Autodesk® Scaleform®

Scaleform Best Practices Guide

This document provides best practices for asset creation for use with Scaleform 4.0 and higher.

Author: Matthew Doyle, Sven Dixon
Version: 3.01
Last Edited: October 30, 2014

Autodesk® **GAME**WARE

Copyright Notice

Autodesk® Scaleform® 4.4

How to Contact Autodesk Scaleform:

Document	Scaleform Best Practices Guide
Address	Autodesk Scaleform Corporation
	6305 Ivy Lane, Suite 310
	Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com

Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Overview

This document contains a growing list of best practices when developing Adobe® Flash® content for use with Autodesk® Scaleform® 4.0 and higher versions. These practices are specifically geared towards improving memory and performance of Flash content in games; however they can also be applicable for other use cases. The content creation section in this document is targeted toward artists and designers, whereas the ActionScript™ (AS) section is targeted toward technical artists and engineers. The Heads-Up Display (HUD) Development section is a general overview of the possible development scenarios for HUD creation. We encourage both artists and engineers to read this section before developing a HUD system with Flash and Scaleform.

This document provides a compilation of a wide variety of information from different sources such as customer support requests, developer forum posts, and various Flash and AS resources on the web. Scaleform can be used on many different platforms (from cell phones to cutting-edge consoles and PCs) and integrated with many different engines, and as a result there are unfortunately no "one size fits all" solutions. Every game and project requires a different solution for optimal memory consumption and performance (e.g., code tweaks may be required when porting from a high-end PC to a handheld platform).

We have tried our best to provide relevant performance data along with these best practices; however, these results will not be the same for your project, as there are many variables to account for. We recommend all User Interface (UI) work be thoroughly tested and alternative solutions should be stress tested for each use case early on in the UI development process.

Flash designers and developers must write code and structure applications in a way that is intuitive and beneficial to themselves as well as to the other people who are working on the same project. This is particularly important in FLA files with many assets. Note that Flash Studio CS5 introduced an XML based content format that replaces the binary .FLA format. This is very useful for versioning and changelist tracking.

As it is common for more than one designer or developer to work on a single Flash project, teams benefit when everyone follows a standard set of guidelines for using Flash, organizing FLA files, and writing AS code. The sections in this document outline the best practices for writing AS and best practices when using the Flash authoring tool to create rich media content. Adopt best practices at all times, whether a designer or a developer, working alone or as part of a team. The following are a few reasons why learning and following best practices is beneficial:

- When working on Flash or AS documents:
    - Adopting consistent and efficient practices helps speed up workflow. It is faster to develop using established coding conventions, and easier to understand and remember how a

document was structured when editing it further. Additionally, code is often more portable within the framework of a larger project, and easier to reuse.

- When sharing FLA or AS files:
    - Other people editing the document can quickly find and understand AS, consistently modify code, and find and edit assets.
- When working on applications:
    - Multiple authors can work on an application with fewer conflicts and greater efficiency. Project or site administrators can manage and structure complex projects or applications with fewer conflicts or redundancies.
- When learning or teaching Flash and AS:
    - Learning how to build applications by using best practices and following coding conventions reduces the need to relearn particular methodologies. If students learning Flash practice consistent and better ways to structure code, they might learn the language faster and with less frustration.

Developers are bound to discover many more when they read these best practices and develop their own good habits. Consider the following topics as a guideline when working with Flash; developers may choose to follow some or all of the recommendations. Developers can also modify the recommendations to suit the way they work. Many of the guidelines in this chapter will help developers utilize a consistent way of working with Flash and writing AS.

Areas of possible optimization in Flash are as follows:

- Increase animation performance by optimizing graphics for fast redraw.
- Increase computing performance by writing code that executes quickly.

If code runs slowly, it is often a sign that the developer needs to reduce the scope of the application or investigate another method that solves the problem in a more efficient way. Developers should identify and remove bottlenecks, for example, by optimizing graphics or reducing the work done by ActionScript.

Often performance is about perception. If the developer attempts to perform too much work in a single frame, Flash doesn't have time to render the Stage, and the user perceives a slowdown. If the developer breaks up the amount of work being performed into smaller chunks, Flash can refresh the Stage at the prescribed frame rate, and there is no perceived slowdown.

1.1 General Usage

- Consider reducing the number of objects on the Stage. Add objects one at a time and take note when and by how much performance degrades.

- Avoid using the standard UI components (available in the Components panel in Flash). These components are designed to run on desktop computers and are not optimized to run in Scaleform 4.0 and higher. Utilize the Scaleform Common Lightweight Interface Kit (CLIK™) components instead.

1.2 Setup Considerations

- Consider reducing the depth of the object hierarchy. For example, if some objects don't move or rotate independently, they don't have to exist within their own transform group.
- A large user interface is typically best broken into separate Flash files. For example, an MMORPG's trading room would be one Flash interface and the life-meter HUD would be another. On the AS side, different files can be loaded through the various SWF loading APIs in both AS2 and AS3. These functions enable the application to load only the interfaces necessary at any given time and free content when no longer needed. Another alternative is to load and unload different SWF files via C++ API (creating different GFx::Movie instances).
- Scaleform 4.0 and higher supports multithreaded loading and playback. Background loading of multiple Flash files will not adversely impact performance if extra CPU cores are available. However, the developer must take care when loading multiple files from CD or DVD, as disc seeking is slow and may significantly slow down file loading.
- Avoid using Scenes as they often result in large SWF files.
- Realistic amounts of UI memory used for a game may fall in the following ranges:
  o Simple game HUD overlay: 400K–1.5M
  o Startup/Menu screen with animations and several screens: 800K–4M
  o Simple game fully in ActionScript with assets (Pacman): 700K–1.5M
  o Prolonged vector animation with constantly changing assets: 2M–10M+

2 Content Creation Memory and Performance Considerations

When developing Flash content, there are a number of considerations and optimizations that should be followed and implemented in order to get the best performance.

2.1 Draw Primitives

Draw primitives (DPs) are mesh objects created by Scaleform to render Flash elements to the display, such a shape . On platforms supporting batched (or instanced) rendering, Scaleform attempts to group shapes on the same and adjacent layers with the compatible properties into a single DP. Each DP is rendered independently, incurring a performance cost. In general, Display performance will tend to decrease linearly as more DPs are introduced into the scene; therefore it is a good practice to keep the DP count as low as possible. The number of DPs can be determined via the Scaleform Player HUD by pressing the (F2) key. This screen displays triangle counts, DPs, memory use and other optimization information.

The following are a few facts to help keep draw primitive count low:

1. DPs can only contain items which have the same properties (excluding shape with different solid colors). For example, items with different textures or blending modes can never be combined in a DP.
2. Overlapping objects on different layers cannot be combined into a DP, even if they have the same properties.
3. Gradient fills can increase the number of DPs if several of them are used in a shape at the same time.
4. Vector graphics with solid color fills and no strokes are very cheap.
5. Empty movie clips do not require a DP; however, a movie clip with objects in it will require the amount of DPs dictated by those objects.

The batching and/or instancing groups can be visualized in the Scaleform Player by enabling the option in AMP, or by pressing (Ctrl+J). In this mode, items with the exact same color are rendered as a single DP.

2.2 Movie Clips

1. Rather than hiding movie clips, it is best to delete them completely from the timeline when not in use, otherwise they may take up processing time during Advance.
2. Avoid excessive nesting of movie clips, as this will affect performance.
3. If it is necessary to hide a movie clip then use `_visible=false` rather than `_alpha=0` (in AS2) and `visible=false` rather than `alpha=0` (in AS3). Make sure you have stopped the animation in hidden movie clips by calling the "stop()" function. Otherwise, invisible animation will still take place and affect performance.

2.3 Artwork

2.3.1 Bitmaps vs. Vector graphics

Flash content can be created with vector art as well as images and Scaleform can seamlessly render both vector and bitmap graphics. However, each type has its advantages and disadvantages. The decision to use vector or bitmap graphics is not always clear, and often depends on several factors. This section discusses some of the differences between vector and bitmap graphics to help make content authoring decisions.

Vector graphics maintain their smooth shapes when scaled in size, unlike bitmaps images that can appear box-like, or pixelated, when scaled. But unlike bitmaps, vector graphics require more processing power to generate. Although simple solid-color shapes will usually be as fast as bitmaps, complex vector graphics with many triangles, shapes and fills can be expensive to render. Consequently, heavy use of vector shapes can sometimes reduce overall application performance.

Bitmap graphics can be a better choice for some applications because they don't require as much processing time to render as vectors, however, bitmap graphics significantly increase the amount of memory required, compared to vector graphics.

2.3.2 Vector Graphics

Vectors graphics are more compact than other image formats because vectors define the math (points, curves, and fills) required to render the image at runtime rather than the raw graphic (pixel) data of a bitmap. However, converting the vector data to the final image is time consuming and must be done whenever there is significant change in appearance or scale of a graphic. If the movie clip contains complex shape outlines that change every frame, animations may run slowly.

The following are several guidelines to help render vector graphics efficiently:
- Experiment with converting complex vector graphics to bitmaps and test how this affects performance.
- Keep the following in mind when using alpha blends.
  - Solid-filled strokes are cheaper to compute than alpha-blended strokes, since they can use a much more efficient algorithm.
  - Avoid using transparency (alpha). Flash must check all pixels underneath a transparent shape, which slows rendering down considerably. To hide a clip, set its `_visible` (AS2) or `visible` (AS3) property to `false` rather than setting the `_alpha` (AS2) or `alpha` (AS3) property to 0. Graphics may render faster when their `_alpha/alpha` property is set to 100. Setting the movie clip's timeline to an empty keyframe (so that the movie clip has no content

to show) is usually an even faster option. Sometimes Flash still tries to render invisible clips; move the clip off stage by setting its `_x` and `_y` properties (in AS2) or x and y properties (in AS3) to a position off the visible stage, in addition to setting the `_visible/visible` property to `false`, so Flash doesn't try to draw it at all.

- Optimize vector shapes.
  - In using vector graphics try to simplify the shapes as much as possible eliminating redundant points. This will reduce the amount of calculations that the player has to compute for each vector shape.
  - Use primitive vectors including circles, squares, and lines.
  - Flash's drawing performance is tied to how many points are drawn per frame. Optimize shapes with the Modify -> Shape submenu, then select either Smooth, Straighten or Optimize (depending on the graphic in question) to reduce the number of points required to draw it. This helps reduce the mesh data that is created by the Scaleform vector tessellation code.
- Corners are cheaper than curves.
  - Avoid complex vectors with too many curves and points.
  - Corners can be mathematically simpler to render than curves. When possible, stick with flat edges, especially with very small vector shapes. Curves can be simulated in this way.
- Use gradient fills and gradient strokes sparingly.
- Avoid shape outlines (strokes).
  - Whenever possible, do not use strokes with vector shapes, because doing so increases the number of rendered lines.
  - Outlines around vector images have a performance hit.
  - Whereas a fill has only an outside shape to render, outlines have an inside and an outside to render. This requires twice as much work to draw a line over a fill.
- Minimize the use of Flash's Drawing API. It may cause significant performance overhead if used unnecessarily. If needed, use the Drawing API to draw on a movie clip once. There are no performance penalties for rendering such a custom movie clip.
- Consider when to use mask. Though mask rendering has been optimized in Scaleform, it can be expensive to render them compared to a single shape. Note that in many cases the visual effect that artists use masks for does not require a mask. In particular, it is common to use a mask to cut a shape out of a bitmap. The same thing can be achieved much more efficiently by applying a bitmap fill to a shape directly in Flash Studio. This also provides the added benefit of Scaleform's patent-pending EdgeAA anti-aliasing, which is not available when using masked primitive.
- Convert multiple non-overlapping objects to have the same properties, whenever possible to avoid generating extra draw primitives (as discussed above).
- After a shape is created, it can be translated, rotated and blended with no additional memory use. However, bringing in new large shapes or doing significant scaling will consume more memory from tessellation.

- With EdgeAA enabled, shapes built out of multiple solid colors will render faster than those built out of multiple gradients/bitmaps. Caution is advised when connecting gradients/bitmaps within one shape, as this will cause the number of draw primitives to increase quickly.

2.3.3 Bitmaps

The first step in creating optimized and streamlined animations or graphics is to outline and plan your project before its creation. Specify a target for the file size, memory usage and length of the animations that you want to create, and test throughout the development process to ensure that you are on track.

In addition to draw primitives described earlier, a significant factor that affects rendering performance is the total surface area drawn. Every time a visible shape or bitmap is placed on the Stage, it needs to be rendered even if it is hidden by other overlapping shapes, consuming video card fill-rate. Although today's video cards are an order of magnitude faster than software Flash, large overlapping alpha-blended objects on screen can still greatly reduce performance, especially on lower-end and older hardware. For this reason it is important to flatten overlapping shapes and bitmaps, and explicitly hide obscured objects.

When hiding objects, it is best to set the `_visible` (in AS2) or `visible` (in AS3) property of a movie clip instance to `false` instead of changing the `_alpha` (in AS2) or alpha (in AS3) level to 0 in a SWF file. Although Scaleform will not draw objects with `_alpha/alpha` value of 0, their children may still incur CPU processing cost due to animation and ActionScript. If the instance visibility is set to false, then there is potential for CPU cycles and memory savings, which can give your SWF files smoother animations and provide better overall performance for your application. Instead of unloading and possibly reloading assets, set the `_visible/visible` property to `false`, which is much less processor-intensive. The following are several guidelines to help render bitmap graphics efficiently:

- Create all textures/bitmaps using a power of two for width and height. Examples of bitmap sizes include 16x32, 256x128, 1024x1024 and 512x32.
- Do not use compressed images not supported by the target hardware (for example JPEG images are compressed, but not supported in hardware by any platforms). This will require decompression time during file load.
- Use the gfxexport tool with texture compression switch(es) to create compressed images supported by your target platform's hardware (for example DXT compression is supported by many platforms). The final SWFs convert into GFX format to reduce bitmap memory requirements via texture compression. Compressed textures can provide huge savings in bitmap memory compared to uncompressed textures. Many compressed texture formats uses lossy compression (including DXT); therefore, make sure the quality of the resulting bitmaps is satisfactory. Use gfxexport's option `-qp` or `-qh` to get highest quality texture (Note that these options may take a long time to process the bitmap images).
- Use fewer bitmaps and/or bitmaps with smaller dimensions.

- If a bitmap is used to display a large simple shape, recreate the bitmap using vector graphics. This will result in higher quality with Edge AA and will save memory.
- Consider loading and unloading large bitmaps through ActionScript as needed.
- The size of the SWF/GFX file should not be used to judge its memory use. Even if a SWF file size is small, it can still use significant amounts of memory when loaded. For example, if a SWF file contains a 1024 x 1024 embedded JPEG image, 4 MB is used for the image at runtime, after the image has been decompressed.
- It is important to keep track of the number and size of image files being used in your UI. Count all the image sizes, add them up and multiply that by four (usually there are four bytes per pixel). Note that the gfxexport tool should be used with the compressed texture options (for example –i DDS) to reduce image memory. Use AMP to check memory consumption by bitmaps.

- Overall, in most cases bitmaps will be faster for any complex shapes; however, vectors will look better. The exact tradeoff will depend on your system's fill rate, transform shader performance and CPU speed. Ultimately, the only real way to know is to test performance on the target system.
- Create a master gradient.swf file containing only gradient textures and import it into other SWF files as needed. Use the gfxexport tool to export the gradients.swf using the `–d0` switch. This switch disables compression, and will apply to all textures in that SWF file. It will ensure that all textures in this file, which make use of gradients, are free of banding.
- Avoid bitmaps with alpha channel whenever possible.
- Optimize bitmaps in your image processing app, such as Adobe Photoshop®, not in Flash.
- Use PNGs if bitmap transparency is required and try to reduce the total surface area drawn.
- Avoid overlapping big bitmaps since this affects fill-rate performance.
- Import bitmap graphics at the size that they will be used in the application; don't import large graphics and scale them down in Flash, as this wastes file size and runtime memory.

2.3.4 Animation

When adding animation to an application, consider the frame rate of the FLA file. It can affect the performance of the final SWF file. Setting a frame rate too high can lead to performance problems, especially when many assets are used or AS is used to create animations that are ticked with the document's frame rate.

However, the frame rate setting also affects how smoothly the animation plays. For example, an animation set to 12 frames per second (FPS) plays 12 frames of the timeline each second. If the document's frame rate is set to 24 FPS, the animation appears to animate more smoothly than if it is set to 12 FPS. However, an animation at 24 FPS also plays twice as fast as it would at 12 FPS, so the total duration (in seconds) would be half the duration. Therefore, in order to make a 5-second animation using a higher frame rate, additional frames are required to fill those five seconds than at a lower frame rate, which raises the total file size.

Note: When using an `onEnterFrame` (in AS2) or `Event.ENTER_FRAME` (in AS3) event handler to create scripted animations, the animation runs at the document's frame rate, similar to creating a motion tween on the timeline. An alternative to the `onEnterFrame/Event.ENTER_FRAME` event handler is `setInterval`. Instead of depending on frame rate, functions are called at a specified interval of milliseconds. Like `onEnterFrame/EVENT.ENTER_FRAME`, the more frequently `setInterval` is used to call a function, the more resource intensive the animation will be.

Use the lowest possible frame rate that renders a smooth animation at runtime. This will help reduce the performance hit on the processor. Try not to use a frame rate that's more than 30 to 40 FPS; high frame rates beyond this point increase CPU cost and do not greatly improve the animation smoothness. In most cases, Flash UI can be safely set to half the target frame rate of the underlying game.

The following are several guidelines to help design and create efficient animations:

- The number of objects on the stage and how fast things move affect the overall performance.
- If there are a large amount of movie clips on the stage and they are required to switch on/off quickly, then `_visible/visible` = `true/false` should be used to control their visibility instead of attaching/removing the movie clips.
- Pay close attention to the use of tweens.
    - Avoid tweening too many items simultaneously. Reduce the number of tweens and/or sequence animation so that one begins when another ends.
    - Use timeline motion tween instead of the standard Flash Tween class when possible because it has much less performance overhead.
    - Scaleform recommends the use of CLIK Tween class (gfx.motion.Tween in AS2 or scaleform.clik.motion.Tween in AS3) instead of the standard Flash Tween class since it is smaller, faster and cleaner.
- Keep the framerate low, as the difference between high and low framerates are often not noticable. The higher the framerate, the smoother the animation, but the performance impact increases. A game running at 60 frames per second does not require a Flash file set to 60 FPS. The Flash framerate should be the minimum required to produce the necessary visual effects.
- Transparency and gradients are processor intensive tasks and should be used sparingly.
- Make the area of focus well designed and animated, then reduce animation and effects in other areas of the screen.
- Pause passive background animations (e.g., subtle background effects) during transitions.
- Test adding/removing animated elements to weigh their impact on performance.
- Use tween easing wisely. On slower hardware it can create an "appearance" of lag.
- Avoid shape morphing animations, such as transforming a circle into a square, as they are very CPU intensive operations. Shape tweens (morphing) have a very significant CPU hit because the shape is recomputed every frame; the cost of the hit will depend on the complexity of the shape (number of edges, curves and intersections). It may still be useable for some scenarios, but profile it to verify that

the cost is acceptable; the cost of a four-triangle tween may be acceptable. Essentially, there are performance/memory trade-offs to be aware of. Displaying the regular shape causes tessellation and caching of the shape so that it is displayed efficiently in future frames. With a morph, the trade-off is changed, since any change in the shape causes the old mesh to be released and a new one created.

- The most efficient animations are translation and rotation. It is best to avoid scaling animations, as they may cause retessellation (which can have a noticeable performance hit) and the resulting mesh may consume more memory.

2.3.5 Text and Fonts

- Text glyph font sizes should be smaller than the font cache manager SlotHeight or the size planned to use with gfxexport (the default is 48 pixels). If a larger font is used, then vectors will be used and consequently be much slower because of many resulting DPs (each vector glyph generates a DP).
- Turn off the border and background of a text field if possible, since this will save one draw primitive.
- Updating a text field's content every frame is one of the most common performance-sapping actions that can be easily avoided. Instead, change text field values only when their content truly changes or at the slowest rate possible. For example, when updating a timer that displays seconds, it is not necessary to update it at a frame rate of 30 FPS. Instead, record the old value and reassign the value of the text field only when the new value is different from the previous value.
- Do not use variables linked to a text field ("TextField.variable" property), since the text field will retrieve and compare the variable every frame, thus affecting performance.
- Minimize updating text by reassigning the "htmlText" property. Parsing HTML is a relatively expensive process.
- Use gfxexport with options `-fc`, `-fcl`, `-fcm` to compact fonts in order to save memory on glyph shapes (especially, if Asian fonts are embedded). See the "Font Overview" document for more details.
- Embed only the necessary symbols for fonts or use the fontlib mechanism if localization is necessary (again, please refer to the "Font Overview" document).
- Use the smallest number of TextField objects necessary, combining multiple items into one whenever possible. A single text field can typically be rendered with one DP even if it uses different colors and font styles.
- Avoid scaling text fields or using large font sizes; after a certain size the text field will switch to vector glyphs and each vector glyph becomes a draw primitive. If clipping is necessary (only part of a vector glyph is visible) then a mask will be used. Clipping of rasterized glyphs does not require a mask.
- Make sure the glyph cache size is large enough to hold all (or most) used glyphs. If the cache size is not enough there will be a serious performance hit due to frequent rerasterization of glyphs.
- Using text effects such as blur, drop shadow or knockout filters require extra space in the font cache, and also affects performance. Minimize the use of text filters if possible.

- Use DrawText API instead of creating separate movies when possible. The DrawText API allows the developer to programmatically draw text from C++ using the same Flash font and text systems used in a Scaleform-created user interface. Rendering of name billboards over moving avatars on screen or text labels next to items on a radar can often be done more efficiently through C++ if your game cannot afford having a Flash UI for those items. Refer to the "DrawText API Reference" document for further details.

3   ActionScript Optimizations

ActionScript is not compiled to native machine code; instead it is converted to bytecode, which is faster than an interpreted language but not as fast as compiled native code. Although AS can be slow, in most multimedia presentations, the assets such as graphics, audio, and video—and not the code—are often the limiting performance factor.

Many optimization techniques are not specific to AS but simply well-known techniques for writing code in any language without an optimizing compiler. For example, loops are faster if items that don't change with every loop iteration are removed from within the loop and placed outside the loop instead.

3.1   General ActionScript 2 Guidelines

The following optimizations will increase AS2 execution speed.

- The less AS used, the better the file performance. Always minimize the amount of code written to perform a given task. AS should be used primarily for interactivity, not for creating graphical elements. If your code consists of heavy use of `attachMovie` calls, reconsider how the FLA file is constructed.
- Keep the AS as simple as possible.
- Use scripted animation sparingly; timeline animation will typically perform better.
- Avoid heavy string manipulation.
- Avoid too many looping movie clips that use an "if" statement to avoid termination.
- Avoid using `on()` or `onClipEvent()` event handlers. Use `onEnterFrame`, `onPress`, etc instead.
- Minimize placing primary AS logic on a frame. Instead, place large chunks of important code inside functions. The Flash AS compiler can generate significantly faster code for source located within the function body as compared to that located directly inside of frames or old-style event handlers (`onClipEvent`, `on`). It is however acceptable to keep simple logic in the frame, such as timeline control (`gotoAndPlay`, `play`, `stop`, etc.) and other non-critical logic.
- Avoid using "long-distance" `gotoAndPlay/gotoAndStop` in movie clips with long timelines having several animations.
  - In case of forward `gotoAndPlay/gotoAndStop`, the farther the target frame from the current one, the more expensive is the `gotoAndPlay/gotoAndStop` timeline control. Thus, the most expensive forward `gotoAndPlay/gotoAndStop` is from the first frame to the last one.
  - In case of backward `gotoAndPlay/gotoAndStop`, farther the target frame from the beginning of the timeline, the more expensive is the timeline control. Thus the most

- expensive backward `gotoAndPlay/gotoAndStop` is from the last frame to the frame before the last.
- Use movieclips with short timelines. The cost of the `gotoAndPlay/gotoAndStop` highly depends on the number of keyframes and the complexity of timeline animation. Thus, do not create long and complex timelines if you are planning to navigate by calling `gotoAndPlay/gotoAndStop`. Instead, split the movieclip's timeline into several independent movieclips with shorter timelines and less `gotoAndPlay/gotoAndStop` calls.
- If updating many objects simultaneously, it is essential to develop a system in which these objects can be updated as a group. On the C++ side use a GFx::Movie::SetVariableArray call to pass large amount of data from C++ to AS. This call can then be followed by a single invoke that updates multiple objects at once based on the uploaded array. Grouping multiple invokes into one call is usually several times faster than calling them for each individual object.
- Don't attempt to perform too much work in a single frame, or Scaleform may not have time to render the Stage, and the user may perceive a slowdown. Instead, break up the amount of work being performed into smaller chunks, allowing Scaleform to refresh the Stage at the prescribed frame rate with no perceived slowdown.
- Don't overuse the Object type.
  - Data-type annotations should be precise, because it allows compiler type checking to identify bugs. Use the Object type only when there is no reasonable alternative.
- Avoid using the eval() function or array access operator. Often, setting the local reference once is preferable and more efficient.
- Assign `Array.length` to a variable before a loop to use as its condition, rather than using `myArr.length` itself. For example:
  Use the following code

```
var fontArr:Array = TextField.getFontList();
var arrayLen:Number = fontArr.length;
for (var i:Number = 0; i < arrayLen; i++) {
    trace(fontArr[i]);
}
```

  instead of:

```
var fontArr:Array = TextField.getFontList();
for (var i:Number = 0; i < fontArr.length; i++) {
    trace(fontArr[i]);
}
```

- Manage events wisely and specifically. Keep event listener arrays compacted by using conditions to check whether a listener exists (is not `null`) before calling it.
- Explicitly remove listeners from objects by calling `removeListener()` before releasing the reference to the object.

- Minimize the number of levels in package names to reduce startup time. At startup, the AS VM has to create the chain of objects, one object per level. Moreover, before the creation of each level object, the AS compiler adds an "if" conditional to check whether the level is already created or not. Thus, for the package `"com.xxx.yyy.aaa.bbb"` the VM will create objects `"com"`, `"xxx"`, `"yyy"`, `"aaa"`, `"bbb"` and before each creation there will be an "if" opcode that checks the object's existence. Accessing of such deeply nested objects/classes/functions is also slow, since resolving names requires parsing each level (resolve `"com"`, then `"xxx"`, then `"yyy"` inside the `"xxx"`, etc). To avoid this additional overhead, some Flash developers use preprocessor software to reduce the path to a single-level unique identifier, such as `c58923409876.functionName()`, before compiling the SWF.
- If an application consists of multiple SWF files that use the same AS classes, exclude those classes from select SWF files during compilation. This can help reduce runtime memory requirements.
- If AS on a keyframe in the timeline requires a lot of time to complete, consider splitting that code up to execute over multiple keyframes.
- Remove `trace()` statements from the code when publishing the final SWF file. To do this, select the Omit Trace Actions check box on the Flash tab in the Publish Settings dialog box, then comment them out or delete them. This is an efficient way to disable any trace statements used for debugging at runtime.
- Inheritance increases the number of method calls and uses more memory: a class that includes all the functionality it needs is more efficient at runtime than a class that inherits some of its functionality from a superclass. Therefore, it may be necessary to make a design trade-off between extensibility of classes and efficiency of code.
- When one SWF file loads another SWF file that contains a custom AS class (for example: `foo.bar.CustomClass`) and then unloads the SWF file, the class definition remains in memory. To save memory, explicitly delete any custom classes in unloaded SWF files. Use the `delete` statement and specify the fully qualified class name, as the following example shows:
  ```
  delete foo.bar.CustomCLass
  ```

- Not all code has to run every frame, use flip flops (where portions of code are alternated each frame) for non-100% time critical items.
- Try to use as few `onEnterFrames` as possible.
- Precalculate data tables instead of using math functions
  - If doing a lot of math, then consider precalculating the values and storing them in a (pseudo) array of variables. Pulling those values from a data table can be much faster than having Scaleform to do it on the fly.

3.1.1 Loops

- Focus on optimizing loops, and any repeating actions.

- Limit the number of loops used and the amount of code that each loop contains.
- Stop frame-based looping as soon as it is no longer needed.
- Avoid calling a function multiple times from within a loop.
    - It is better to include the contents of a small function inside the loop.

3.1.2 Functions

- Whenever possible, avoid deeply nested functions.
- Do not use `with` statements inside functions. This operator turns off optimizations.

3.1.3   Variables / Properties

- Avoid referencing nonexistent variables, objects, or functions.
- Use the "var" keyword whenever possible. The use of the "var" keyword inside functions is especially important since the ActionScript compiler optimizes access to local variables using internal registers with direct access by index rather than putting them into a hash-table and accessing by names.
- Don't use class variables or global variables when local variables will suffice.
- Limit the use of global variables, because they are not garbage collected if the movie clip that defined them was removed.
- Delete variables or set them to `null` when no longer needed. Doing this marks the data for garbage collection. Deleting variables helps optimize memory use during runtime, because unneeded assets are removed from the SWF file. It is better to delete variables than to set them to `null`.
- Always try to access properties directly rather than using AS getter and setter methods, which have more overhead than other method calls.

3.2 General ActionScript 3 Guidelines

The following optimizations will increase AS3 execution speed.

3.2.1 Strict Data Typing

- Always declare data type of a variable or a class member.
- Try to avoid declaring of variables and class members of type Object. Object is the most generic data type in AS3. Declaring a variable of type Object is the same as not declaring type at all.
- Try to avoid using of class Array. Use class Vector instead. Array is a sparse data structure. Unless you need to access/set value at index 4294967295 (or close) there is no need to use Array.
- Vector allows you to specify type of elements. Type Vector<*> means that you can store data of any type in an instance of this vector. Avoid using generic types like Vector<*>. Use specific types as elements of a Vector instead. For example Vector<int>, Vector<String>, or Vector<YourFavoriteClass> will use much less memory and will perform better than Vector<*>.
- Do not use Object as a hash table. Use class flash.utils.Dictionary instead.
- Avoid using of dynamic classes (and dynamic attributes). Access to dynamic attributes cannot be optimized at this time neither in Scaleform nor in Flash.
- Avoid using/changing of object's prototype. Prototype is a part of AS2. It is kept for compatibility in AS3, but in AS3 the same functionality can be achieved by using static functions and members.

3.3 Advance

If advance is taking too long to execute, there are seven possible optimizations:

1. Do not execute AS code on every frame. Avoid continuous `onEnterFrame/Event.ENTER_FRAME` handlers as well, which invoke code on each frame.
2. Use event-driven programming practices, changing text field and UI state values through an explicit Invoke notification only when a change takes place.
3. Stop animation in invisible movie clips (`_visible (AS2) or visible (AS3)` property should be set to `true`; use the `stop()` function to stop animation). This will exclude such movie clips from the Advance list. Note that it is necessary to stop **every** single movie clip in the hierarchy, including children, even if the parent movie clip has stopped.
4. There is an alternative technique that involves usage of `_global`.noInvisibleAdvance (AS2) or `scaleform.gfx.Extensions.noInvisibleAdvance (AS3)` extension. This extension might be useful to exclude groups of invisible movie clips from the Advance list without stopping each of them. If this extension property is set to "true" then invisible movie clips are not added into the Advance list (including their children) and therefore performance is improved. Keep in mind that

this technique is not fully Flash compatible. Ensure that the Flash file does not rely on any type of per frame processing within hidden movies. Don't forget to turn on Scaleform extensions on by setting `_global.gfxExtensions` (AS2) or scaleform.gfx.Extensions.enabled (AS3) to `true` to use this (and any other) extension.

5. Reduce the number of movie clips on stage. Limit unnecessary nesting of movie clips, as each nested clip incurs a small amount of overhead during advance.
6. Reduce timeline animation, number of keyframes, and shape tweens.

3.4 onEnterFrame and Event.ENTER_FRAME

Minimize the use of `onEnterFrame` (AS2) or `Event.ENTER_FRAME` (AS3) event handlers, or at the very least install and remove them when necessary, rather than having them executed all the time. Having too many `onEnterFrame/Event.ENTER_FRAME` handlers may drop performance significantly. As an alternative, consider using `setInterval` and `setTimeout` functions. When using `setInterval`:

- Do not forget to call `clearInterval` when the handler is no longer needed.
- `setInterval` and `setTimeout` handlers may be slower than `onEnterFrame/Event.ENTER_FRAME` if they are executed more frequently than `onEnterFrame/Event.ENTER_FRAME`. Use conservative values for the time intervals to avoid this.

3.4.1 Cleaning up onEnterFrame

To remove `onEnterFrame` handlers use the `delete` operator:

```
delete this.onEnterFrame;
delete mc.onEnterFrame;
```

Do not assign `null` or `undefined` to `onEnterFrame` (e.g., `this.onEnterFrame = null;`), as this operation does not remove the `onEnterFrame` handler completely. Scaleform will still attempt to resolve this handler, as the member with the name `onEnterFrame` will still exist.

3.5 ActionScript 2 Optimizations

3.5.1 onClipEvent and on Events

Avoid using `onClipEvent()` and `on()` events. Instead, use `onEnterFrame`, `onPress`, etc. There are several reasons for doing this:

- Function-style event handlers are run-time installable and removable.
- The byte-code inside functions is better optimized than inside old-style `onClipEvent` and on handlers. The main optimization is in precaching `this`, `_global`, `arguments`, `super`, etc., and using the 256 internal registers for local variables. This optimization works only for functions.

The only problem exists when you need an `onLoad` function-style handler to be installed before the first frame is executed. In this case, you may use the undocumented event handler `onClipEvent(construct)` to install the `onEnterFrame`:

```
onClipEvent(construct)
{
      this.onLoad = function()
      {
            //function body
      }
}
```

Or, use `onClipEvent(load)` and call a regular function from it. This approach is less effective however, as there will be additional overhead for the extra function call.

3.5.2 Var Keyword

Use the `var` keyword whenever possible. It is especially important to do so inside functions, as the AS compiler optimizes access to local variables by using internal registers with direct access by index rather than putting them into hash tables and accessing by names. Using the `var` keyword can double the AS function's execution speed.

Un-optimized Code:

```
var i = 1000;
countIt = function()
{
      num = 0;
      for(j=0; j<i; j++)
      {
            j++;
            num += Math.random();
      }
      displayNumber.text = num;
}
```

Optimized Code:

```
var i = 1000;
countIt = function()
{
      var num = 0;
      var ii = i;
      for(var j=0; j<ii; j++)
      {
            j++;
            num += Math.random();
            }
      displayNumber.text = num;
}
```

3.5.3 Precaching

Precache frequently accessed read-only object members in local variables (with `var` keyword).

Un-optimized Code:

```
function foo(var obj:Object)
{
      for (var i = 0; i < obj.size; i++)
      {
            obj.value[i] = obj.num1 * obj.num2;
      }
}
```

Optimized Code:

```
function foo(var obj:Object)
{
      var sz = obj.size;
      var n1 = obj.num1;
      var n2 = obj.num1;
      for (var i = 0; i < sz; i++)
      {
            obj.value[i] = n1*n2;
      }
}
```

Precaching can be used effectively for other scenarios as well. Some further examples include:

```
var floor = Math.floor
var ceil = Math.ceil
num = floor(x) - ceil(y);
```

```
var keyDown = Key.isDown;
var keyLeft = Key.LEFT;
if (keyDown(keyLeft))
{
    // do something;
}
```

3.5.4 Precache Long Paths

Avoid repeating usage of long paths, such as:

```
mc.ch1.hc3.djf3.jd9._x = 233;
mc.ch1.hc3.djf3._x = 455;
```

Precache parts of the file path in local variables:

```
var djf3 = mc.ch1.hc3.djf3;
djf3._x = 455;

var jd9 = djf3.jd9;
jd9._x = 223;
```

3.5.5   Complex Expressions

Avoid complex, C-style expressions, such as:

```
this[_global.mynames[queue]][_global.slots[i]].gosplash.text =
_global.MyStrings[queue];
```

Split this expression into smaller parts, storing intermediate data in local variables:

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
var slot_i = _global.slots[i];
_splqueue[slot_i].gosplash.text = _splstring;
```

This is especially important if there are multiple references on those split parts, for example in the following loop:

```
for(i=0; i<3; i++)
{
    this[_global.mynames[queue]][_global.slots[i]].gosplash.text =
```

```
                                        _global.MyStrings[queue];
    this[_global.mynames[queue]][_global.slots[i]].gosplash2.text =
                                        _global.MyStrings[queue];
    this[_global.mynames[queue]][_global.slots[i]].gosplash2.textColor =
                                        0x000000;
}
```

An improved version of the previous loop would be as follows:

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
for (var i=0; i<3; i++)
{
    var slot_i = _global.slots[i];
    _splqueue[slot_i].gosplash.text = _splstring;
    _splqueue[slot_i].gosplash2.text = splstring;
    _splqueue[slot_i].gosplash2.textColor = 0x000000;
}
```

The above code can be further optimized. Eliminate multiple references to the same element of array, if possible. Precache the resolved object in a local variable:

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
for (var i=0; i<3; i++)
{
    var slot_i = _global.slots[i];
    var elem = _splqueue[slot_i];
    elem.gosplash.text = _splstring;
    var gspl2 = elem.gosplash2;
    gspl2.text = splstring;
    gspl2.textColor = 0x000000;
}
```

4   HUD Development

The following section is a high-level introduction to Scaleform's recommended best practices when creating and iterating on a heads-up display (HUD). The implementations listed below are by no means required; however they should be reviewed for ideas and guidance that will help developers achieve better performance and optimal memory usage when creating a HUD with Scaleform.

Our recommendations are listed in an increasing order of complexity and length of time to implement. If creating multiple iterations of a HUD, we suggest starting with method 4.1 and moving deeper into the list as iterations toward a final version are created. This is a great way to rapidly prototype HUD elements that can operate in-game. As the HUD and resource requirements are refined, optimize using our recommended practices.

Please keep in mind that if developing an extremely complex, multilayered HUD with very high performance and very low memory requirements, development with C++ is still highly effective, and may be the best option.

4.1   Multiple SWF Movie Views

Overall, this type of HUD is very fast to both develop and iterate. It is purely artist driven and allows for more effects and better graphical representations in a very short period of time. This is optimal for prototyping and iterative design prior to optimization, but can increase memory use. Individual movie views create new player instances that add about 80K of memory overhead. The tradeoff is a faster HUD and individual movie control, but more memory usage. Take into consideration memory and performance issues vs. the flexibility of this dynamic, multilayered system.

Using multiple SWF movie views can offer benefits described below:

1. *Ability to Call Advance on different threads*. A multithreaded HUD interface allows for the execution, or Advance, of each Flash movie on different threads (not the rendering but the processing, e.g., timeline, animation, AS execution, flash processing). This enables individual advance control; breaking up Flash into multiple movies allows the developer to stop and not Advance certain elements, or call Advance at different times for different threads. Some elements of the HUD can Advance at a higher rate, or the developer can actually stop calling Advance on a static HUD element until an event happens in the game requiring an action.

   NOTE: Scaleform allows calling Advance on different threads for different GFx::Movie objects; however, since each movie instance is not thread-safe Display cannot be called on a different thread without explicit synchronization. Input and Invoke calls also need to be synchronized with Advance.

2. *Utilize render-to-texture caching*. This involves rendering HUD elements to textures and keeping them cached, only updating when necessary. This will minimize draw primitives, but will take up more memory, because it requires having a texture memory buffer that is the same size as the HUD element. It could have a positive performance impact, but a negative memory impact. Consider this option only if the element is rarely updated and fairly uncomplicated.

4.2   Single Movie View Containing Multiple SWFs

This method involves multiple Flash files loaded into a single full screen movie view via the AS SWF loading APIs. Benefits of this approach include more efficient memory use and slightly better Display performance.

We recommend the following guidelines when loading multiple Flash files into a single movie view.

1. Carefully group objects that can be hidden in batches and mark them with `_visible = false` (AS2) or visible = false (AS3) while having `_global.noInvisibleAdvance` (AS2) or `scaleform.gfx.Extensions.noInvisibleAdvance` (AS3) set to true to minimize advance processing overhead. **This is probably one of the most important things that can be done when creating a HUD**: grouping objects that can be hidden and adding a parent to manage them. Use `_visible/visible = false` to stop processing after the objects are invisible (NOTE: This is NOT to control visibility but to stop calling Advance on an object that is already not visible). By hiding certain groups of objects, execution logic will not be called on those elements inside of the Flash file. This is particularly useful where certain parts of the HUD are hidden and shown (e.g., pause menu, map, health bar). We also recommend that when hiding/showing the entire HUD, the developer should stop calling Advance altogether. Don't forget to turn on Scaleform extensions by setting _global.gfxExtensions (AS2) or scaleform.gfx.Extensions.enabled (AS3) to `true`.
2. Group multiple variable updates from the application through `SetVariableArray` and a single Invoke. This is useful if there is an element that has several different components, a high degree of complexity (e.g., a map with moving elements), and involves grouping updates into a single call vs. calling an individual invoke for every icon on the map. Call `SetVariableArray` to pass an array of data (e.g., new positions of each map element) and then call a single Invoke after that to process and use the array of data that is set to move the items all under a single function execution. However, if there is only a small amount of data to be updated, do not use this method as it could negatively impact performance.
3. Be judicious with the use of `onEnterFrame (AS2)` or `Event.ENTER_FRAME (AS3)` when creating HUD elements.
   If there are multiple elements inside the HUD that have `onEnterFrame/Event.ENTER_FRAME`, every time the developer calls Advance they will be executed, even if that particular element does not change.

4. Keep the animation frame rate of movies at half the game's frame rate and only call AS when needed. A common game programming paradigm is to call tick every frame in the game engine. This is definitely not optimal for using Flash. It is essential that the developer avoid calling AS Invokes every frame to reduce memory usage and increase performance. For example, if the game is running at 30–40 FPS, only update the animations every 15–20 FPS. However, consider that HUD animation maybe laggy, jittery and not as smooth if a very high framerate is used and there are animations that are called too slowly.
5. Shorten timeline animations as much as possible, as lengthy timeline animations tend to use more memory. However, this must be carefully managed, as too much shortening can cause jittery animations.

4.3 Single Movie View

Use this method to have very efficient Flash with a complex multi-element interface (e.g., radar screen). It is important to carefully examine how Flash vs. C++ is used to render elements. Multiple Flash layers cause separate draw primitives, reducing performance. Be sure to utilize all of the guidelines from section 4.2 in addition to the following:

1. Use the game engine to draw elements inside of an interface, but use Flash to draw a border and frame and Scaleform for text. When there are several rapidly changing elements within a HUD, Flash can be used to draw the static elements and C++ for rendering the items that are changing often.
2. Use C++ for positioning elements while still having them drawn by Flash. A good example of this would be a radar screen; create a single Flash set of dots, but manage their position inside the Render::Renderer2D, tagging the element using the RenderString identifier. Use the C++ engine to reposition the element correctly before it is rendered. This can avoid some of the AS update overhead, but it is fairly complex and requires extra programming.

4.4 Single Movie View (Advanced)

Using this method is significantly more advanced and time consuming, but can provide some additional memory savings. We do not recommend using this method until HUD iteration and creation is nearly complete. In addition to utilizing the methods detailed in section 4.3, consider the following techniques:

1. Only call Advance on graphic changes in the HUD, or try to do all updates through a single Invoke. This could be used on a single movie that has a background layer of HUD animation updating, with more complicated HUD interfaces (e.g., those containing text and progress bars) that sit on top of it that are not animating and not getting an Advance called. A good example of this would be a Health bar that typically isn't animated; there is no reason to call an Advance on it until there's a change. (e.g., play the first frame, freeze, and call the display and only Advance/invoke when there is a change to the

character's health).This is more efficient to render, with less CPU overhead, but it is a lot more complex to manage.

2. Utilize custom static buffer management for vertex data, and override the Render::Renderer. This method is C++ intensive, and would require heavy programmer intervention on the part of highly skilled C++ graphics programmers. In overriding the Render::Renderer, use different (custom) video memory vector data storage, and override other parts of the Scaleform system to use static buffers instead of dynamic buffers to manage HUD elements.

3. Use threaded rendering and override the Render::Renderer. This is probably the most complicated method; it requires rewriting the renderers and calling a separate Advance on a HUD that is rendered by the game engine. The tradeoff in this level of complexity is a potentially large performance gain.

   Note: Threaded rendering exists in the Scaleform–Unreal® Engine 3 integration, but it still would require a great deal of programming effort to implement this method.

4.5 Custom HUD Creation without Flash

This process is by far the most complex and time consuming. Ultimately, this HUD would be purely C++ and bitmap based. Scaleform and Flash could be utilized throughout the process of creating and iterating on your HUD, but then removed for the final version by converting Flash interfaces to bitmaps. At this point, Scaleform would not be doing any advance or taking memory for HUD elements, except for one occurrence of the Scaleform Player in memory.

Depending on what you can afford, you can combine custom-tuned C++ rendering for performance-critical HUD items with Scaleform rendering for everything else. Areas that can most benefit from external custom rendering are mini-maps and inventory/stat screens with many items; these are areas that can be optimized the most from DP batching and efficient multi-item updates, something that is difficult for Scaleform to do automatically. Other HUD elements such as borders, panels, stats and animated pop-ups can be left using Scaleform and replaced only if they become a bottleneck.

Regardless, we recommend that the developer continues to use the Scaleform font/text engine, particularly because Scaleform includes a DrawText API that allows the developer to programmatically draw text from C++ using the same Flash font and text systems used in a Scaleform-created user interface. This may save memory by not having to have two separate font systems: one for the HUD and a separate one for the rest of the menu system. For more information on our font/text engine, as well best practices for font and text usage please refer to the "Font & Text" section of our FAQ and "Font Overview"/"DrawText API Reference" documents.

Overall, please keep in mind the following considerations when creating and iterating on a HUD:

- Minimize the number of movie clips. Only nest items when absolutely necessary.
- Use masks sparingly mainly to avoid incurring performance cost. For example, to render the simplest masked primitive, three draw primitives are required (the mask clear, the mask shape, and the primitive itself). These costs scale with the number of masks, however, mask clears and mask shapes may be batched together, so it is generally slower than linear growth. Additionally, masks require that the render target contain either depth or stencil buffer. If an application does not have either of these buffers, allocating them solely for Scaleform masking may consume large amount of memory. For more information please refer to the "Graphics Rendering & Special Effects" section of our FAQ.
- Disable mouse and keyboard on PC and Wii™ (other consoles already disabled by default):
  ```
  GFx::Movie::SetMouseCursorCount(0);
  ```
  - If disabled, do not feed input.
- Group movie clip items that are shown/hidden together. Use `noInvisibleAdvance (AS2) or scaleform.gfx.Extensions.noInvisibleAdvance` (AS3) with `_visible(AS2) or visible (AS3)= false` for HUD panels.
- Call Invoke only when items change. Batch item changes into a single Invoke if two or more items are always changed together (same frame). Do not process items that are not updated often (per frame).
- Make sure to optimize usage of bitmaps and gradients. For more information please refer to section 2.1 or the "Art & Assets" section of our FAQ.

5   General Optimization Tips

5.1   The Flash Timeline

Timeline frames and layers are two important parts of the Flash authoring environment. These areas show where assets are placed and determine how your document works. How a timeline and the library are set up and used affect the entire FLA file and its overall usability and performance.

- Use frame-based loops sparingly. A frame-based animation is dependent on the application's framerate, as opposed to a time-based model which is not tied to FPS.
- Stop frame-based loops as soon as they are not needed.
- Distribute complex blocks of code across more than one frame if possible.
- Scripts with hundreds of lines of code will be just as processor intensive as timelines with hundred of frames based tweens.
- Evaluate content to determine whether animation/interaction can be easily achieved with the timeline or simplified and made modular using AS.
- Avoid using the default layer names (such as Layer 1, Layer 2), because it can be confusing to remember or locate assets when working on complex files.

5.2   General Performance Optimization

- There are ways to combine transforms for better performance. For example, instead of nesting three transforms, calculate one matrix manually.
- If things slow down over time, check for memory leaks. Be sure to dispose of things which are no longer needed.
- At authoring time, avoid a lot of `trace()` statements or updating text fields dynamically as these consume performance. Update them as infrequently as possible (i.e. only when something changes rather than constantly).
- If applicable, place layers that include AS and a layer for frame labels at the top of the layer stack in the timeline. For example, it is a good and common practice to name the layer that contains AS *actions.*
- Do not put frame actions in different layers; instead, concentrate all actions in one layer. This will simplify management of AS code and improve performance by eliminating the overhead incurred by multiple AS execution passes.

6 Rendering Optimizations

6.1 Using Bitmap Fill to Reduce Overdraw

This section will explain how to reduce overdraw using bitmap fills instead of bitmap images by themselves. Some devices overdraw can be more of a problem than triangle counts. This example will lead to a slight increase the number of triangles but reduces the amount of overdraw. However, since overdraw is much more expensive than rendering some extra triangles, this technique can lead to a significant improvement in rendering performance.
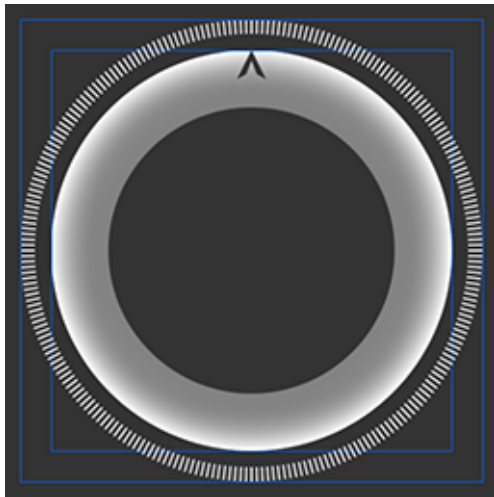
Figure 1: Inner and outer rings
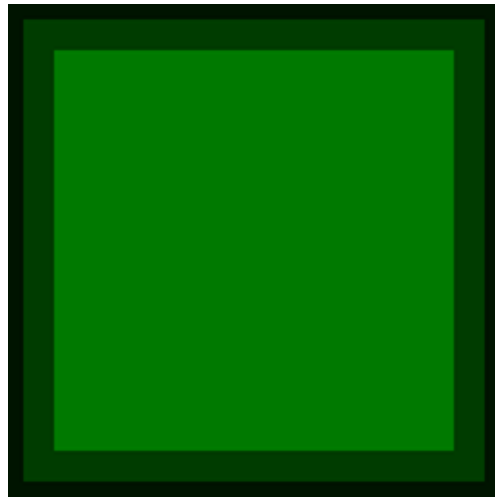
Figure 2:Overdraw view in Scaleform

Here we have two bitmap images, the inner ring and outer ring, one on top of the other. However, only the rings themselves are visible, not the transparent space in the corners and in the center. However those transparent pixels are still rendered and hence add to the rendering time. When rendered out using the overdraw view option [Ctrl+E] in the Scaleform Player you can see where overdraw is taken place, the brighter the green the more layers of overdraw.

The triangle count is low but there is a lot of overdraw. To reduce the amount of overdraw we will convert the images to shapes and using the bitmap as a fill within the shape. A quick easy way of doing this in Flash is to select the bitmap on the stage and hit [Ctrl+B], converting the image to a shape with the bitmap as a fill.
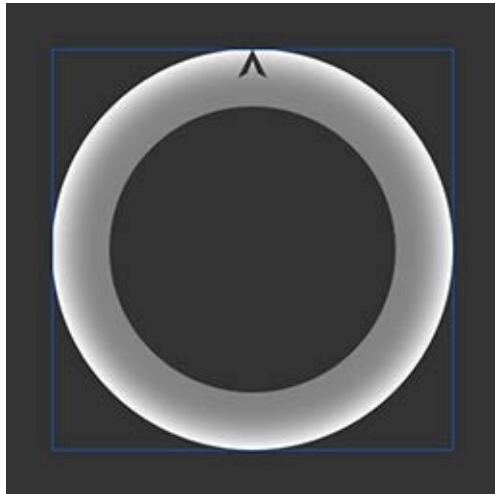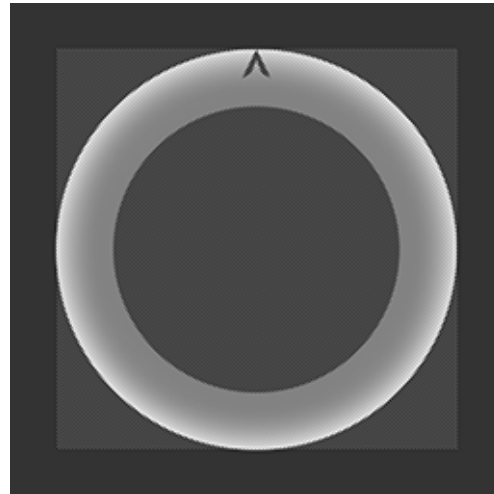
Figure 3: Bitmap

Figure 4: Shape with a bitmap fill

We then need to cut out the areas within the shape we don't intend to use, using either other shapes or lines. When cutting out the shape you increase the number of triangles, but this increase can be kept low by using a coarse shape (a shape that has a few corners) as opposed to a shape that has many corners.

In this example we will be using the PolyStar Tool because it allows us to easily cut out in a circular area with the least impact on triangle count. In the fill and stroke properties, we have the stroke set to a color and the fill set to transparent. In the tool settings options we change the number of sides to 8.
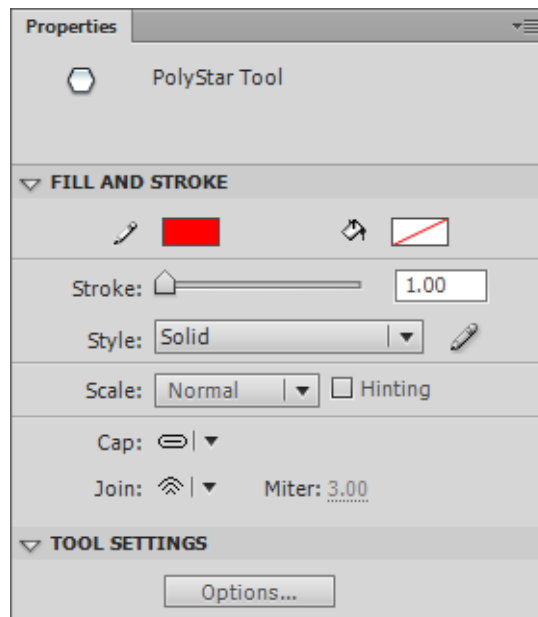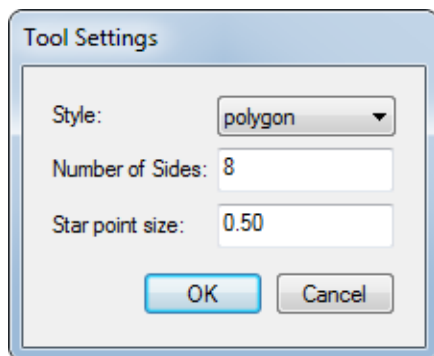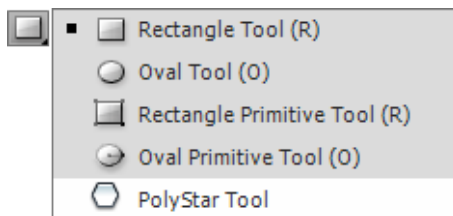
On the stage we draw out where we want to divide our shape. When finished we then delete the excess areas and the stroke leaving only the final shape.
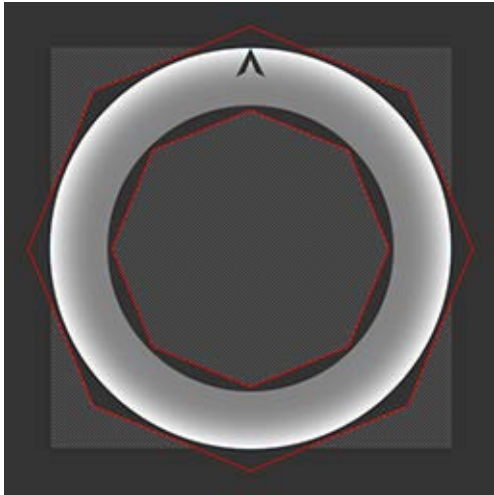

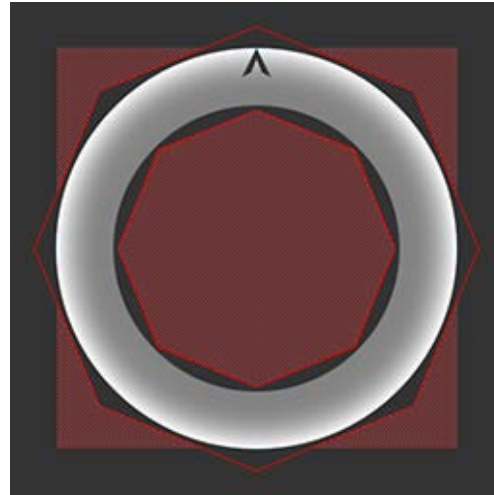**Figure 5: Stroke is used to divide the shape.**

Figure 6: Delete the excess areas and the stroke.

We then apply the same process to the other bitmap image and preview in the Scaleform player. The final result looks no different than the bitmap version until you turn on the overdraw preview mode. The shape is no longer a simple quad and therefore the triangle count is slightly higher but the overdraw area is significantly lower.
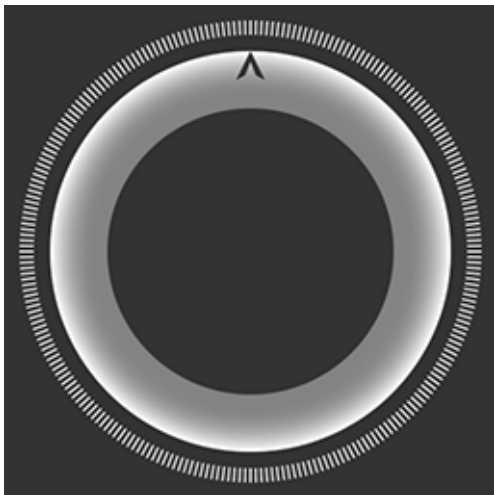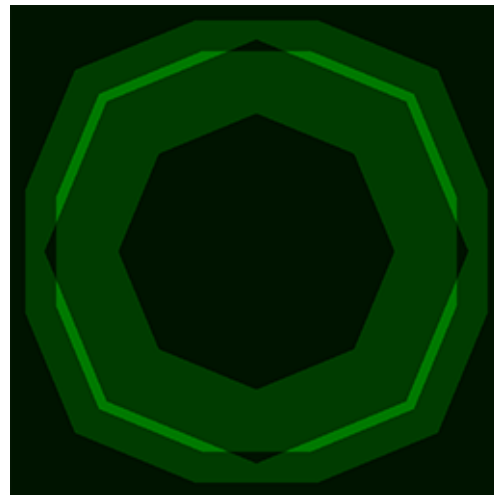

Figure 7: The final result.

Figure 8: The final result in overdraw view.

Using this technique on overlapping shapes can lead to significant rendering time improvement on older generation devices such as iPad1 where the graphics hardware handles overdraw poorly.

7 Additional Reading

ActionScript 2.0 Best Practices
http://www.adobe.com/devnet/flash/articles/as_bestpractices.html

Flash 8 Best Practices
http://www.adobe.com/devnet/flash/articles/flash8_bestpractices.html

Flash ActionScript 2.0 Learning Guide http://www.adobe.com/devnet/flash/articles/actionscript_guide.html