Autodesk®
Scaleform®

Scaleform 最佳優化指南

本文檔提供了使用 Scaleform 4.0 和更新的版本創建資源的最佳優化指南

作者： Matthew Doyle
版本： 3.0
最後修訂： 2011 年 4 月 25 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.4

Autodesk Scaleform 聯繫方式：

文檔	Scaleform 最佳實踐指南
地址	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
網站	www.scaleform.com

郵箱	info@scaleform.com
電話	(301) 446-3200
傳真	(301) 446-3199

目錄

1 概要

本文檔包括了使用 Autodesk Scaleform® 4.0 和更新的版本開發 Adobe® Flash®內容，並不斷完善的一個最佳優化條目清單。這些優化方法主要針對於改進遊戲中 Flash 運行的記憶體和性能；同時也可以用於其他的應用情況。文檔中的內容創建章節主要面向美工和設計師，而 ActionScript™ (AS)章節面向技術設計師和工程師。人物 (HUD) 開發章節爲 HUD 創建的開發場景的概要描述。我們鼓勵設計師和工程師都能夠在使用 Flash 和 Scaleform 開發 HUD 之前閱讀這些章節。

本文檔提供了不同來源的多種類型資訊彙編，如客戶支援請求、開發者論壇帖子、各種 web 上的 Flash 和 AS 資源。Scaleform 可以在多種不同的平臺上使用（從手機到先進的遊戲機和電腦）並集成了衆多不同的引擎，從而，不幸的是沒有一個"萬能"解決方案。每個遊戲和專案需要一個不同的解決方案以優化記憶體使用和性能（例如，從高端桌上電腦移植到手持終端時可能需要對代碼進行調整）。

我們儘量提供這些最佳優化方法的相關性能資料；然而，這些資料與你的應用專案中的結果不盡相同，因爲還需要考慮很多變數。我們建議在 UI 開發過程中所有的用戶介面（UI）應該進行徹底得測試，可選方案應該在不同的應用場景中提前進行著重測試。

Flash 設計師和開發者必須直觀的編寫代碼和構建應用結構，有利於自身以及工作在相同專案的其他成員，這在具有多個資源的 FLA 文件中尤爲重要。注意，Flash Studio CS5 引入一种基于 XML 的内容格式，取代二进制 .FLA 格式。这对于版本控制和更改列表跟踪非常有用。

通常單個 Flash 專案有多個設計師或開發者協同工作，當每個成員都按照統一標準來設定指導方針以使用 Flash、組織 FLA 文件以及編寫 AS 代碼。文檔的章節中略述了 AS 代碼編寫的最佳優化方法和使用 Flash 創作工具創建豐富媒體內容的最佳實踐。在任何時候、設計師或開發者、獨立工作或團隊協作都可以使用最佳優化方法。以下列出了一些學習和使用最佳優化方法而獲益的原因：

- 當工作在 Flash 或 AS 文檔：
    - 採用一致和有效的做法，有助於加速工作流程。使用已制定的編碼約定可以更快得進行開發，當進一步編輯時更益於理解和回憶文檔的組織結構。並且，代碼在一個大專案的架構中更易於移植和重用。
- 當共用 FLA 或 AS 文件：
    - 其他成員編輯的文檔可以快速查找和理解 AS 腳本代碼，從而修改代碼，查找和編輯資源。
- 當工作于應用軟體：

- 多個創作者可以共同工作于一個應用軟體之上，較少的衝突、更高的效率。專案或者現場管理員可以管理並構建複雜專案或應用，控制極少的衝突或冗餘。

- 當學習或教導 Flash 和 AS：
  - 學習如何使用最優方法創建應用並遵循編碼慣例以減少重復學習特殊方法的必要。如果學生不斷得學習 Flash 優化以及更佳的代碼結構方法，他們可以更快得學習編程語言，少走很多彎路。

開發者在閱讀這些最優方法時必定會發現更多並發展他們自身的好習慣。考慮到下列作爲與 Flash 協作時的方法指南；開發者可能選擇採用其中一些或全部建議。開發者也可以修改建議以適合他們自身工作。本章中的許多指導方針將幫助開發者利用一種一致的方法用在 Flash 和 AS 代碼編寫上面。

Flash 中的優化方法分類如下所示：

- 通過優化圖形提高動畫性能以快速重繪。
- 通過編寫代碼提高運算性能以加快執行速度。

如果代碼運行緩慢，意味著開發者需要減少使用範圍或者採用其他更加有效的方法解決問題。開發者應該確定並消除瓶頸，例如，優化圖形或通過使用 ActionScript 腳本代碼減少工作量。

通常性能表現可以被感覺到。如果開發者試圖在單個幀中執行過多工，Flash 沒有足夠時間來渲染場景，用戶可以感覺到速度降低。如果開發者將執行任務分成更小的任務塊，Flash 能夠在指定幀速率內刷新場景，不會感覺到速度降低。

1.1 常用用法

- 減少場景中物件數量。一次增加一個物件觀察何時以及多少性能將被降低。
- 避免使用標準 UI 元件（在 Flash 的元件面板中可以獲取）。這些元件設計爲運行在桌面電腦之上，並沒有進行優化以在 Scaleform 3.3 上運行。利用 Scaleform 通用精簡介面工具包(CLIK™)元件代替。

1.2 設置事項

- 減少物件層級深度。例如，如果有些物件不單獨移動或旋轉，則不需要將其轉換成組。
- 一個大的用戶介面通常最好打散爲單獨的 Flash 文件。例如，在一個 MMORPG 交易場所爲一個單獨的 Flash 介面，而 HUD 生命值爲另外一個單獨 Flash 介面。在 AS 端，不同文件可以通过 AS2 和

AS3 中的各种 SWF 加载 API 进行加载。。这些函數能夠在應用程式中在任何給定時間內導入介面所需內容並且在不需要時釋放這些內容。另外一個功能爲通過 C++ API 函數導入和釋放不同的 SWF 文件（創建不同的 GFx::Movie 實例）。

- Scaleform 3.3 支援多線程導入和播放。如果有額外的 CPU 運算內核，後臺導入多個 Flash 文件不會對性能産生負面影響。但是，開發者必須注意何時從 CD 或 DVD 導入多個文件，因爲光碟搜索速度緩慢可能會影響文件導入速度。
- 避免使用情景，情景通常爲較大的 SWF 文件。
- 在遊戲中的實際 UI 記憶體消耗量可能爲以下範圍：
    - 簡單遊戲 HUD 覆蓋：400K–1.5M
    - 動畫開始/功能表介面以及多個介面：800K–4M
    - 完全採用 ActionScript 腳本代碼並具有資源（Pacman）的簡單遊戲：700K–1.5M
    - 頻繁變化的向量動畫資源：2M–10M+

2 創建內容的記憶體和性能

當開發 Flash 內容時，很多考慮和優化需要遵循和實現以達到最佳性能。

2.1 繪圖原型

绘画单元 (DP) 是 GFx 创建的网格对象，用来将 Flash 元素渲染到显示屏，例如，一个形状。在支持批处理（或实例）渲染的平台上，GFx 尝试将具有兼容属性的相同和相邻层上的形状组合到单个 DP 中。每个 DP 单独渲染，因而招致性能成本。繪圖原型的數量可以通過 Scaleform Player HUD 來判斷，只需要點擊快捷鍵(F2)，將彈出 AMP HUD 摘要介面。介面顯示三角形計數、DP、記憶體使用和其他優化資訊。

以下爲一些可以幫助降低繪圖原型數量的方法：

1. 如 DP 只能包含具有相同属性的项目（具有不同纯色的形状除外）。例如，具有不同纹理或混合模式的项目永远不能合并到 DP 中。
2. 不同层的重叠对象不能合并到一个 DP 中，即使它们具有相同的属性。
3. 如果同时在一个形状中使用多个梯度填充，则这些梯度填充会增加 DP 的数量。
4. 具有纯色填充和没有笔画的矢量图形非常廉价。
5. 空的电影剪辑不需要 DP，不过，其中含有对象的电影剪辑则必须具有那些对象所确定的 DP 个数。

可以通过在 AMP 中启用该选项或按 (Ctrl+J) 来在 Scaleform 播放器中可视化批处理组和/或实例组。在此模式下，具有完全相同的颜色的项目以单个 DP 进行渲染。

2.2 動畫剪輯

1. 與其隱藏動畫剪輯，不如不再使用時從時間軸徹底刪除，否則，在前進播放時佔用處理時間。
2. 避免過多的動畫剪輯嵌套，因爲嵌套將影響性能。
3. 如果需要隱藏一個動畫剪輯，則使用`_visible=false` ，比使用`_alpha=0_alpha=0` (in AS2) and `visible=false` rather than `alpha=0` (in AS3).更有優勢。確保調用“stop()”函數停止隱藏動畫剪輯中的動畫。否則，隱藏的動畫仍然發生動作並影響性能。

2.3 藝術品

2.3.1 點陣圖與向量圖對比

Flash 內容能夠通過向量技術以及圖像進行創建，Scaleform 可以無縫渲染向量和點陣圖。但是，每種類型都有其優勢和劣勢。何時使用向量或點陣圖的判斷界限並不是十分清晰，通常由多個因素決定。本章將討論向量圖和點陣圖之間的差異以幫助創作者進行選擇。

向量圖在進行縮放時可以維持平滑的圖形，而點陣圖在縮放時按矩形縮放或者按圖元拉伸。與點陣圖不同，向量圖需要更多的運算處理。儘管簡單的純色圖形處理速度接近點陣圖，具有大量三角形、圖形和填充的複雜向量圖渲染開銷較大。
有些應用中最好使用點陣圖，因爲點陣圖不需要太多處理時間來渲染向量圖，但是，相對於向量圖點陣圖顯著增加記憶體消耗。

2.3.2 向量圖

向量圖比其他格式圖像更加簡潔，因爲向量圖使用數學運算（點、曲線、填充）在運行中即時渲染圖像而不是如點陣圖一樣採用圖像原型（圖元）。但是，將向量資料轉換成最終的現實圖像需要消耗時間，且在外觀發生變化或圖形縮放時必須重新執行。如果動畫剪輯包含複雜圖形且在每個幀都發生變化，動畫將運行緩慢。

以下爲幫助有效渲染向量圖的一些方法：

- 對負責向量圖轉換爲點陣圖進行試驗並測試性能影響。
- 在使用混合透明時注意一下幾點
    - 純色塊比混合透明塊需要更少運算開銷，因爲純色塊處理能夠使用更多有效演算法。
    - 避免使用透明化效果（alpha）。Flash 必須檢查透明化形狀之下的每個圖元，顯著降低了渲染速度。需要隱藏一個剪輯，設置_visible(AS2) or visible (AS3) 屬性爲 false，而不要將_alpha_alpha (AS2) or alpha (AS3) 屬性設置爲 0。在_alpha alpha property 屬性設置爲 100 時圖像渲染速度更快。設置動畫剪輯的時間軸到一個空的幀（這樣動畫剪輯無可顯示的內容）通常可以加快速度。有時 Flash 仍然試圖渲染隱藏的剪輯；將_x 和_y(in AS2) or x and y properties (in AS3)座標移出到場景的可視範圍也可以隱藏剪輯，也可以將 _visible /visible 屬性設置爲 false，Flash 不再繪製該圖像。
- 優化向量圖
    - 使用向量圖可以盡可能得簡化圖形消除冗餘點。這將減少計算每個向量圖的播放器計算量。
    - 使用向量原型包括曲線、矩形和直線。

    - Flash 繪圖性能與每個幀繪製的點數有關。通過 Modify -> Shape submenu 來優化圖形，然後選擇 Smooth、Straighten 或 Optimize（根據討論的圖像決定）以減少需要繪製的點的數量。這將減少 Scaleform 向量鑲嵌碼產生的網格資料。
- 拐角比曲線高效
    - 避免使用大量曲線和點的複雜向量。
    - 拐角渲染數學演算法比曲線簡單，盡可能得堅持用扁平邊緣，特別是非常小的向量圖形，曲線可以模仿該方法。
- 儘量少用梯度顏色填充和梯度塊
- 避免圖形輪廓描繪（打散）
    - 如果可以，任何時候都不要使用向量圖形塊，因爲這樣增加渲染線條數量。
    - 向量圖像周邊輪廓描繪影響性能。
    - 儘管填充只是一個外部形狀渲染，輪廓繪製需要進行內部和外部渲染。這比在填充上繪製直線多兩倍工作量。
- 減少 Flash 繪圖 API 函數的使用，如果不必要的使用將導致性能超標。如果需要，可以使用繪圖 API 函數繪製一次動畫剪輯。渲染這樣一個定制動畫剪輯將不會產生性能影響。
- 限制蒙板的使用。蒙板下面的圖元將仍然消耗渲染時間並對性能產生負面影響，甚至不進行任何繪製也是如此。多蒙板影響與使用的蒙板數相關聯。注意在很多情況下設計師需使用蒙板達到的可視效果未必需要使用蒙板才能實現。在有些情況，通常使用蒙板從點陣圖剪切一個圖形，直接在 Flash Studio 中在圖像上填充一個圖形可以實現同樣的效果。這爲 Scaleform 的 EdgeAA 反鋸齒效果提供了額外優勢。
- 盡可能得將多個物件轉換爲一個圖形以避免產生額外的繪圖原型。
- 在創建一個圖形後，可以進行轉換、旋轉和混合，不需要額外記憶體開銷。但是，引入新的較大圖形或進行明顯縮放將從鑲嵌方格中消耗更多記憶體。
- 使 EdgeAA 有效，純色圖形比多個色階/圖像渲染速度更快。在一個圖形中檔鏈結色階/圖像時建議需要慎重處理，因爲這將導致繪圖原型快速增加。

2.3.3 點陣圖

創建優化並改進的動畫或圖形第一步爲在創建前規劃工程，指定所要創建的物件的文件尺寸、記憶體使用和動畫長度的目標數，完整測試整個開發過程確保全程跟蹤。

除了前面描述的繪圖原型，還有一個影響渲染性能的重要因素爲整個表面區域的繪製。每次一個可視圖形或圖像放置到場景中，需要進行渲染，甚至在被其他圖形覆蓋時也是如此，消耗顯卡的填充率。儘管當前顯卡比 Flash 軟體速度高出一個數量級，螢幕上較大的覆蓋透明混合物件仍然能夠極大降低性能，特別是應用在低級終端和老的硬體當中。鑒於此原因，簡化重疊圖形和點陣圖將尤爲重要，並明確隱藏模糊或剪切物件。

當隱藏物件，最好在 SWF 文件中將動畫剪輯實例的_visible 屬性設置爲 false，替代將_alpha 值設爲 0。儘管 Scaleform 在_alpha 值爲 0 時不會繪製物件，它們的子物件動畫和 ActionScript 仍然消耗 CPU 處理開銷。如果將實例視覺化屬性設置爲 false，有可能可以減少 CPU 迴圈時間以及節省記憶體，可以使 SWF 文件更加平滑的動畫並爲應用程式提供更優的整體性能。代替釋放和重復導入資源，設置_visible 屬性設置爲 false，這可以大大減輕處理器負荷。以下爲幫助進行高效點陣圖渲染的一些指導方法：

- 創建所有紋理/點陣圖的寬度和高度值使用 2 的整數次方。例如點陣圖尺寸爲 16x32、 256x128、1024x1024 和 512x32。
- 不要使用目标硬件不支持的经过压缩的图像（例如，JPEG 图像经过压缩，但在硬件中不受任何平台支持）。，進行 JPEG 壓縮在導入文件時需要解壓縮時間。
- 使用具有纹理压缩开关的 gfxexport 工具来创建您的目标平台的硬件所支持的压缩图像（例如，DXT 压缩就受许多平台支持）。最终的 SWF 转换成为 GFX 格式，以通过纹理压缩来减少位图内存要求。与未压缩纹理相比，压缩纹理可提供巨大位图内存节约。许多压缩的纹理格式都使用有损压缩（包括 DXT），因此，要确保结果产生的位图的质量令人满意。。使用 gfxexport 中的選項–qp 或 –qh 以獲得最高等級的 DDS 紋理質量（注意這些選項可能需要很長時間來處理點陣圖圖像）。
- 儘量少用點陣圖和/或小尺寸點陣圖。
- 如果一個點陣圖用來顯示一個較大而簡單的圖形，使用向量圖重新創建該點陣圖。這將產生具有 Edge AA 的更高質量，且可以節省記憶體。
- 根據需要可以用 ActionScript 腳本導入和釋放較大點陣圖。
- SWF/GFX 文件尺寸不應該用來判斷其記憶體占用量。即使一个 SWF 文件很小，加载时它也会使用大量内存。例如，如果一个 SWF 文件包含一个 1024 x 1024 嵌入式 JPEG 图像，则图像解压后，运行时图像会使用 4 MB 内存。
- 追蹤在 UI 中使用的圖像文件數量和尺寸非常重要。計算所有的圖像尺寸，加起來然後乘以 4（通常每個圖元 需要四個位元組）。注意 gfxexport 工具應該設置爲–i DDS 選項以使用紋理壓縮減少圖像記憶體消耗。使用 AMP 檢查點陣圖記憶體消耗。
- 總之，在大多數情況下，點陣圖來描述任何圖形速度更快；但是，向量圖外觀效果更好。最終決定應該參考系統填充速率、轉換影音性能和 CPU 速率。最終，最實際的方法爲在目標系統測試性能。
- 創建一個 gradient.swf 主文件，只包含色階紋理並根據需要導入到 SWF 文件。使用 gfxexport 工具的-d0 開關導出 gradients.swf。該開關禁止壓縮，應用到所有的 SWF 文件中的紋理當中。需要確保所有的紋理在該文件中，利用色階，不受邊界限制。
- 盡可能避免點陣圖的透明通道。
- 在圖像處理應用程式中優化點陣圖，如在 Adobe Photoshop®，而不再 Flash 中。
- 如果點陣圖需要透明化處理則使用 PNG 圖像，視圖減少表面區域繪製數量。

- 避免較大點陣圖的重疊，那樣將影響填充性能。
- 導入的點陣圖圖像尺寸應符合應用大小；不要導入圖像然後在 Flash 中將其縮小，這樣浪費文件大小以及運行記憶體。

2.3.4 動畫

當添加動畫到一個應用程式中，考慮 FLA 文件的幀速率。將影響最終 SWF 文件的性能。設置幀速率過高將導致性能問題，特別是當使用很多資源或 AS 腳本代碼來創建動畫，因爲這些資源都與幀速率緊密相關。

然而，幀速率設置同樣也影響動畫播放的平滑度。例如，一個動畫設置爲每秒 12 幀（FPS），每秒在時間軸上播放 12 幀。如果文檔幀速率設置爲 24 FPS，則動畫將比 12 FPS 表現得更加平滑。但是，一個動畫在 24 FPS 幀速率下比 12 FPS 快兩倍，所以同樣數量幀的動畫持續時間（單位爲秒）爲後者一半。因此，爲了使 5 秒動畫使用更高幀速率，需要添加更多的幀，這樣增加了整個文件的大小。

注釋：當使用一個 onEnterFrame onEnterFrame (in AS2) or Event.ENTER_FRAME (in AS3 事件控制碼來創建腳本動畫時，動畫運行在文檔的幀速率，類似於在時間軸上創建一個運行動作。onEnterFrame/Event.ENTER_FRAME 事件控制碼可以用 setInterval 進行替換。函數調用由特定的時間間隔決定，以毫秒爲單位，不依賴於幀速率。如 onEnterFrame/Event.ENTER_FRAME，用 setInterval 調用函數越頻繁，動畫資源消耗越多。

盡可能使用最低的幀速率可以在運行時渲染一個平滑動畫。這將降低處理器性能的影響程度。儘量不要使用超過 30 到 40 FPS 的幀速率；超過這個範圍的幀速率將增加 CPU 開銷且不能明顯改進動畫平滑度。在多數情況下，Flash UI 可以安全得設置爲遊戲目標幀速率的一半。

以下爲有助於有效設計和創建動畫的一些指導方法：

- 場景中的物件數量和移動速度影響整體性能。
- 如果在場景中擁有大量動畫剪輯，且需要快速切換 on/off，則可以使用_visible/visible = true/false 設置來控制可視屬性，而不需要使用附加/刪除動畫剪輯操作。
- 密切關注 tween 動畫的使用
    - 避免過多條目同時使用 tween，減少 tween 數量並且/或者順序動畫，這樣一個動畫開始，另外一個動畫結束。
    - 盡可能使用時間軸動畫 tween 代替標準 Flash Tween 類因爲對性能影響較小。

- Scaleform 推薦使用 CLIK Tween 類（gfx.motion.Tween in AS2 or scaleform.clik.motion.Tween in AS3）代替標準 Flash Tween 類，因爲前者更小、更快、更簡潔。

- 保持較低幀速率，幀速率的不同往往不被注意到，幀速率越高，動畫越平滑，但是增加對性能的影響。一個遊戲運行在 60 幀每秒的幀速率不需要將 Flash 文件設置爲 60 FPS。Flash 幀速率應該設置爲滿足必要視覺效果的最低幀速率。
- 透明和色階消耗處理器資源應儘量少用。
- 精心設計焦點區域使其爲動畫，在螢幕中其他區域減少動畫和特效。
- 在轉換過程中終止背景動畫（例如，微妙的背景效果）。
- 測試添加/刪除動畫元素衡量性能影響。
- 巧妙使用 tween 動畫。在較慢的硬體上可以創建一個"外觀滯後"效果。
- 避免使用形變動畫，如將曲線變成矩形，因爲形變動畫需要消耗大量 CPU 資源。形變動畫（morphing）消耗大量 CPU 資源是因爲形變動畫在每幀都需要重新計算；開銷由圖形的複雜程度決定（邊緣數、曲線和交叉點）。在某些情景中能發揮作用，但是需要確保開銷在可以接受的範圍內；消耗爲四三角形動畫可以被接受。實質上，需要明白性能/記憶體的相互關係。顯示常規圖形導致鑲嵌格子緩存中保存圖形，因此能夠在將來的播放幀中有效得顯示出來。使用形變動畫，改變了其相互關係，因爲圖形的任何變化爲原來的柵格釋放新的柵格建立。
- 最有效的動畫爲轉換和旋轉。最好避免使用縮放動畫，因爲縮放動畫導致鑲嵌柵格（具有明顯的性能影響）產生的柵格佔用更多記憶體。

2.3.5 文本和字體

- 文本輪廓字體尺寸應該小於字體緩存管理器 SlotHeight 或者 gfxexport（默認爲 48 個圖元）計劃使用的尺寸。如果使用了一個更大的字體，則將使用向量從而產生很多 DP 降低速度（每個向量輪廓產生一個 DP）。
- 盡可能關閉文本區域邊框和背景，因爲這將節省一個繪圖原型。
- 每幀中更新一個文本域內容嚴重影響性能，這可以簡單得避免。改爲，只當其內容確實改變或者在盡可能低的幀速率下改變文本域的值。例如，當更新一個顯示時鐘的計時器，不需要在 30 FPS 下每幀都進行更新。記錄原來的值並在值發生改變時重新分配文本域的值。
- 不要使用鏈結到文本域的變數("TextField.variable"屬性)。由於文本域在每幀都要將重復訪問並比較變數，因此影響性能。
- 重新分配“htmlText”屬性減少更新文本。解析 HTML 開銷非常高昂。
- 使用 gfxexport 的`-fc`, `-fcl`, `-fcm` 選項來使字體更加緊湊節省字形輪廓佔用的記憶體（特別在內嵌亞洲字體中）。見“字體概述”文檔獲得更多詳細資訊。

- 只插入需要的字形或者使用字體庫機制用於本地化（也可以見“字體概述”文檔獲得更多詳細資訊）
- 使用最少數量必須的 TextField 物件，盡可能將多個條目合併成一個。單個文本域通常可以用一個 DP 進行渲染，甚至在使用不同顏色和字體時也是如此。
- 避免縮放文本域或使用較大字體；在超過一定尺寸的大小後，文本域將切換到向量字形每個向量字形需要一個繪圖原型。如果需要剪切塊（只有部分向量淪落可視）則需要用到蒙板。蒙板速度緩慢且需要額外的繪圖原型，字體裁剪的光柵不需要蒙板。
- 確保字形緩存有足夠大空間存放所有（或多數）使用的字形。如果緩存尺寸不夠，則某些字形可能會消失，或者頻繁柵格化字形嚴重影響性能。
- 使用文本特效如模糊、陰影或濾鏡需要使用更多字體緩存，同時也影響性能。盡可能少使用文本濾鏡。
- 盡可能使用 DrawText API 函數代替單獨動畫。DrawText API 允許開發者通過編程方式用 C++語言繪製文本，使用與 Flash 字體和 Scaleform 用戶介面中相同的文本。在螢幕移動物體上渲染指示牌名稱或者爲一個雷達上的條目標簽，如果遊戲不能使用 Flash UI 處理，則用 C++將更加有效。更多詳細資訊請見文檔“DrawText API 參考”。

3 ActionScript 優化

ActionScript 不編譯成本地機器碼；而是轉換成位元組碼，比解釋語言運行速度要快但是沒有編譯的本地代碼速度快。儘管 AS 腳本代碼速度較慢，但是在大多數多媒體展現中，資源如圖像、聲音、視頻 – 而不是代碼 – 作爲性能的瓶頸。

很多優化技術並不是專門針對 AS 腳本代碼，只是衆所周知的技術可以用在無優化編輯器且用任何語言編寫的代碼當中。例如，在迴圈中的條目在每次迴圈不發生改變時放到循環體的外部，可以使迴圈運行得更快。

3.1 通用 ActionsScript2 指南

以下優化可以增加 AS 執行速度。

- AS 腳本代碼使用越少，文件性能越高。完成既定任務盡可能少的使用代碼。AS 主要用戶交互，而適合用來創建一個圖形元素。如果你的代碼包含了大量 `attachMovie` 調用，重新考慮 FLA 文件如何構造。
- 保持 AS 儘量簡潔。
- 儘量少用腳本動畫；時間軸動畫通常具有更高性能。
- 避免使用大量字串。
- 避免過多迴圈動畫剪輯使用一個“if”語句以避免終止。
- 避免使用 `on()` 或 `onClipEvent()`事件控制碼，使用 `onEnterFrame`、`onPress` 來代替。
- 在幀上最小化主要的 AS 邏輯代碼，而在函數內部放置較大代碼塊。用 Flash AS 編譯器編譯生產的代碼相對於在幀上直接運行或用老式控制碼(`onClipEvent, on`)運行，速度更加快。但是在幀上使用簡單邏輯代碼也是可以接受的，如時間軸控制(`gotoAndPlay`、`play`、`stop` 等)以及其他非關鍵邏輯。
- 避免在包含多個動畫的長時間軸電影剪接中使用“遠距”`gotoAndPlay/gotoAndStop`。
    - 如果是前向 `gotoAndPlay/gotoAndStop`,目標幀距離當前幀越遠,`gotoAndPlay/gotoAndStop` 時間軸控制項就越貴。因而,最貴的前向 `gotoAndPlay/gotoAndStop` 是從第一個幀到最後一個幀的部分。

- 如果是後向 gotoAndPlay/gotoAndStop,目標幀距離時間軸的開頭越遠,時間軸控制項就越貴。因而,最貴的後向 gotoAndPlay/gotoAndStop 是從最後一個幀到倒數第二個幀的部分。

- 使用短時間軸電影剪輯。gotoAndPlay/gotoAndStop 的成本在很大程度上取決於關鍵幀的數量和時間軸動畫的複雜程度。因此,如果計畫通過調用 gotoAndPlay/gotoAndStop 進行導航,就不要創建複雜的長時間軸。或者,將電影剪輯時間軸拆分成具有較短時間軸和較少 gotoAndPlay/gotoAndStop 調用的若干個獨立的電影剪輯。
- 如果同時更新多個物件，則需要開發一個系統其中物件可以按組更新。在用 C++時可以使用 GFx::Movie::SetVariableArray 調用從 C++傳遞大量資料到 AS。該調用基於上載陣列可以通過單個調用更新多個物件。將多個調用組合成組通常比單獨調用速度要快好幾倍。
- 不要在一個幀內試圖執行太多工，Scaleform 可能沒有足夠時間渲染場景，用戶會感到速度減慢。而需將大量任務分散爲小的塊，使 Scaleform 在給定的幀速率內進行刷新不至於感到速度緩慢。
- 不要過度使用物件類型
  - 資料類型注釋須精確，因爲這使編譯器類型檢查定位 bug，只有在沒有其他方法時才使用物件類型描述。
- 避免使用 eval()函數或陣列訪問操作，通常，設置一次本地索引將更加可取且有效。
- 在迴圈使用前將 Array.length 分配給一個變數作爲迴圈條件，而不是用 myArr.length，例如：

  使用下列代碼

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

  代替：

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

- 精確管理實踐。保持事件監聽器陣列緊湊，使用條件判斷調用前監聽器是否存在（不爲 null）。
- 在釋放物件索引之前須調用 removeListener()從物件刪除監聽器。
- 最大限度地減小包名的級數以便於縮短啟動時間。啟動時,AS VM 必須創建物件鏈,每級一個物件。而且,在創建每級物件之前,AS 編譯器添加一個 "if" 條件句,以檢查是否已經創建那級物件。因此,對於包 "com.xxx.yyy.aaa.bbb",VM 將創建物件 "com"、"xxx"、"yyy"、"aaa"、"bbb",並且在每次創建之前,將會有一個檢查物件是否存在的 "if" 運算代碼。訪問此類深度嵌套的物件/類/函數

的速度也很慢,因為解析名稱時需要分析每個級(解析 "com",然後解析 "xxx",然後解析 "xxx" 內的 "yyy",以此類推)。為了避免此額外開銷,有些 Flash 開發人員使用預處理軟體在編譯 SWF 之前縮短到達單級唯一識別碼(如 `c58923409876.functionName()`)的路徑。

- 如果一個應用程式包括多個 SWF 文件且使用相同的 AS 類，除了那些編譯時選自 SWF 文件的類，這可以幫助減少運行記憶體需求。
- 如果時間軸上的關鍵幀中的 AS 腳本代碼需要很長執行時間，可以考慮將代碼分割到多個關鍵幀中去。
- 當發佈最終 SWF 文件時，從代碼中刪除 `trace()`語句。完成此操作，在 Publish Settings 對話方塊中的 Flash 標簽中選擇 Omit Trace Actions 核取方塊，然後加以注釋或直接刪除。這是用來禁止即時調試跟蹤語句的有效方法。
- 繼承增加了方法調用的數量和記憶體的使用：一個類包括了所有需要的函數執行起來比從上級類繼承功能的類執行起來更加高效。因此，在類的擴展和代碼效率上需要有一個設計權衡。
- 當一個 SWF 文件導入另外一個包含一個自定義 AS 類（例如：`foo.bar.CustomClass`）然後釋放 SWF 文件，該類的定義仍然在記憶體中。爲了節省記憶體，在釋放 SWF 文件時需要刪除自定義類。使用 `delete` 語句並制定類的完整名稱，如下例所示：

```
delete foo.bar.CustomCLass
```

- 不是所有代碼都在每個幀中都需要運行，對於非 100%有嚴格時間要求的項可以使用動態調度方法（每個幀中部分代碼爲可選）。
- 試圖儘量少用 `onEnterFrames`。
- 預先計算資料表而不使用資料函數
    - 如果處理過多數學運算，可以將值預先計算出來並保存到一個變數資料（假設）中。從資料表獲取這些值比 `Scaleform` 來處理更加有效。

3.1.1 迴圈

- 重點優化迴圈中的任何重復動作。
- 限制使用的迴圈數量和每個迴圈包含的代碼量。
- 在不再需要時立即停止基於幀的迴圈。
- 避免多次調用一個迴圈內部的函數。
    - 最好在循環體內包含少量函數。

3.1.2 函數

- 盡可能避免函數深沈嵌套。
- 不要在函數內使用 `with` 語句。該操作將關閉優化功能。

3.1.3 變數/屬性

- 避免參考不存在的變數、物件或函數。
- 盡可能使用“var”關鍵字。在函數內部使用“var”關鍵字特別重要，因爲 ActionScript 編輯器優化了使用內部寄存器對本地邊路按的訪問，直接使用訪問索引而不將資料放到哈希表中根據名字查找。
- 在本地變數足夠情況下不要使用類變數或總體變數。
- 限制總體變數使用，因爲在動畫剪輯定義然後刪除時不會進行垃圾回收。
- 刪除變數或在不需要後設置爲 `null`。這個操作用於垃圾回收，刪除變數可以優化運行時記憶體使用，因爲不需要的資源從 SWF 文件按中刪除。最好是直接刪除變數，比設置爲 `null` 更加有效。
- 總是儘量直接訪問屬性，比使用 AS 獲取和設置有效，後者需要更多開銷。

3.2 一般 ActionScript 3 指导原则

下面的优化将会提高 AS3 执行速度。

3.2.1 严格数据类型确定

- 始终声明变量或类成员的数据类型。
- 努力避免声明对象 (Object) 类型的变量和类成员。对象是 AS3 中的最普通的数据类型。声明“对象”类型的变量与根本不声明类型相同。
- 努力避免使用数组 (Array) 类。而要使用矢量 (Vector) 类。数组是一种罕见的数据结构。除非需要在索引 4294967295（或接近）位置访问/设置值，否则不需要使用数组。
- 矢量允许您指定元素类型。类型 Vector<*> 意味着您可以在此矢量的实例中存储任何类型的数据。避免使用像 Vector<*> 这样的通用类型。而是将具体类型作为某个矢量的元素。例如，Vector<int>、Vector<String> 或 Vector<YourFavoriteClass> 占用的内存小得多，而且性能比 Vector<*> 好。
- 不要把对象用作散列表。而是使用类 flash.utils.Dictionary。
- 避免使用动态类（和动态属性）。此时在 GFx 或 Flash 中都不能优化对动态属性的访问。
- 避免使用/更改对象的原型。原型是 AS2 的组成部分。保留它的目的是保持在 AS3 中的兼容性，但在 AS3 中，同样的功能可以通过使用静态函数和成员来实现。

3.3 前進播放

如果前進播放執行時間太長，有以下六種優化方法：

1. 不要在每個幀都執行 AS 代碼，避免 `onEnterFrame/ Event.ENTER_FRAME` 控制碼調用，該控制碼在每個幀都需要調用代碼。
2. 使用時間驅動編程條例，在發生改變時，通過外部調用改變文本域和 UI 狀態值。
3. 在隱藏動畫剪輯中停止動畫(`_visible(AS2) or visible (AS3)` 屬性應該設置爲 `true`；使用 `stop()` 函數停止動畫)。這樣可以從 Advance 列表中排除這樣的動畫剪輯。注意必須停止每個層級的單獨動畫剪輯，包括其子黨員，甚至在上級黨員動畫剪輯已經停止的情況下也是如此。
4. 一種可供選擇的做法爲`_global.noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)`擴展的使用。該擴展函數可以從 Advance 列表中排除隱藏視頻剪輯分組，而不需要將其停止。如果擴展屬性設置爲“`true`”則隱藏視頻剪輯不添加到 Advance 列表（包括其子物件）從而提高性能。注意這項技術不完全與 Flash 相容。確保 Flash 文件不依賴於任何類型的隱藏動畫幀處理。不要忘記通過使用該擴展函數（或其他函數）設置`_global.gfxExtensions` 爲 `true` 以打開 Scaleform 擴展屬性。
5. 減少場景中的動畫剪輯數量。限制不需要的動畫剪輯嵌套，因爲每個嵌套剪輯在前進中都需要一定數量的開銷。
6. 減少時間軸動畫、關鍵幀數量和形變動畫。

3.4 onEnterFrame and Event.ENTER_FRAME

儘量少用 `onEnterFrame`(AS2) or `Event.ENTER_FRAME` (AS3)事件控制碼，或採用最小安裝在不需要時及時刪除，而不要在所有時間都執行。具有大量 `onEnterFrame/Event.ENTER_FRAME` 控制碼將明顯降低性能。作爲補充，可以使用 `setInterval` 和 `setTimeout` 函數。當使用 `setInterval` 時：

- 不要在控制碼不再需要時忘記調用 `clearInterval`。
- `setInterval` 和 `setTimeout` 控制碼在比 `onEnterFrame/Event.ENTER_FRAME` 控制碼執行頻繁時可能比 `onEnterFrame/Event.ENTER_FRAME` 控制碼要慢。使用常數設定時間間隔來避免。

3. 4. 1 清理 onEnterFrame

使用 `delete` 操作刪除 `onEnterFrame` 控制碼：

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

不要分配 null 或 undefined 到 onEnterFrame (例如 this.onEnterFrame = null;)，因爲這個操作不能將 onEnterFrame 控制碼完全刪除。Scaleform 將試圖解決這個控制碼，因爲名爲 onEnterFrame 的成員函數仍然存在。

3.5 ActionScript 2 Optimizations

3.5.1 onClipEvent 和 on Events

避免使用 onClipEvent()和 on()事件，而使用 onEnterFrame、 onPress 等。這樣做的原因如下：

- 功能型事件控制碼可在運行時安裝和刪除。
- 函數中的位元組碼比老式 onClipEvent 事件以及控制碼具有更多優化屬性。主要的優化爲預先保存在緩存中的 this、 _global、arguments、super 等。並使用 256 個內部寄存器存放本地變數。該優化屬性只用於函數。

在第一幀執行前需要安裝 onLoad 函數類型控制碼存在的唯一問題，可以使用無正式文檔支援的事件控制碼 onClipEvent(construct)來安裝 onEnterFrame：

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

或者，使用 onClipEvent(load)並從中調用一個常規函數。該方法效率較低，因爲將導致額外的函數調用開銷。

3.5.2 Var 關鍵字

盡可能的使用 var 關鍵字。在函數內部該操作尤爲重要，因爲 AS 編譯器使用內部寄存器優化本地變數訪問，直接通過索引而不將資料放到哈希表中查找。使用 var 關鍵字可以使 AS 函數執行速度加倍。

未經優化代碼：

```
var i = 1000;
countIt = function()
{
      num = 0;
```

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

優化代碼：

```
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 預存

將頻繁訪問的唯讀物件成員預存到本地變數（使用 `var` 關鍵字）。

未經優化代碼：

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

優化代碼：

```
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;
      }
}
```

預存可以用於其他情景更加有效，一些例子包括：

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

```
var keyDown = Key.isDown;
var keyLeft = Key.LEFT;
if (keyDown(keyLeft))
{
      // 執行操作
}
```

3.5.4 預存長路徑

避免重復使用長路徑，如：

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

將文件路徑預存到本地變數：

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

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

3.5.5 複雜運算式

避免複雜、類 C 運算式，如下所示：

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

將這些運算式分割成更小單元，將交互資料保存在本地變數：

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

如果在分割單元中有多個索引這個操作就特別重要，例如如下迴圈：

```
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;
}
```

之前迴圈的改進版本如下所示：

```
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;
}
```

以上代碼還能夠進一步優化，盡可能取消指向相同陣列元素的多個索引。將本地變數中的既定變數預先存放在緩存中：

```
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 開發

本章概要介紹 Scaleform 推薦的創作和人物顯示(HUD)的最佳優化方法。下面列出的執行絕不是必須的；但是可以作爲指導可以幫助開發者用 Scaleform 創作 HUD 時實現更好的性能和記憶體優化。

我們的建議按照複雜性和執行時間長度列出在遞增列表中。如果多次創建一個 HUD，我們建議使用 4.1 小節中的方法並深入到列表反復直到創建最終版本。這是一種快速原型化可以在遊戲中操作的 HUD 元素的方法。使用我們的推薦優化方法對 HUD 和資源進行提煉和優化。

請牢記如果開發一個極其複雜、具有高性能多個層的 HUD 並且需要很少記憶體的專案，使用 C++將非常高效，也許是最佳選擇。

4.1 多 SWF 動畫視圖

總之，該類型的 HUD 對於開發和複用都很快捷。完全是藝術驅動並且在很短的周期內實現更好的效果以及更佳的圖形描繪，這在優化之間進行原型化和重用設計是最佳選擇，但是將增加記憶體使用。單個動畫試圖創建新的播放器實例增加大概 80K 的記憶體空間。需要在更快的 HUD 和單獨的動畫控制之間進行權衡，單獨的動畫控制將使用更多記憶體。將記憶體和性能因素與該動態、多個圖層系統的易用性對比進行考慮。

使用多個 SWF 動畫試圖能夠提供以下優勢：

1. *可以在不同的線程調用 Advance*：一個多線程 HUD 介面允許在不同線程上執行或播放每個 Flash 動畫（不是渲染而是整個播放過程，例如，時間軸、動畫、AS 執行、Flash 處理）。這使得可以進行獨立的前進播放控制；將 Flash 打散成多個動畫可以使開發者控制特定元素的停止和前景，或者在不同時間用不同線程調用 Advance。HUD 的某些元素能夠以更高的速率進行播放，或者開發者能夠實際上在一個靜態 HUD 元素上停止調用 Advance，直到遊戲中發生一個事件需要執行一個動作爲止。
   注釋：Scaleform 允許在不同線程上調用不同 GFx::Movie 物件的 Advance；但是，由於每個動畫實例不是線程安全顯示，不能在沒有同步的情況下調用不同線程。在使用 Advance 時同步還需要進行輸入和調用。
2. *利用紋理渲染緩存*：調用渲染 HUD 元素和紋理並保存在緩存中，只在需要時進行更新。這將最少使用繪圖原型，但是將佔用更多記憶體，因爲需要一個紋理緩存需要和 HUD 元素一樣的記憶體空

間。可以提高性能，增加記憶體使用量。只在元素極少更新並且非常簡單的情況下可以考慮使用該方法。

4.2 單個動畫試圖包含多個 SWF

本方法包含了多個 Flash 文件通過 AS `loadMovie` 命令導入到單個全屏動畫視圖中。該方法的優勢包括更高效的記憶體使用和略微顯示性能的提升。

我們推薦在導入多個 Flash 文件到單個動畫試圖時遵循以下事項：

1. 仔細將物件分組可以分批隱藏並將標記設爲`_visible = false`(AS2) or visible = false (AS3)，同時將`_global.noInvisibleAdvance`(AS2) or `scaleform.gfx.Extensions.noInvisibleAdvance` (AS3)設置爲 `true` 以減少前進播放處理開銷。**以下可能爲創建 HUD 時可以進行的最重要的操作：**將可以分批隱藏的物件分組並添加一個主幹來進行管理。使用`_visible/visible = false` 在物件隱藏時停止處理（注意：這不是用來控制可視狀態而是在物件已經隱藏的情況下停止 Advance 調用）。通過隱藏特定物件的組，在那些 Flash 文件內部的元素不會調用執行邏輯。這在部分 HUD 需要隱藏和顯示的地方特別有用（例如，暫停功能表、地圖、身體狀況欄）。我們也建議當需要隱藏/顯示整個 HUD 時，開發者應該完全停止那個 Advance。不要忘記將_global.gfxExtensions(AS2) or scaleform.gfx.Extensions.enabled (AS3)設置爲 `true` 開啓 Scaleform 擴展屬性。
2. 通過 `SetVariableArray` 和一個調用將應用程式中的多個變數分組更新。這在一個元素擁有多個不同元件時可以起到作用，一個高度複雜（例如，具有移動元素的地圖），將更新分組採用單個調用對比地圖上的每個圖示進行單獨調用。調用 `SetVariableArray` 來傳遞一個陣列資料（例如，每個地圖元素的新位置）然後單獨調用以處理並使用陣列資料，將條目都集中到單個函數執行。但是，若只有少量資料需要更新，不要使用這種方法，因爲這樣可能需要降低性能。
3. 當在創建 HUD 元素是明智得使用 `onEnterFrame (AS2)` or `Event.ENTER_FRAME (AS3)`，如果在 HUD 中有多個元素具有 `onEnterFrame//Event.ENTER_FRAME`，每次開發者調用 Advance 將被執行，甚至在特殊元素不被改變也是如此。
4. 保持動畫幀速度爲遊戲幀速度的一半，只在需要時調用 AS。一個通常的遊戲編程範例爲在遊戲引擎的每個幀都進行調用。這在使用 Flash 時明顯不是理想方法。開發者需要避免每幀都調用 AS 腳本代碼以減少記憶體使用並增加性能。例如，如果遊戲運行速率爲 30-40 FPS，只需以 15-20 FPS 的速率更新動畫。但是，幀速率太慢動畫調用不夠頻繁可能導致 HUD 動畫可能會延遲、跳躍以及不夠平滑。

5. 儘量縮短動畫時間軸，因爲延長的時間軸動畫需要使用更多記憶體。但是，這必須小心處理，因爲過短的時間軸將導致畫面跳躍。

4.3 單個動畫視圖

使用這種方法可以非常有效得處理 Flash 複雜多元素介面（例如，雷達介面）。需要仔細對比 Flash 和 C++如何用來渲染元素。多個 Flash 圖層導致獨立的繪圖原型，降低性能。確保充分利用 4.2 小節描述的指導方法另外還有以下方法：

1. 使用遊戲引擎在介面內部繪製元素，而使用 Flash 來繪製邊框和幀，用 Scaleform 處理文本。當一個 HUD 具有多個快速變化的元素時，Flash 可以用來繪製靜態元素並用 C++渲染頻繁改變的項。
2. 使用 C++進行元素定位而 Flash 進行繪製。雷達螢幕爲一個很好的例子；創建一個 Flash 設置的點陣列，用 Render::Renderer 進行管理，使用 RenderString 標識進行元素標記。使用 C++引擎在渲染前重新正確定位元素。這將避免有些 AS 更新過度，但這十分複雜且需要額外編程。

4.4 單個動畫視圖 (Advanced)

使用該方法更加高級並且消耗更多時間，但是能夠節省一些記憶體。在 HUD 創建和完善之前我們不建議使用這種方法。除了使用 4.3 小節描述的方法，還可以使用以下技術：

1. 只在 HUD 中圖像改變時調用 Advance，或者用一個調用處理所有更新。該方法可以用於具有一個 HUD 動畫更新的背景圖層的單個動畫，以及更多複雜 HUD 介面（例如，那些包含文本和進度條）在頂層沒有動畫並沒有調用 Advance。身體狀況條爲一個很好的例子，通常不爲動畫；在沒有發生變化時不需要調用 Advance。（例如，播放第一幀，停止，直到特性改變時候調用顯示和 Advance/invoke）。這裏進行的渲染更加有效，需要較少的 CPU 開銷，但是管理複雜。
2. 利用自定義靜態緩存管理用於頂點資料，並使用 Render::Renderer。該方法與 C++緊密相關，需要技術深厚的 C++圖形程式師的參與。在 Render::Renderer 中，使用不同（自定義）視頻記憶體向量資料存儲，並使用其他部分的 Scaleform 系統以使用靜態緩存代替動態緩存來管理 HUD 元素。
3. 使用線程渲染和 Render::Renderer，這也許是最複雜的方法；需要重寫渲染器並在 HUD 上調用一個獨立的 Advance，由遊戲引擎進行渲染。複雜層次的權衡可以獲得更多潛在性能。

注釋：Scaleform –Unreal® Engine 3 集成環境中存在線程渲染，但是仍然需要大量編程來實現該方法。

4.5 不用 Flash 創建自定義 HUD

這個過程最爲複雜且消耗大量時間。最終，HUD 應該純粹用 C++和點陣圖處理。Scaleform 和 Flash 可以在 HUD 創建和重用過程中始終發揮作用，但是在將 Flash 介面轉變成點陣圖形成最終版本後刪除。在這一點上，Scaleform 不應執行任何播放操作或者將記憶體用戶 HUD 元素，除非 Scaleform Player 在記憶體中產生事件。

根據所能承擔，可以結合自定義調整的 C++渲染器用於性能要求較高的 HUD 用 Scaleform 進行其他的渲染操作。可以從外部自定義渲染器獲益的區域爲具有詳細條目的最小地圖和詳細目錄/狀態介面；這些區域可以通過 DP 批次處理和多個條目的有效更新進行優化，有些任務用 Scaleform 自動操作比較困難。其他的 HUD 元素如邊框、面板和動畫彈出框可以使用 Scaleform 處理，只有在這碰到編程瓶頸時進行別的方法替換。

不管上面說明，我們建議開發者繼續使用 Scaleform 字體/文本引擎，特別是因爲 Scaleform 包含一個 DrawText API 函數包可以使開發者使用 C++進行編程繪製文本，並且使用 Scaleform 用戶介面中相同的 Flash 字體和文本系統。這樣無需擁有兩個單獨字體系統可以節省記憶體：一個用戶 HUD 另外一個用戶其餘的功能表系統。更多關於字體/文本引擎的資訊，以及字體和文本的最佳優化方法，請參考"字體和文本" 章節的 FAQ 和 "字體概述"/"DrawText API 參考" 文檔。

總之，當在 HUD 上進行創建和重用時請注意遵循以下條例：

- 少用動畫剪輯，只在必須時才使用嵌套條目。
- 盡可能不要使用蒙板，最多只能用一到兩次。更多資訊請參考 “圖像渲染和特效” 章節的 FAQ。
- 禁止 PC 和 Wii™（其他控制器已某人禁止）上的滑鼠和鍵盤：
    - `GFx::Movie::SetMouseCursorCount(0);`
    - 如果禁止，不要進行輸入。
- 將動畫剪輯分組進行現實/隱藏，在 HUD 面板將 `noInvisibleAdvance` `noInvisibleAdvance (AS2) or scaleform.gfx.Extensions.noInvisibleAdvance` (AS3 設置爲 `_visible(AS2) or visible (AS3) = false`。

- 只在條目發生變化時進行調用。如果有兩個或更多條目總是同時改變（相同的幀）則將條目變化集中到一個調用進行批量處理。不要對不常改變的幀使用這種處理方法（每幀）。
- 確保點陣圖和色階使用的優化。更多資訊請參考 2.1 節或"藝術和資源"章節中的 FAQ。

5 常用優化提示

5.1 Flash 時間軸

時間軸上的幀和圖層爲 Flash 創作環境的兩個重要部分。這些區域展示了資源存放位置和確定文檔工作。時間軸和庫如何設置和使用將影響整個 FLA 文件和總體可用性以及性能。

- 儘量少用基於幀的迴圈。基於幀的動畫獨立于應用程式幀速率，而基於時間模型則與 FPS 相關。
- 在不需要時立即停止基於幀的迴圈。
- 允許多個幀的複雜代碼塊分佈。
- 與具有大量幀的 tween 動畫時間軸類似，具有大量代碼的腳本也需要大量處理器開銷。
- 評估內容來判斷動畫/交互是否易於通過時間軸實現或者使用 AS 腳本簡化進行模組化處理。
- 避免使用默認圖層名字（如 Layer 1、Layer 2），因爲在複雜文件中不易於記憶或資源的定位。

5.2 常用性能優化

- 結合轉換可以實現更好的性能，例如，嵌套三種轉換，手工計算一個矩陣。
- 如果速度變慢，檢查記憶體泄漏。確保去除不再需要的東西。
- 在創作時，避免過多 `trace()`語句或動態更新文本域，因爲這將影響性能。儘量不要更新太過頻繁（例如，只有當發生改變時進行更新）。
- 如果可以，在時間軸圖層頂部放置包括 AS 的圖層以及一個包含幀標簽的圖層。例如，通常較好的慣例爲將包含 AS 腳本的圖層命名爲 *actions*。
- 不要將幀腳本放在不同的圖層；而將所有的腳本集中在一個圖層。這將簡化 AS 代碼的管理並提高性能，消除了多個 AS 執行代碼傳遞的開銷。

6 附錄

ActionScript 2.0 最佳優化
http://www.adobe.com/devnet/flash/articles/as_bestpractices.html

Flash 8 最佳優化
http://www.adobe.com/devnet/flash/articles/flash8_bestpractices.html

Flash ActionScript 2.0 學習向導
http://www.adobe.com/devnet/flash/articles/actionscript_guide.html