
    - Release_Static

  For Mac and Linux version:

    - Debug
    - Debug+NoRTTI
    - DebugOpt
    - DebugOpt+NoRTTI
    - Release
    - Release+NoRTTI

  "+" in configuration names becomes "_" (underscore) on disk.

2. API changes in Scaleform GFx 3.0

2.1. GFxTranslator

GFxTranslator API in GFx 3. 0 was changed in order to provide more information to the Translate method. In particular, the Translate method has access to a textfield's instance name that might be used as a key (or part of it).

2.1.1. GFxTranslator::Translate

Signature of this method has been changed to:

```
 virtual void Translate(TranslateInfo* ptranslateInfo);
```

Translate method implements a UTF-8/UCS-2 translation interface and performs a lookup of the 'ptranslateInfo->GetKey()' string for language translation, filling in the destination string buffer by calling 'SetResult' or 'SetResultHtml' method. 'ptranslateInfo' is guaranteed to be not null. If neither 'SetResult' nor 'SetResultHtml' is called then original text will not be changed.

Parameters:
ptranslateInfo – Pointer on TranslateInfo class that provides data exchange between GFx core and user code and is guaranteed to be not null.

The TranslateInfo class provides data to and from the Translate method such as the original text, name of textfield's instance, resulting translated text. It has the following methods:

```
const char* GetInstanceName()
```
An input method that returns the instance name of the text field being translated.

```
const char* GetKey()
```
An input method that returns the "key" string, or the original text value of the text field being translated (UTF-8).

```
bool IsKeyHtml()
```
Returns true if the key string (returned by GetKey()) is HTML.

```
void SetResult(const wchar_t* presultText, UPInt resultLen = GFC_MAX_UPINT);
void SetResult(const char* presultTextUTF8, UPInt resultLen =
                                                           GFC_MAX_UPINT);
```

SetResult sets the translated string as a plain text. Wide character and UTF-8 variants are supported. Length is optional; and a null-terminated string should be used if length is omitted.

```
void SetResultHtml(const wchar_t* presultHtml, UPInt resultLen =
                                                           GFC_MAX_UPINT); void
SetResultHtml(const char* presultHtmlUTF8, UPInt resultLen =
                                                           GFC_MAX_UPINT);
```

SetResultHtml sets the translated string as a HTML text. HTML will be parsed before applying to the textfield. Wide-character and UTF-8 variants are supported. Length is optional; and a null-terminated string should be used if length is omitted.

An example of Translate method implementation:

```
      virtual bool Translate(TranslateInfo* ptranslateInfo)
      {
          // Translate '$hi' into bold 'Hello!'.
          const char* pkey = ptranslateInfo->GetKey();
          if (pkey[0] == '$')
          {
              GString key(pkey);
              if (pkey == "$hi")
              {
                 ptranslateInfo->SetResultHtml("<b>Hello!</b>");
              }
           }
      }
```

2.1.2. GFxTranslator::TranslatorCaps

Flags Cap_CustomWordWrapping and Cap_ReturnHtml have been removed.

The Cap_CustomWordWrapping was used to force the custom word-wrapping callback OnWordWrapping to be invoked once the necessity of word wrapping for any text field was determined. This flag is no longer necessary; use the GFxTranslator constructor with a parameter instead:

```
explicit GFxTranslator(UInt wwMode);
```

If the parameter `wwMode` is set to value other than WWT_Default (0), then the OnWordWrapping virtual method will be invoked once the necessity of word wrapping for any text field is determined.

The Cap_ReturnHtml has been replaced by the TranslateInfo::SetResultHtml method. Use this method if the returning result is HTML, or use SetResult if the result is plain text. There is no longer a need to set a single global flag.

2.2. Garbage Collection Related Changes

1. The method GFxMovieDef::CreateInstance, which creates a movie view instance, now has two optional parameters instead of one. The first parameter is a reference to MemoryParams structure that contains memory configuration parameters set for the movie heap. The second parameter is used to initialize the frame 1 tag of the instances:

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

   Precise memory limits can be enforced on the movie heap through the MemoryParams structure. If the limit set for the movie heap is reached, the heap will garbage collect and try to maintain the size at the specified limit. The MemoryParams structure contains the following members that describe garbage collector and heap behavior.

   ```
   struct MemoryParams
   {
         // Heap descriptor for the created movie heap. A memory limit on
         // heap can be specified through Desc.limit.
         GMemoryHeap::HeapDesc Desc;

         // Multiplier to determine the expansion of the heap limit if the
         // memory limit for movie frame exceeds.
         Float HeapLimitMultiplier;

         // Number of roots before garbage collection is executed. This is
         // an initial value of max roots cap; it might be increased
         // during the execution.
         UInt MaxCollectionRoots;

         // Number of frames after which collection is forced, even if the
         // max roots cap is not reached. It is useful to perform
         // intermediate collections in the case if current max roots cap
         // is high, to reduce the cost of that collection when it
         // occurs
         UInt FramesBetweenCollections;

   };
   ```

Refer to the Memory System Overview document for further details on configuring memory heap to be used with garbage collection.

2. The method GFxMovieView::ForceCollectGarbage has been added:

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

   This method might be used to force garbage collector execution by user's application. It does nothing if garbage collection is OFF.

3. Changes in Scaleform GFx 3.1

3.1. Renderer Migration from GFx 3.0

This section is intended for users migrating from GFx 3.0 to 3.1 and provides information on customization of the renderer. If you have not customized GRenderer, replace the old implementation with the one from GFx 3.1. The new sample renderers support non power of 2 textures when available.

3.1.1. GTexture Interface

The GTexture interface no longer requires the targetWidth and targetHeight properties. InitTexture now takes a usage parameter, specifying how the texture will be used in a fill style. Currently only one flag is defined, `Usage_Wrap`. This flag must be set if the texture will be used in a fill style with wrapping (i.e., repeating).

Old relevant prototypes:

```
bool InitTexture(UInt texID, SInt width, SInt height, bool deleteTexture);
bool InitTexture(GImageBase* pim, int targetWidth=0, int targetHeight=0);
bool InitTexture(int width, int height, GImage::ImageFormat format, int mipmaps,
                                         int targetWidth=0, int targetHeight=0);
bool InitTextureFromFile(const char* pfilename, int targetWidth=0, int
                                                               targetHeight=0);
bool InitMappableTexture(int width, int height, GImage::ImageFormat format, int
                               mipmaps, int targetWidth=0, int targetHeight=0);
```

New prototypes:

```
bool InitTexture(UInt texID, bool deleteTexture);
bool InitTexture(GImageBase* pim, UInt usage = Usage_Wrap);
bool InitTexture(int width, int height, GImage::ImageFormat format, int mipmaps);
bool InitMappableTexture(int width, int height, GImage::ImageFormat format, int
                                                                       mipmaps);
```

The first InitTexture function is platform specific and may be slightly different in your renderer.

3.1.2. RenderCaps

The capability flags (GRenderer::RenderCapBits) for non power of 2 texture support are provided:

`Cap_TexNonPower2`	Texture size does not have to be a power of two for clamped, non-mipmapped use.

`Cap_TexNonPower2Wrap`	Non power of 2 textures can be wrapped (Usage_Wrap).
`Cap_TexNonPower2Mip`	Non power of 2 textures can have mipmap levels.

These caps control whether images passed to InitTexture(GImageBase*,UInt usage) will be resized internally. All provided sample renderers now support non power of 2 textures when available.

3.1.3. GRenderer::FillTexture::TextureMatrix

In GFx 3.0 and earlier versions, the TextureMatrix used a coordinate system based on the size of the image as it was defined in the Flash content. This required the renderer to track those dimensions (`targetWidth, targetHeight`) as well as the actual texture dimensions in case the sizes were different, for example, to convert to power of 2 sizes or for a render target texture.

In GFx 3.1, the TextureMatrix uses normalized texture coordinates (0-1). The renderer does not have to store the original Flash image size. The GFx Flash player now handles fill images sizes internally. See the ApplyFillTexture function in one of the included sample renderers.

Typically, ApplyFillTexture would be changed from:

```
Float   InvWidth = 1.0f / ptexture->Width;
Float   InvHeight = 1.0f / ptexture->Height;
const GRenderer::Matrix&    m = fill.TextureMatrix;
Float   p[8];

p[0] = m.M_[0][0] * InvWidth;
p[1] = m.M_[0][1] * InvWidth;
p[2] = 0;
p[3] = m.M_[0][2] * InvWidth;
p[4] = m.M_[1][0] * InvHeight;
p[5] = m.M_[1][1] * InvHeight;
p[6] = 0;
p[7] = m.M_[1][2] * InvHeight;
```

to

```
const GRenderer::Matrix&    m = fill.TextureMatrix;
Float   p[8];

p[0] = m.M_[0][0];
p[1] = m.M_[0][1];
p[2] = 0;
p[3] = m.M_[0][2];
p[4] = m.M_[1][0];
p[5] = m.M_[1][1];
p[6] = 0;
```

```
        p[7] = m.M_[1][2];
```

If your renderer uses 4x4 matrices, the indices of p[] will be different but the logic should be similar.

3.1.4. Interaction with Image Packer

The image packer is primarily intended to be used from gfxexport, but it can also be enabled at runtime. See GFxImagePackParams in GFxLoader.h. If the packer is enabled at runtime, it will use the NonPower2 caps above to configure itself.

A custom renderer needs no changes specific to the image packer; the TextureMatrix is adjusted by GFxPlayer before calling GRenderer.

3.2. Direct Access API

GFx 3.1 introduces the Direct Access API which allows users to access and manipulate objects in the AS2 runtime much more efficiently than previously possible. This API adds functionality to the GFxValue to support AS2 Objects, Arrays and display objects (such as MovieClips, TextFields and Buttons). Several new additions were also introduced to GFxMovie, which allows users to create new Object and Array instances, as well as managed strings in the runtime (See GFxMovie::CreateObject/GFxMovie::CreateArray). With the new API, methods that return GFxValues such as ExternalInterface, are now able to send back a direct reference to the object that can be stored in application space. The lifetime of the reference is guaranteed as long as the runtime is alive.

As an example, the following sample code

```
GFxValue x, y, custom;
pMovie->GetVariable("_root.foo.bar._x", &x);
pMovie->GetVariable("_root.foo.bar._y", &y);
pMovie->GetVariable("_root.foo.bar.custom", &custom);
x.SetNumber(x.GetNumber() + 5);         // Horizontal translation
pMovie->SetVariable("_root.foo.bar._x", x);
```

can be rewritten with the new Direct Access API for better performance as:

```
GFxValue barMC, x, y, custom;
pMovie->GetVariable("_root.foo.bar", &barMC);
GFxValue::DisplayInfo barInfo;
```

```
barMC.GetDisplayInfo(&barInfo);
barMC.GetMember("custom", &custom);
barInfo.SetX(barInfo.GetX() + 5);         // Horizontal translation
barMC.SetDisplayInfo(barInfo);
```

The new API provides a much faster code path for retrieving and updating the display properties of display objects (MovieClips, TextFields and Buttons) on stage such as position, rotation, scaling, visibility and alpha, as well as providing access to the display matrix for advanced manipulation.

4. Changes in Scaleform GFx 3.2

This section features the new functionalities and renderer changes when migrating from GFx 3.1 to 3.2.

4.1. *Flash Filter Support*

This release includes preliminary support for some Flash filters. Not all platforms are supported:

D3D9 -11, PS 2.0 and higher	Blur, Drop Shadow, Bevel, Glow and Adjust Color
GL 2.0+	Blur, Drop Shadow, Bevel, Glow and Adjust Color
PS3 and Xbox 360	Blur, Drop Shadow, Bevel, Glow and Adjust Color
Wii	Blur, Drop Shadow, Glow and Adjust Color
All others	Not supported

Blur, Drop Shadow, Glow, Bevel and Adjust Color are supported, as shown above.

If you use any filters, your application must call the new GRenderer function, SetDisplayRenderTarget. See the Direct3DWin32App, OpenGLWin32App etc, and the sample source files. Unlike GFx 3.1 and earlier versions, changing the render target externally will result in corruption if filters are used.

If you use any filters with a renderer that returns `Cap_RenderTargetPrePass`, your application must call the new GFxMovieView function, DisplayPrePass. Call this function after Advance for the current frame, and before your application begins drawing the primary framebuffer. DisplayPrePass will overwrite the limited framebuffer memory on these systems.

4.2. *AMP Integration*

The AMP profiler allows for analyzing the CPU usage, graphics rendering and memory allocations within an application and optimize their Flash contents by removing memory and performance bottlenecks. The visual profiler in AMP lets you view the detailed profile statistics and easily pinpoint the bottlenecks in the GFx application.

This section briefly describes how to connect AMP and use the visual profiler in your applications.

4.2.1. AMP Support Setup

1. Application Control

In addition to memory and performance statistics, AMP has the ability to remotely control settings of the profiled application, such as wireframe mode, or visual profiling mode. Many of the supported settings are application-specific, and may be implemented by your application. Any such functionality you choose not to implement will simply not be available to AMP.

Follow the following steps to handle app-control messages sent from AMP:

- Implement a custom app-control callback class that derives from GFxAmpAppControlInterface, and override the HandleAmpRequest method to handle GFxAmpMessageAppControl messages from AMP.

- Install the custom handler by calling `GFxAmpServer::GetInstance().SetAppControlCallback` on startup.

- Inform AMP of the supported functionality by calling `GFxAmpServer::GetInstance().SetAppControlCaps`. The argument to this method is a GFxAmpMessageAppControl message, with the supported functionality set to true. For example:

```
 GFxAmpMessageAppControl caps;
caps.SetCurveToleranceDown(true);
caps.SetCurveToleranceUp(true);
caps.SetNextFont(true);
caps.SetRestartMovie(true);
caps.SetToggleAaMode(true);
caps.SetToggleAmpRecording(true);
caps.SetToggleFastForward(true);
caps.SetToggleInstructionProfile(true);
caps.SetToggleOverdraw(true);
caps.SetToggleStrokeType(true);
caps.SetTogglePause(true);
caps.SetToggleWireframe(true);
GFxAmpServer::GetInstance().SetAppControlCaps(&caps);
```

2. Status

AMP has the ability to display the name of the connected application in its status bar. This information is communicated by calling `GFxAmpServer::GetInstance().SetConnectedApp.`

Also, the AMP UI can display the current application state, but for this it needs to be informed whenever the application state changes. Use the following methods to update the state:

`GFxAmpServer::GetInstance().SetState`, with the first argument one of the following AmpServerStateType enumeration types:

- `Amp_RenderOverdraw`
- `Amp_App_Wireframe`
- `Amp_App_Paused`
- `Amp_App_FastForward`

```
GFxAmpServer::GetInstance().SetAaMode
GFxAmpServer::GetInstance().SetStrokeType
GFxAmpServer::GetInstance().SetLocale
GFxAmpServer::GetInstance().SetCurveTolerance
```

3. Port

Your application will create a listening socket on port 7534 by default. A different port may be set by calling `GFxAmpServer::GetInstance().SetListeningPort`.

4.2.2. Frame Callback

AMP needs to receive information every game frame in order to display memory and performance statistics. If your application does not use a custom renderer, you need not do anything to update AMP, as the information is sent from within GRenderer::EndFrame. If your application does not call GRenderer::EndFrame, then you should insert a call to update AMP directly:

```
GFxAmpServer::GetInstance().AdvanceFrame();
```

4.2.3. Visual Profiling

To use the AMP visual profiler in your application, you need to provide a GAmpRenderer instance to the GFxLoader instead of your existing GRenderer instance. GAmpRenderer is a template that "wraps" the actual renderer and adds support for the visual profiling features. Construct the instance like this (replace GRendererOGL with whatever type you are using):

```
GAmpRendererImpl<GRendererOGL>* pamprenderer = new
                                   GAmpRendererImpl<GRendererOGL>(pRenderer);
Loader.SetRenderer(pamprenderer);
GFxAmpServer::GetInstance().AddAmpRenderer(pamprenderer);
```

With the visual profiling on, all objects are displayed in green. Bright green indicates overdraw (16 layers for full brightness), masks are red and filters are blue (8 layers of masks or filters for full brightness). These colors blend together, so a bright yellow area is high overdraw with nested masks.

If you have a custom renderer, see section 4.3.

4.3. Custom Renderer Migration

If you do not have a GRenderer customized for your engine, and use one of the provided GRenderer implementations, you can skip this section (but see GRenderTarget and SetDisplayRenderTarget if you use Flash filters). When reviewing our sample renderers, use only GRendererD3DxImpl, GRendererOGL2Impl, GRendererPS3Impl, or GRendererXbox360Impl as the others do not have any of the new features (except 3D support).

4.3.1. 3D Support

```
void  SetPerspective3D(const GMatrix3D &projMatIn)
void  SetView3D(const GMatrix3D &viewMatIn)
void  SetWorld3D(const GMatrix3D *pWorldMatIn)
```

Set 3D matrices. These matrices completely replace the 2D ortho-matrix specified by the frame rect in BeginDisplay (but not the 2D matrix passed to SetMatrix).

The multiplication operators, (operator * and operator *=) in GMatrix2D were changed in GFx 3.2 to perform an Append operation instead of a Prepend. So to preserve the same behavior, code which uses those operators could be changed as follows:

Old code:
```
GMatrix2D m = a * b;
```

New code:
```
GMatrix2D m = a;
m.Prepend(b);
```

4.3.2. GTexture

The functions
```
bool InitTexture(int width, int height, GImage::ImageFormat format, int mipmaps)
bool InitMappableTexture(int width, int height, GImage::ImageFormat format,
                                             int mipmaps)
```

have been replaced with

```
bool InitDynamicTexture(int width, int height, GImage::ImageFormat format,
                                         int mipmaps, UInt usage)
```

The new usage parameter is one of:

`Usage_Update`	Same as old `InitTexture(int width, int height, GImage::ImageFormat format, int mipmaps).`
`Usage_Map`	Same as old InitMappableTexture
`Usage_RenderTarget`	Creates a texture that can be attached to a render target.

Multiple flags are not supported, but some systems will work anyway.

4.3.3. GAmpRenderer

Each renderer must define these typedefs (D3D9 is used as an example):

```
class GRendererD3D9
{
   typedef  GTextureD3D9       TextureType;
   typedef  IDirect3DTexture9*       NativeTextureType;
   typedef  GRenderTargetD3D9::D3D9RenderTargetParams   NativeRenderTargetType;
}
```

The TextureType class must derive from GTexture, have a default constructor, and contain two additional methods:

```
NativeTextureType   GetNativeTexture() const
bool  InitTexture(NativeTextureType nativeTexture, bool extra = 0)
```

Also, ensure that your renderer does not have any abstract functions which are not present in GRenderer.

If you have an assert as shown below in GFxFillStyle::Apply, it needs to be disabled:

```
   GASSERT (BitmapColorTransform.M_[3][1] < 1.0f);
```

To control it you can use the GAmpRenderer interface:

```
  pAmpRenderer->ProfileCounters(
       (UInt64(GAmpRenderer::Profile_Fill   | 16) << GAmpRenderer::Channel_Green) |
       (UInt64(GAmpRenderer::Profile_Mask   | 32) << GAmpRenderer::Channel_Red) |
       (UInt64(GAmpRenderer::Profile_Filter | 32) << GAmpRenderer::Channel_Blue));
```

This will make all drawing green (to count overdraw; 16 layers for full brightness), masks red, and filters blue (8 layers for full brightness). The colors will blend together, so masks may appear as yellow or brown.

GImageInfoBase::GetBytes should return the GFx-owned memory that is allocated for the image. AMP uses this value to create memory reports. This should include mips, if they are kept in memory. It should not include texture memory, which is not owned by GFx.

Similarly, GImageInfoBase::GetExternalBytes should return the non-GFx-owned memory for the image, in the case where images are allocated and owned by the game engine. You probably want to see texture memory associated with the image as part of the memory report, in which case you should include texture memory in the returned value.

4.3.4. GRenderTarget

This class is preliminary and subject to change in the near future. It is not necessary to use GRenderTarget unless you use Flash filters. When BitmapData is supported, it will use GRenderTarget also.

GRenderTarget represents a render target (framebuffer, surface, etc) to which rendering can be directed. Previously, render targets were managed entirely outside of GFx. However, recently introduced features require render targets internally. Since some platforms do not allow querying and restoring the previously set render target, applications that use those features (Flash filters) must set the primary render target using a GRenderTarget object.

`bool        InitRenderTarget(GTexture* ptarget)`

Create a render target that uses `ptarget` as the color buffer. Texture must have been created with `Usage_RenderTarget` (some systems won't enforce this). Allocates an internal depth/stencil buffer.

`bool        InitRenderTarget(...)`

Create a render target which references a native render target. Parameters are system specific.

4.3.5. GRenderer Render Target Support

These functions are preliminary and subject to change in the near future.

It is not necessary to use these functions unless you use Flash filters.

New caps from GetRenderCaps :

`Cap_RenderTargets`	Use render targets at any time. This flag is set on most systems that support render targets.
`Cap_RenderTargetPrePass`	Use render targets during prepass; one render target at a time. This flag is used on systems with limited framebuffer memory, allowing

the application to use the entire framebuffer when drawing the primary scene.

```
GRenderTargetXXX*   CreateRenderTarget()
```

Create an un-initialized render target.

```
void        SetDisplayRenderTarget(GRenderTarget* prt, bool setstate = 1)
```

Set the primary output render target. Use `setstate = false` if this render target is already set on the GPU (by the application) to avoid unnecessary expensive render target changes.

```
void        PushRenderTarget(const GRectF& frameRect, GRenderTarget* prt)
```

Push a render target onto the stack. The view is adjusted so that `frameRect` fills the framebuffer.

```
void        PopRenderTarget()
```

Finish the last pushed render target, and restore the previous one. The texture attached to the last render target can now be used for sampling.

```
GTextureXXX*   PushTempRenderTarget(const GRectF& frameRect, UInt targetW,
                                                              UInt targetH)
```

Push an internal render target and return the texture used to read from it. The returned texture must be AddRefed before calling this renderer again, and must be released when its contents are no longer needed.

4.3.6. Functions for Flash Filter Support

These functions are preliminary and subject to change in the near future. We recommend only supporting DrawColorMatrixRect in your custom implementation at this time, since improvements and optimizations will be added to our filter support in upcoming releases, but color matrix is simple and provides a way to integrate and test with render targets before the filter support is finished. Please contact support if there is a problem with the render target interface.

Currently a mix of static and dynamic (runtime compiled) shaders are used. In the future, more renderers will provide options for static shaders.

New caps from GetRenderCaps :

`Cap_Filter_Blurs`	DrawBlurRect is supported, if CheckFilterSupport returns non-zero for those parameters.
`Cap_Filter_ColorMatrix`	DrawColorMatrixRect is supported.

```
UInt    CheckFilterSupport(const BlurFilterParams& params)
```

Check whether requested filter parameters are supported and if multiple passes are required.

```
void    DrawBlurRect(GTexture* psrc, const GRectF& srcrect, const GRectF& destrect,
                     const BlurFilterParams& params)
```

Draw a quad from `psrc` to `destrect`, blurring or adding shadows as specified by `params`. `srcrect` is in pixels.

```
void   DrawColorMatrixRect(GTexture* psrcin, const GRectF& srcrect, const GRectF&
                           destrect, const Float *matrix)
```

Draw a quad from `psrc` to `destrect`, multiplying colors by the specified matrix (5x4 homogenous, 5th row is 00001 and not actually stored). `srcrect` is in pixels.

4.3.7. Call to Update AMP Every Frame

GRenderer::EndFrame now contains a call to update AMP. Therefore, if your custom renderer overrides EndFrame, it should also call GRenderer::EndFrame. If EndFrame is not called at all, then a call should be inserted to update AMP directly:

```
GFxAmpServer::GetInstance().AdvanceFrame();
```