Autodesk®
Scaleform®

Scaleform 효과적인 실습

본 문서는 Scaleform 4.0 으로 애셋을 만들기 위한 효과적인 방법을 제공한다.

저자: Matthew Doyle
버전: 3.0
최종편집일: 2011 년 4 월 25 일

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

연락처:

문서	Scaleform 효과적인 실습 가이드
주소	Autodesk Scaleform Corporation
	6305 Ivy Lane, Suite 310
	Greenbelt, MD 20770, USA
웹사이트	www.scaleform.com
이메일	info@scaleform.com
직통전화	(301) 446-3200
팩스	(301) 446-3199

목차

1 개요

이 문서는 Autodesk® Scaleform® 4.0 을 사용해서 Adobe® Flash® 컨텐츠를 개발할 때 효과적인 실습 내용을 포함하고 있다. 이러한 훈련은 게임상에서 Flash 컨텐츠가 메모리와 성능을 한층 발전시키기 위해 확실한 도움이 된다. 물론, 여기서 다른 용도로 응용하는 것도 가능하다. 이 문서의 컨텐츠 생성 절은 아티스트와 디자이너를 대상으로 한 것이다. 그에 반하여 ActionScript™ (AS) 절은 거의 테크니컬 아티스트와 엔지니어를 대상으로 한 것이다. 헤드업 디스플레이(HUD) 개발 섹션은 HUD 를 만들기 위한 개발 시나리오의 일반적인 개요다. 아티스트와 엔지니어 모두 Flash 와 Scaleform 3.1 으로 HUD 시스템을 개발하기에 앞서 이 섹션을 읽을 것을 추천한다.

이 문서는 고객의 지원 요청, 개발자 포럼의 개시 글들과 다양한 Flash 와 웹상의 AS 리소스처럼 서로 다른 소스들로부터 폭넓고 다양하게 편집되어 제공된다. Scaleform 3.x 은 다양한 플랫폼에서 사용이 가능하고(최첨단 휴대전화와 PC 등) 다양한 엔진들과 통합되었고, 결론적으로 유감스럽지만 "one size fits all"(한가지로 모든 것을 할 수 있는) 솔루션은 아니다. 모든 게임과 프로젝트는 최적의 메모리 소비와 성능을 위한 서로 다른 솔루션을 필요로 한다. (다시 말해서 첨단 PC 에서 휴대용 플랫폼으로 포팅 할 때는 코드수정이 필요할 수도 있다는 얘기다.)

본문에서 서술하는 best practices 와 함께 관련 성능 데이터를 제공하기 위해 최선을 다했으나 본 결과물은 고려해야 할 변수들이 많아 실제 프로젝트 결과와 다를 수 있다.
모든 유저인터페이스(UI) 작업에 있어서 철저한 테스트를 할 것과 대체 솔루션에 대한 부하 테스트를 각 상황에 맞게 UI 개발 프로세스 초기 단계에서 하길 권한다. Flash Studio CS5 는 바이너리 .FLA 형식을 대체하는 XML 기반 콘텐츠 유형을 도입했습니다. 이 유형은 버전을 관리하고 변경 목록을 추적하는 데 매우 유용합니다.

한 명 이상의 디자이너 또는 개발자가 하나의 플래시 프로젝트에서 작업하는 것이 일반적인 것처럼, 플래시를 사용하고, FLA 파일을 구성하고, AS 코드를 작성하기 위한 표준 가이드 라인을 정하고 모두가 그것에 따르는 것이 팀에게 유리하다. 디자이너이든지 개발자든지 또는 혼자서

일하거나 팀원이 되어 일하든지 언제나 유용한 예제들을 받아들여라. 다음은 효과적인 실습이 필요한 몇 가지 이유들이다.

- Flash 또는 AS 문서 상에서 작업할 때:
    - 일관성을 취하는 것과 효과적인 연습들은 작업흐름의 속도를 빠르게 한다. 하나의 문서를 편집할 때 이해와 기억이 쉬운 구조화된 코딩 습관을 사용하는 것이 개발을 빠르게 한다. 게다가 큰 프로젝트일수록 프레임워크가 있을 경우에 포팅과 재사용이 쉬운 코드가 만들어 진다.
- FLA 또는 AS 파일들을 공유할 때:
    - 다른 사람들이 빠르게 AS 를 이해하고, 찾을 수 있고, 일관성 있는 코드 수정이 가능하고, 에셋 찾기와 편집에 대한 것들을 할 수 있도록 문서를 편집한다.
- 애플리케이션 상에서 작업할 때:
    - 다수의 프로그래머들이 마찰이 적으면서도 대단히 효과적으로 애플리케이션상에서 일할 수 있다. 프로젝트 혹은 사이트 관리자는 복잡한 프로젝트나 애플리케이션을 관리하고 구조화 할 수 있다.
- 플래시와 액션스크립트 기술을 가르치거나 배울 때:
    - 효과적인 실습을 통해서 어떻게 어플리케이션을 빌드 하는지 배우고 코딩 규칙을 따름으로써 특별한 방법론을 다시 배울 필요성이 줄어든다. 교육생이 실습을 학습하고 코드를 더 낫게 구조화할 수 있다면 언어습득을 빠르게 하면서도 덜 지칠 것이다.

이 best practice 를 읽음으로 해서 개발자들은 더 많은 것을 발견할 수 있으며 자신의 좋은 습관을 개발할 수도 있다. 여기에서 소개되는 토픽들은 플래시 개발 시에 가이드라인으로 생각하자.
여기서 제안하는 것들의 일부 혹은 전부를 따를 수도 있을 것이다.
개발자들은 이러한 제안들을 작업 방식에 맞게 수정할 수도 있다. 이 단원의 많은 지침사항들은 개발자들에게 플래시와 액션스크립트를 작업하는 방식에 모순이 없도록 도움이 되어 줄 것이다.

Flash 의 최적화가 가능한 부분은 다음과 같다.

- 빠른 재생을 위해 그래픽을 최적화해서 애니메이션 성능을 향상시킨다.
- 빠르게 실행되는 코드를 작성해서 컴퓨팅 성능을 향상시킨다.

만약 코드가 느리게 구동된다면, 개발자는 어플리케이션 또는 문제에 대한 보다 효과적인 방법으로 해결할 수 있는 또 다른 메소드를 찾거나 유효범위(scope)를 줄여야 할 필요가 있다는 의미다. 개발자는 장애요소를 확인하고 제거 해야 한다. 개발자들은 병목현상을 확인하고 제거한다. 그래픽을 최적화하거나 액션스크립트를 통해서 완료된 일들을 줄이는 것이 그 예다.

때로는 수행성능이 통찰력과 관계가 있다. 만약 개발자가 하나의 프레임에서 너무 많은 일을 수행하려 한다면, 플래시는 스테이지에 출력할 시간이 부족해지고, 사용자에게 느리게 보일 것이다. 만약 개발자가 작업의 수행 량을 작은 덩어리로 세분화한다면 플래시는 스테이지를 규정된 프레임 속도로 갱신할 수 있으며, 결과적으로 프레임이 느려지는 일이 없을 것이다.

1.1 일반적인 사용법

- 스테이지상의 오브젝트 수를 줄이는 것을 고려하라. 하나씩 오브젝트를 추가하고 언제 얼마만큼의 성능이 떨어지는지 주목하라.
- (플래시의 컴포넌트 패널에서 사용 가능한)표준 UI 컴포넌트 사용을 피하라. 이 컴포넌트들은 데스크탑 컴퓨터에서 작동하도록 설계되었고 Scaleform 3.3 에서 사용되도록 최적화되어 있지 않다. 대신에 Scaleform Common Lightweight Interface Kit (CLIK™) 컴포넌트를 사용한다.

1.2 고려사항 설정하기

- 오브젝트 계층구조의 깊이를 줄이는 것을 고려하라. 예를 들어, 만약 몇몇 오브젝트가 독립적으로 움직이거나 회전하지 않는다면, 자신의 변환(transform)그룹 내에 존재할 필요가 없다.
- 크기가 큰 유저 인터페이스는 일반적으로 분할된 플래시 파일이 낫다. 그 예로, MMORPG 의 상점은 하나의 Flash 인터페이스로 이루어져 있고, 목숨이 얼마나 남아있는지 보여주는

HUD 는 또 다른 파일이다. AS 에서 SWF 를 불러오는 AS2 및 AS3 의 다양한 API 를 이용해 여러 다른 파일을 불러올 수 있습니다.

- . 이 함수들은 애플리게이션이 인터페이스가 필요할 때 로드하고 필요 없을 때 해제해 주기도 한다. 또 다른 방법은 서로 다른 SWF 파일들을 C++ API 를 거쳐 load 와 unload 하는 것이다. (서로 다른 GFx::Movie 인터페이스들을 생성함.)
- Scaleform 3.3 은 멀티스레드 로딩과 재생을 지원한다. 여러 개의 플래시 파일들을 백그라운드 로딩하는 것은 여분의 CPU 코어가 가용한 상태라면 성능 면에서 불리한 것이 아닐 것이다. 그렇지만 개발자는 디스크에서 찾는 것은 느리기도 하거니와 파일로드가 상당히 느려질 수 있으므로 CD 또는 DVD 에서 다수의 파일을 로딩할 때 주의를 기울여야 한다.
- 큰 SWF 파일을 사용하는 씬은 피한다.
- 게임에서 UI 메모리 사용의 실제 총량은 대략 다음과 같을 것이다.
    - 간단한 게임 HUD 오버레이: 400K–1.5M
    - 애니메이션과 몇몇 장면들로 이루어진 시작/메뉴 스크린: 800K–4M
    - 액션스크립트와 애셋으로 이루어진 간단한 게임(Pacman): 700K–1.5M
    - 주기적으로 변화되는 애셋이 포함된 장시간의 백터 애니메이션: 2M–10M+

2 메모리에 컨텐츠를 생성하는 것과 성능상의 고려사항

플래시 컨텐츠를 개발할 때는 반드시 따라야 하며 또한 구현되어야만 최적의 성능을 발휘시켜주는 고려사항과 최적화가 있다.

2.1 Draw Primitives

Draw primitives(DP)는 GFx 가 셰입(Shape)과 같은 Flash 요소를 렌더링하여 이를 디스플레이에 표시하기 위해 만드는 메쉬 객체입니다. 일괄처리되거나 인스턴스화된(batched or instanced) 렌더링을 지원하는 플랫폼에서 GFx 는 동일한 레이어 및 인접 레이어에 있는 셰입중 속성이 서로 호환되는 셰입를 단일 DP 로 한데 모아줍니다. 각 DP 가 독립적으로 렌더링되므로 성능적인 비용이 발생합니다.

대게, 장면 안에서 DP 가 많을수록 디스플레이 성능은 선형적으로 떨어진다. 그러므로 되도록 DP 의 수를 낮게 유지하도록 훈련하는 것이 좋다. Scaleform Player 에서 F2 키를 누르면 나오는 AMP HUD 요약 화면을 통해서 DP 의 수를 정확히 알 수 있다. 이 화면은 삼각형과 DP 의 수, 메모리 사용량, 그리고 다른 최적화 정보를 보여준다.

다음은 DP 수를 낮추는데 도움을 주는 몇 가지 방법들이다.

1. DP 는 속성이 동일한 항목(순색이 다른 셰입 제외)만을 포함합니다. 예를 들어, 텍스처 또는 블렌딩 모드가 다른 항목은 같은 DP 에 합칠 수 없습니다.
2. 서로 다른 레이어에 중첩되는 객체는 속성이 동일하다 해도 같은 DP 에 합칠 수 없습니다.
3. 한 셰입에 그라디언트 채우기를 동시에 여러 번 사용하면 DP 수가 증가할 수 있습니다.
4. 순색으로 채워지고 스트로크가 없는 벡터 그래픽은 매우 저렴합니다.
5. 빈 동영상 파일은 DP 가 필요하지 않지만 객체가 포함된 동영상 파일은 해당 객체가 필요로 하는 양만큼의 DP 가 필요합니다.

AMP 에서 해당 옵션을 활성화하거나 Ctrl+J 를 누르면 Scaleform Player 에서 일괄처리 및/또는 인스턴스 그룹이 출력됩니다. 이 모드에서는 색상이 동일한 항목이 단일 DP 로 렌더링됩니다.

2.2 무비클립

1. 무비클립을 숨기기 보다는 사용하지 않을 때 타임라인에서 완전히 삭제하는 것이 제일 좋은 방법이다. 그렇지 않으면 Advance 중에 계속 프로세싱 타임을 잡아먹는다.
2. 무비클립의 과도한 내포관계(nesting)는 피한다. 이것은 성능에 영향을 준다.
3. 만약 무비클립을 숨기는 것이 필요하다면 _alpha=0 보다 _visible=false (in AS2) and visible=false rather than alpha=0 (in AS3).를 사용한다. 숨겨진 무비클립 안에서의 애니메이션이 멈춰졌는지 stop()함수를 이용해 확인하라. 그렇지 않으면 보이지 않는 애니메이션은 여전히 공간을 차지하며 성능에 영향을 미친다.

2.3 Artwork

2.3.1 비트맵 vs 백터 그래픽

플래시 컨텐츠는 백터 아트와 이미지로 만들어질 수 있고 Scaleform 는 백터와 비트맵 그래픽 양쪽을 끊김 없이 렌더링 할 수 있다. 하지만 각각의 타입은 장점과 단점이 있다. 벡터를 쓸지, 비트맵을 쓸지를 결정하는 것이 항상 확실한 것은 아니다. 이번 장은 백터와 비트맵의 몇몇 차이점에 대해 논의해 봄으로써 컨텐츠 저작 결정에 도움이 되고자 한다.

백터 그래픽이 크기를 조절할 때 매끄러운 형태를 유지하는 것에 비해서 비트맵 이미지는 깍두기처럼 픽셀이 도드라진다. 그러나 비트맵과는 다르게 백터 그래픽은 더 많은 연산시간을 필요로 한다. 간단한 단색 컬러 형태라면 비트맵만큼 빠르겠지만, 복잡한 백터 그래픽은 많은 삼각형과 셰입, 그리고 내부 채우기 때문에 출력비용이 비싸다. 따라서 백터 셰입의 과다한 사용은 가끔 전반적인 애플리케이션의 성능을 저하시킨다.

비트맵 그래픽은 벡터처럼 많은 연산시간을 잡아먹지 않기 때문에 몇몇 애플리케이션에서는 더 나은 선택일 수 있다. 하지만 비트맵 그래픽은 벡터 그래픽에 비해서 메모리 소요량이 엄청나게 높다.

2.3.2 백터 그래픽

백터 그래픽이 다른 이미지 포맷보다 컴팩트한 이유는 원본 그래픽(픽셀) 데이터인 비트맵과 다르게 백터에서 실시간으로 렌더링 하기 위한 수학(점, 곡선, 채우기)이 정의되어있기 때문이다. 그러나 벡터 데이터를 최종 이미지로 변환하는 것은 시간이 소요되고 항상 그래픽의 크기 또는 외관이 변하는 것들에 대한 중요한 정보가 있어야 한다. 만약 셰입의 윤곽이 복잡하고 매 프레임마다 변화한다면 애니메이션이 느려질 것이다.

다음은 효과적인 벡터 그래픽 랜더를 위한 지침이다

- 복잡한 벡터 그래픽을 비트맵으로 변환한 후에 이것이 성능에 어떻게 영향을 미치는지 실험하라.
- 알파 블랜딩을 사용할 때 다음 사항에 염두 하라.
    - 단색 채우기(solid fills)를 사용하는 것이 알파 블랜드 외곽선 보다 더 효과적인 알고리즘을 사용할 수 있기 때문에 비용이 싸다.
    - 투명(알파) 사용은 피하라. 플래시는 투명한 셰입 내의 모든 픽셀을 체크해야 하므로 렌더링이 상당히 느려진다. 클립을 숨길 때는 _alpha(AS2) or alpha (AS3)속성을 0 으로 바꾸기보다는 _visible(AS2) or visible (AS3) 속성을 false 로 한다. 그래픽들은 _alpha/alpha property 가 100 일때 가장 빠르게 렌더된다. 무비클립의 타임라인을 셋팅할때 키프레임이 비어 있으면(그 결과 무비클립은 보여지는 것이 없음) 일반적으로 더 빠르다. 가끔 플래시는 보이지 않는 무비클립을 렌더하려고 하기도 한다. _x 와 _y(in AS2) or x and y properties (in AS3)속성을 조정해서 클립을 바깥 스테이지로 이동시키는 것 외에도 _visible/visible 속성을 false 로 해두면 플래시는 그릴 시도조차 않는다.
- 벡터 셰입 최적화
    - 벡터 그래픽에서 간단한 셰입을 사용하는 것보다 중복되는 점들을 삭제하는 것에 중점을 둔다. 이렇게 하면 재생기의 벡터 셰입 계산량을 줄여줄 것이다.
    - 원, 사각형, 선을 포함한 프리미티브 벡터를 사용하라.

    - 플래시의 그리기 성능은 프레임마다 얼마나 많은 점들을 그리는 지에 달려있다. Modify -> Shape 서브메뉴에서 Smooth, Straighten 나 Optimize 를 선택(그래픽에 따라 다르다)하면 그리기에 필요한 점들의 개수를 줄여준다.
- 모서리가 곡선보다 비용 면에서 저렴하다.
    - 너무 많은 곡선과 점을 갖는 복잡한 벡터는 피한다.
    - 모서리는 수학적으로 곡선보다 간단하다. 가능하다면 단순한 단면으로 구성한다. 특히 매우 작은 벡터 셰입에 말이다. 이 방법으로 곡선을 흉내 낼 수 있다.
- 그라디언트 채우기(gradient fills)와 그라디언트 외곽선(gradient strokes)의 사용을 자제한다.
- 셰입의 외곽선(strokes) 사용을 자제한다.
    - 출력되는 선의 양이 너무 많아 질 수 있기 때문에 가능한 한 벡터 셰입의 외곽선을 사용하지 않는 것이 좋다.
    - 벡터 이미지를 감싸는 외곽선은 성능을 떨어뜨린다.
    - 채우기가 셰입의 내부만 렌더링하는데 반해서 외곽선은 내부와 외부를 렌더해야한다. 따라서 두배의 작업량이 된다.
- 플래시의 그리기 API 사용을 최소화한다. 불필요한 사용은 중요한 성능상의 오버헤드를 발생시킨다. 꼭 필요하다면, 그리기 API 를 무비클립에서 한번만 사용하라. 이러한 커스텀 무비클립을 렌더링 할 때는 성능저하가 없다.
- 마스크의 사용을 제한한다. 마스크 처리된 픽셀들은 렌더링 시간을 잡아먹고, 그것들이 비록 그려지지 않더라도 성능상에 부정적인 영향을 끼친다. 다수의 마스크들은 마스크 수만큼 복합적인 영향을 끼친다. 마스크가 필요 없는데도 불구하고 불필요하게 마스크가 사용되는 비주얼 효과가 많다는 사실에 주의하라. 특히, 비트맵의 특정 부분을 잘라내기 위해서 마스크를 사용한다. 플래시 스튜디오에서 직접 셰입에 비트맵으로 채우기를 적용하면 더 효과적으로 동일한 것을 할 수 있다. 이것은 또한 Scaleform 이 특허출원중인 EdgeAA 안티 앨리어싱이라는 부가적인 이득도 제공한다.
- 위에서 설명한 대로 드로 프리미티브가 추가로 생성되지 않도록 가능한 한 많은 비중첩 객체를 변환해 속성이 같아지도록 하십시오.

- 셰입이 생성된 후에는 부가적인 메모리 사용 없이 이동, 회전, 블렌딩 될 수 있다. 그러나 새로운 대폭적인 크기변환은 테셀레이션이 발생하므로 더 많은 메모리를 소모하게 될 것이다.
- EdgeAA 가 사용 가능한 상태에서, 다중 그라디언트/비트맵으로 생성한 셰입보다 다중 단일 색상으로 생성된 셰입이 더 빠르게 렌더된다. 그라디언트/비트맵을 하나의 셰입 내에 연결하는 것은 DP 수를 급격하게 늘리게 되므로 주의해야 한다.

2.3.3 비트맵

최적화되고 간결한 애니메이션 또는 그래픽을 생성하려면 일단 먼저 프로젝트의 계획을 세우고 윤곽을 만들어 보는 것이다. 파일크기, 메모리 사용량, 만들려는 애니메이션의 길이를 결정하고 개발공정을 테스트 해보는 것이다.

앞서 말한 DP 외에도 렌더링에 영향을 주는 큰 요소가 있는데 그것은 바로 그려지는 전체 표면이다. 항상 보여지는 셰입 또는 비트맵은 스테이지 상에 있고, 그것이 비록 겹쳐진 셰입에 의해 숨겨져 있다 하더라도 비디오카드의 채우기 속도(fill rate)를 잡아먹게 된다. 비록 오늘날 비디오카드들이 플래시 소프트웨어보다 많이 빨라졌다 할지라도, 방대하게 겹쳐서 그려지는 알파블랜드 오브젝트, 그리고 오래되고 낮은 사양의 하드웨어는 여전히 성능을 떨어뜨릴 것이다. 따라서 겹쳐진 셰입과 비트맵을 합치고 클리핑 될 객체를 명시적으로 숨기는 것이 매우 중요하다.

SWF 파일에서 오브젝트를 숨길 때의 최상의 방법은 무비클립 인스턴스의 _alpha(in AS2) or alpha (in AS3)값을 0 으로 넣지 않고 _visible (in AS2) or visible (in AS3)속성을 false 로 하는 것이다. 비록 Scaleform 는 _alpha 가 0 인 오브젝트를 그리지 않을 지라도 그들의 자식들은 여전히 애니메이션과 액션스크립트 때문에 CPU 처리 비용을 초래한다. 만약 인스턴스의 보이는 상태가 false 로 되어있다면, CPU 사이클과 메모리를 절약할 것이다. 따라서 SWF 파일들이 어플리케이션에서 부드럽게 애니메이션 되고 성능이 더 좋아질 것이다. 에셋을 언로딩(unloading)과 리로딩(reloading)하기 보다는 _visible 속성을 false 로 설정하는 것이 프로세서에 대한 부하를 덜어준다. 다음은 비트맵 그래픽을 효과적으로 랜더 할 수 있는 몇 가지의 가이드라인이다.

- 모든 텍스쳐/비트맵의 높이과 폭을 2 의 제곱 크기로 사용한다. 비트맵 사이즈를 16x32, 256x128, 1024x1024, 512x32 등으로 사용하는 것이 그 예다.
- JPEG 이미지 압축을 사용하지 않는다. 대상 하드웨어에서 지원하지 않은 압축 이미지는 사용하지 마십시오. 예를 들어 JPEG 형식은 압축 이미지이지만, 하드웨어적으로 이를 지원하는 플랫폼이 없습니다.텍스쳐 압축 변환에는 gfxexport 도구를 사용하여 대상 플랫폼의 하드웨어에서 지원하는 압축 이미지를 만드십시오. 예를 들어 DXT 압축은 많은 플랫폼에서 지원됩니다. 최종 SWF 를 GFX 형식으로 변환하면 텍스처 압축 시 필요한 비트맵 메모리를 줄일 수 있습니다. 압축된 텍스처를 사용하면 그렇지 않은 경우보다 비트맵 메모리 사용량이 크게 줄어듭니다. 텍스처를 압축할 때 이미지 품질이 떨어지는 형식(DXT 포함)이 많으므로 압축 텍스처의 비트맵 품질이 만족스러운지 확인하십시오. gfxexport 의 옵션으로 –qp 나 –qh 를 사용하면 최고 품질의 DDS 텍스쳐를 얻을 수 있다. (이 옵션은 비트맵 이미지를 출력하기까지 많은 시간이 걸리는 것에 유념하라.)
- 더 작은 개수의 그리고 더 작은 크기의 비트맵을 사용하라.
- 만약 하나의 비트맵이 크고 간단한 Shape 에 출력된다면, 그 비트맵을 백터 그래픽을 사용해 재생성 하라. 이렇게 하면 메모리를 절약하고 EdgeAA 와 함께 더 높은 품질의 결과를 보여줄 것이다.
- 큰 비트맵을 액션스크립트를 통해 loading 과 unloading 하는 것에 대해서는 신중을 기하라.
- SWF/GFX 파일의 크기로 메모리 사용량을 판단해서는 안 된다. 각각의 SWF 나 JPG 의 크기가 작다고 하더라도, 로드 되고 압축이 해제되면 많은 양의 메모리를 사용할 수 있다. 저용량 SWF 파일이라도 불러올 때 메모리를 많이 사용할 수 있습니다. 예를 들어 1024 x 1024 크기의 JPEG 이미지를 포함한 SWF 파일의 압축을 풀고 실행하면 이 이미지를 표시하기 위해 4MB 의 메모리가 필요합니다.
- UI 에서 사용되는 이미지 크기와 개수를 추적하는 것이 중요하다. 모든 이미지의 크기를 계산한 후에 그 수를 모두 더하고, 4 를 곱한다. (보통 픽셀당 4 바이트 임) gfxexport 툴에서 이미지 메모리를 줄여주는 옵션인 –i DDS 를 사용하는 것에 유의하라. AMP 를 사용해서 비트맵으로 인한 메모리 소비를 체크하라.

- 종합적으로, 대부분의 경우 비트맵은 어떤 복잡한 셰입들 보다 더 빠르지만, 벡터가 더 좋게 보이는 것도 사실이다. 정확한 상관관계는 시스템의 채우기 속도(fill rate), 변환 셰이더 성능, CPU 속도에 달렸다. 결국 대상 시스템상에서 성능을 시험하는 것이 유일한 방법이다.
- 그라디언트 텍스처들만 들어있는 gradient.swf 마스터 파일을 생성하라. 그라디언트가 필요한 다른 SWF 파일에서 이 파일을 import 한다. gfxexport 툴로 gradients.swf 를 -d0 변환 옵션으로 익스포트한다. 이 옵션은 SWF 파일의 모든 텍스쳐에 압축을 하지 않도록 하는 것이다. 이렇게 하면 이 파일의 모든 텍스처에서 그라디언트를 공짜로 사용할 수 있다.
- 비트맵의 알파채널 사용을 가급적 피하라.
- 비트맵을 플래시가 아닌 포토샵에서 최적화하라.
- 만약 비트맵에 투명한 부분이 필요하고 표면 영역을 그리는 총량을 줄이려면 PNG 를 사용한다.
- 커다란 비트맵이 겹치는 것은 채우기 속도에 영향을 주므로 피한다.
- 비트맵 그래픽을 애플리케이션에서 사용하려는 크기로 임포트하라, 절대로 큰 파일을 임포트해서 플래시에서 줄여 사용하지 말라. 텍스처 크기나 메모리 양쪽의 낭비다.

2.3.4 애니메이션

하나의 어플리케이션에 애니메이션을 추가할 때 FLA 파일의 프레임 속도(Frame rate)를 고려한다. 그것은 최종 SWF 파일의 성능에 영향을 미칠 수 있다. 프레임 속도를 너무 높게 설정하면 성능상의 문제를 야기할 수 있고, 특히 많은 애셋들이 사용될 때나 프레임 속도대로 틱(tick) 되고 있는 애니메이션을 액션스크립트로 생성할 때 그렇다.

그러나 그 프레임 속도 설정은 또한 얼마나 애니메이션이 부드럽게 재생되는지에 영향을 준다. 예를 들면, 어떤 애니메이션이 초당 프레임 수(FPS)를 12 로 설정한다는 것은 초당 타임라인의 12 프레임이 재생된다는 것이다. 만약 그 문서의 프레임 속도가 24 FPS 로 설정되었다면, 그 애니메이션은 12 FPS 로 설정되었던 것보다 더 부드럽게 움직이는 것처럼 보인다. 24FPS 의 애니메이션은 12 FPS 보다 두 배 빠르지만 총 재생시간(초)은 그의 절반이다. 그러므로, 더 높은

프레임 속도를 사용해 5 초의 애니메이션을 만들기 위해서 삽입되는 프레임들은 그 보다 낮은 프레임 속도 보다 5 초를 더 채울 필요가 있다. 더불어 파일의 크기도 증가한다.

주의: onEnterFrame (in AS2) or Event.ENTER_FRAME (in AS3)이벤트 핸들러를 사용해서 스크립트된 애니메이션을 생성할 때는 애니메이션이 타임라인의 모션트윈을 생성하는 것과 비슷하게 그 문서의 프레임 속도로 동작한다. onEnterFrame/EVENT.ENTER_FRAME 을 대체할 수 있는 하나의 방법은 setInterval 이 있다. 프렘임 속도에 의존하는 것이 아닌 밀리 초 간격으로 함수들이 호출된다. onEnterFrame/EVENT.ENTER_FRAME 처럼, 빈번하게 함수를 호출하거나, 좀더 리소스가 강조된 애니메이션에 setInterval 이 사용된다.

실행 시 부드러운 애니메이션이 출력되는 선에서 되도록 낮은 프레임 속도를 사용한다. 이것은 프로세싱 시간을 줄이는데 도움이 될 것이다. 30~40FPS 이상의 프레임 속도를 사용하지 말라. 높은 프레임 속도는 애니메이션을 크게 부드럽게 해주지 못할 뿐만 아니라, CPU 비용만 늘어난다. 대부분의 경우, 플래시 UI 는 게임 프레임 속도의 절반으로 설정하면 안전하다고 할 수 있다.

다음은 디자인과 효과적인 애니메이션에 도움이 되는 몇 가지 가이드 라인이다.:

- 스테이지 상의 오브젝트들의 수와 빠르게 움직이는 것들이 전체 성능에 영향을 준다.
- 스테이지 상에 무비클립의 수가 많이 있고 그것들이 빠르게 on/off 상태로 전환된다면 붙이기 (attaching)와 삭제(removing)보다는 그것들의 보이는 상태를 제어 할 수 있는 _visible/visible = true/false 를 사용한다.
- 트윈을 사용할 때 세심한 주의를 기울인다.
    - 너무 많은 것들을 일제히 트윈하는 것을 피한다. 어떤 것이 시작되면 또 다른 것이 끝날 수 있도록 트윈 또는 반복 진행되는 애니메이션의 수를 줄인다.
    - 되도록 표준 플래시의 Tween 클래스 대신에 성능상 오버헤드가 훨씬 적은 타임라인의 모션 트윈을 사용한다.
    - Scaleform 은 표준 플래시의 Tween 클래스 보다 작고, 빠르고, 깔끔한 CLIK 의 Tween(gfx.motion.Tween in AS2 or scaleform.clik.motion.Tween in AS3) 클래스를 사용하기를 권한다.

- 프레임 속도가 서로 달라 어떤 것이 높고 낮은지 종종 티가 나지 않을 때는 프레임 속도를 낮게 유지한다. 높은 프레임 속도의 애니메이션은 부드럽지만, 성능에 있어서는 역효과다. 게임이 초당 60 프레임으로 동작할 때 플래시 파일은 60 FPS 로 설정할 필요가 없다. 그 플래시의 프레임속도는 필요한 비주얼 효과에 대한 최소값이 필요한 것이다.
- 투명도와 그라디언트는 프로세서 작업에 큰 비용이 소요되므로 절약해서 사용해야 한다.
- 포커싱 되는 영역을 잘 디자인하고 애니메이션 시키고, 화면의 다른 영역은 애니메이션과 효과를 줄인다.
- 장면을 전환하는 동안 배경 애니메이션을 수동으로 잠시 멈춘다. (즉, 미세한 배경 효과)
- 애니메이션 되는 요소들에 대한 삽입과 삭제가 성능상 얼마만큼의 영향을 주는지 테스트한다.
- 트윈을 현명하게 사용한다. 느린 하드웨어에서는 외형이 만들어지는 것이 느려질 수 있다.
- 원에서 사각형으로의 변환 같은 CPU 연산이 많은 셰입 모핑 애니메이션의 사용은 피한다. 셰입 트윈(모핑)은 매 프레임마다 재계산 하므로 CPU 연산이 상당히 많다. 그에 대한 비용은 셰입의 복잡성(모서리, 곡선, 교차점)에 따라 다르다. 몇가지 경우에는 유용할 수도 있지만 비용이 받아들일만한지 테스트부터 해보는 것이 좋다. 4 개의 삼각형 트윈은 받아들일 만할 것이다. 성능과 메모리상에서 상반관계가 있다는 것을 반드시 알아두어야 한다. 일반 셰입을 그릴 때는 테셀레이션과 셰입캐싱이 발생하는데, 이는 나중에 더 효율적으로 그리기 위함이다. 몰프를 사용하면 상반관계가 변하게 되는데 이는 셰입에 대한 모든 변화는 결국 예전 메시를 해제하고 새로 생성하는 것이기 때문이다.
- 가장 효과적인 애니메이션은 이동(translation)과 회전(rotation)이다. 크기변화는 재테셀레이션을 해서 메시가 더 많은 메모리를 차지하고 성능항의 불이익을 줄 수 있으므로 피하는 것이 상책이다.

2.3.5 텍스트와 폰트

- 텍스트 글리프 폰트 크기는 폰트 캐시 관리자의 SlotHeight 또는 gfxexport 에서 사용될 크기보다 작아야 한다(기본 크기는 48 픽셀). 만약 그보다 큰 폰트가 사용되면 벡터들은 DP 가 많아져서 느려질 것이다. (각각의 벡터 글리프는 하나의 DP 를 생성함.)

- 텍스트필드에서 가능하면 외곽선과 텍스트의 배경을 사용하지 않는다. 그러면 DP 를 절약할 수 있을 것이다.
- 매 프레임 마다 텍스트필드를 갱신하는 것은 크게 성능을 저하시키는 것들 중 하나지만 쉽게 피할 수 있다. 텍스트필드의 내용이 진짜로 변경되었을 때 또는 최대한 느린 주기로 갱신한다. 예를 들어, 초 단위로 출력을 갱신할 때, 30 FPS 의 프레임 주기로 업데이트 할 필요가 없다. 그 대신에 예전 값을 저장해두고 텍스트필드의 새로운 값이 예전 값과 다를 때만 다시 대입하는 것이다.
- 텍스트 필드와 링크된 변수를 사용하지 말라(TextField.variable 속성). 이는 매 프레임마다 변수를 가져오고 비교하느라고 성능에 영향을 줄 것이다.
- "htmlText"속성을 재 대입하는 갱신을 최소화한다. HTML 을 파싱하는 것은 비교적 비싼 처리다.
- 글리프 셰입들의 메모리를 절약하기 위해서(특히, 만약 아시아 폰트가 포함되어 있다면) gfxexport 의 컴팩트 폰트 옵션인 –fc, -fcl, -fcm 를 사용한다. 자세한 내용은 "Font Overview" 문서를 참고한다.
- 폰트를 위해 꼭 필요한 심볼만 임베드 하거나 지역화가 필요하다면 fontlib 메커니즘을 사용한다. ("Font Overview" 문서를 참고하기 바람)
- 필요한 텍스트필드 객체의 개수를 최소한으로 사용하고, 되도록 다수의 아이템들을 하나로 결합시킨다. 하나의 텍스트필드는 다른 색깔들과 다른 폰트 스타일을 사용했다면 하나의 DP 로 렌더될 수 있다.
- 텍스트 필드 크기변화나 큰 크기의 폰트 사용을 피하라. 특정 크기 이상의 텍스트 필드는 벡터 글리프로 전환되며, 각 벡터 글리프는 하나의 예가 된다. 만일 클리핑이 필요하다면(백터 글리프의 일부가 보이는 경우) 그때는 마스크가 사용된다. 마스크는 느리고 하나의 DP 를 소모한다. 레스터화 된 글리프의 클리핑은 마스크가 필요 없다.
- 글리프 캐시 사이즈가 사용되는 모든 글리프만큼 충분히 큰지 확인한다. 만약 캐시 사이즈가 충분하지 않다면 몇몇 글리프는 보이지 않게 되거나 빈번한 글리프 레스터화로 인한 심각한 성능 저하가 발생한다.

- 블러(blur)나 드랍쉐도우(drop shadow) 또는 노크아웃(knockOut) 필터 같은 텍스트 효과를 사용하는 것은 폰트 캐시에 추가적인 공간이 필요하고, 또한 성능에 영향을 미친다. 가능하다면 텍스트 필터의 사용을 최소화하라.
- 가능하다면 무비들을 분할해서 생성하는 대신에 DrawText API 를 사용하라. DrawText API 는 개발자가 Scaleform 에서 생성한 UI 의 텍스트 시스템을 사용해서 C++에서 텍스트 그리기를 해주는 것이다. 스크린상에서 움직이는 아바타의 이름 빌보드나 레이더 상에서 다음 아이템의 레이블을 렌더링할 때 플래시 UI 보다 C++을 사용하는 것이 더 효율적일 수 있다. 더 자세한 내용은"DrawText API Reference"를 참고하라.

3 액션 스크립트 최적화

액션스크립트는 기계어 코드로 컴파일 되지 않고, 기계어보다는 빠르지 않지만 인터프리터 언어보다는 빠른 바이트코드로 변환된다. AS 가 느려질 수도 있기는 하지만 (코드가 아닌)그래픽, 오디오, 비디오 에셋들이 성능의 한계를 만들어 버리기도 한다.

액션스크립트의 많은 최적화 기술들이 알려지지는 않았지만 최적화 컴파일러 없이도 모든 언어에서 사용할 수 있는 코딩 방법이 있다. 예를 들어서 반복문 내에서 바뀌지 않는 값이 있다면 그 값은 반복문 바깥으로 빼는 것이 더 빠르다.

3.1 일반적인 액션스크립트 가이드라인

다음의 최적화 방법들은 AS 실행 속도를 빠르게 할 것이다.

- AS 를 적게 사용하면 더 빠르다. 주어진 작업에서 항상 코드의 사용량을 최소화한다. AS 는 그래픽 요소를 생성하는 것이 아닌, 상호작용을 위해 우선 사용해야 한다. 만약 과도한 attachMovie 호출로 코드가 이루어져있다면 FLA 파일의 설계를 어떻게 해야 할지 재고하라.
- AS 는 가능한 한 간단하게 유지한다.
- 스크립트로 작동되는 애니메이션을 아껴서 사용한다. 타임라인 애니메이션이 일반적으로 성능상 더 좋을 것이다.
- 과도한 문자열 조작은 피한다.
- 종료조건을 피하기 위해서 반복 무비클립에서 "if"를 남발하지 말라.
- on()또는 onClipEvent()이벤트 핸들러 사용을 피하고 onEnterFrame, onPress 등을 대신 사용한다.
- 주요한 AS 로직을 프레임에 위치시키는 것을 최소화한다. 대신 중요한 코드의 큰 부분을 함수내에 둔다. AS 컴파일러는 함수 몸체 내에 소스코드를 위치시키는 것이 프레임이나 구형 이벤트 핸들러(onClipEvent, on)보다 훨씬 빠른 코드를 만들어준다. 반면 타임라인

제어(gotoAndPlay, play, stop)나 그 외의 치명적이지 않은 로직이라면 프레임에 두어도 무방하다.

- 몇 가지의 에니메이션이 있는 긴 타임라인의 무비클립안에서 먼 거리의 gotoAndPlay/gotoAndStop 사용을 피하라.
    - 앞으로 gotoAndPlay/gotoAndStop 하는 경우에, 현재의 프래임으로부터 목표 프래임까지가 멀수록 타임라인을 제어하는데 더 많은 비용이 든다. 그러므로, 가장 비싼 비용의 gotoAndPlay/gotoAndStop 은 첫번째 프래임부터 마지막 프래임까지 재생하는 것이다.
    - 뒤로 gotoAndPlay/gotoAndStop 하는 경우에 시작점 타임라인으로부터 목표 프래임까지의 거리가 멀수록 타임라인을 제어하는데 더 많은 비용이 든다. 그러므로, 가장 비싼 비용의 gotoAndPlay/gotoAndStop 은 마지막 프래임부터 마지막의 이전 프래임까지 재생하는 것이다.
- 짧은 타임라인의 무비클립을 사용하라. gotoAndPlay/gotoAndStop 의 비용은 키프래임의 수와 타임라인 에니메이션의 복잡도에 따라 상당히 달라진다. 그러므로, 만약 gotoAndPlay/gotoAndStop 을 호출해서 처리할 계획이 있다면, 길고 복잡한 타임라인 에니메이션을 생성하지 않는 것이 좋다. 대신, 무비클립을 몇몇의 독립적인 무비클립으로 분할하고, 짧은 타임라인을 갖도록 한다. 그리고 gotoAndPlay/gotoAndStop 의 호출을 적게 한다.
- 만약 상당히 많은 오브젝트들이 동시에 갱신된다면, 그 오브젝트들이 하나의 그룹으로 갱신될 수 있게 하는 것이 개발 시스템에서 필수적으로 이루어져야 할 일이다. C++ 상에서 AS 로 대용량 데이더를 전송할 때는 GFx::Movie::SetVariableArray 호출을 사용한다. 이를 사용하면 업로드된 배열에 의해서 한번의 호출로 여러 개의 오브젝트를 갱신할 수 있다. 여러 개의 호출을 하나로 그룹화하면 각각 개별적으로 호출하는 것보다 더 빠르다.
- 단일 프레임에서 너무 많은 일을 시키거나 Scaleform 가 스테이지 렌더링 할 시간조차 없을 경우에는 사용자가 느려짐을 체감 할 수 있다. 대신에 작업량을 좀 더 작은 덩어리로 분리해서 스테이지 프레임 속도에 맞춰서 업데이트 할 수 있도록 한다.
- 객체(Object) 타입을 남용하지 않는다.

- 컴파일러가 타입을 체크할 때 버그가 발생할 수도 있기 때문에 데이터 타입에 대한 선언은 정확히 한다. 객체 타입은 적당한 대안이 없을 때만 사용한다.

- eval() 함수 또는 배열 접근 연산자 사용은 피한다. 가끔은 지역변수로 한번 선언해두고 사용하는 것이 오히려 바람직하고, 더 효과적이기도 하다.
- Array.length 를 미리 변수에 넣어두고 반복문에서 사용한다. myArr.length 를 직접 사용하지 말라.

  다음과 같이 사용하라는 것이다.

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

  다음과 같이 사용하지 말라는 것이다.

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

- 이벤트를 현명하고 명확하게 관리하라. 호출 전에 이벤트 리스너가 존재하는지(즉 null 이 아닌지) 체크함으로써 이벤트 리스너 배열을 간결화하라.
- 오브젝트에 대한 참조를 해제 하기 전에 명시적으로 removeListener()를 호출 함으로서 오브젝트의 리스너를 삭제한다.
- 패키지 이름의 레벨 개수를 최소화해서 시작 시간을 줄입니다. 시작 시, AS VM 은 레벨 하나당 하나씩 연속된 오브젝트를 생성해야 합니다. 또한, 각 레벨 별 오브젝트를 생성하기 전에, AS 컴파일러는 "if" 조건을 추가하여 해당 레벨의 생성 여부를 확인합니다. 따라서, "com.xxx.yyy.aaa.bbb" 패키지에 대해 VM 은 "com", "xxx", "yyy", "aaa", "bbb" 오브젝트를 생성하며, 각각을 생성하기 전에 각 오브젝트의 존재를 확인하는 "if' 조건 코드를 가집니다. 각 레벨의 구문 분석을 요구하는 이름을 해결한 후로는 깊숙히 보관된 오브젝트/클래스/함수에 대한 접근이 느립니다("com"를 해결한 다음 "xxx", 그 다음에 "xxx" 내부의 yyy" 등). 추가적인 오버헤드를 피하기 위해 일부 Flash 개발자들은 SWF 를

컴파일하기 전에 전처리장치 소프트웨어를 사용하여 c58923409876.functionName()와 같은 단일 레벨 고유 식별자로의 경로를 줄입니다.

- 만약 어플리케이션이 동일한 AS 클래스들을 사용하는 여러 개의 SWF 파일로 구성되어있다면, 선택된 SWF 파일들을 컴파일 하는 동안 그 클래스들을 배제시킨다. 이것은 런타임 메모리 소모량을 줄여주는데 도움이 된다.
- 만약 AS 가 키프레임 상에서 타임라인이 완료되기까지 상당한 시간이 필요하다면 여러 개의 키프레임으로 코드를 분할하는 것을 고려하라.
- 최종 SWF 파일을 배포할 때 코드의 trace()구문을 삭제하라. 이렇게 하려면, Publish Settings 대화상자 안의 플래시 탭에 있는 Omit Trace Actions 체크박스를 선택하고 코드를 주석 처리하거나 삭제한다. 이는 실행 시 디버깅용으로 사용되는 trace 구문을 불능화하는데 효과적인 방법이다.
- 상속은 메소드 호출을 증가시키고 메모리를 더 많이 사용한다. 상위클래스로부터 몇 가지 기능을 상속받는 클래스보다 모든 기능을 포함하고 있는 클래스가 실행 시에 더 효율적이다. 그러므로 클래스의 확장성과 코드의 효율에 대한 상충관계를 고려해 디자인할 필요가 있다.
- 하나의 SWF 파일에서 직접 만든 AS 클래스가(예: foo.bar.CustomClass) 포함된 다른 SWF 파일을 로드하거나 언로드 할 때 그 클래스는 여전히 메모리에 잔존한다. 메모리를 절약하려면, 직접 만든 클래스를 언로드하는 SWF 안에서 명시적으로 삭제해준다. delete 구문을 이용해야 하고 다음 예처럼 완전한 클래스 이름을 지정한다.

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

- 모든 코드가 모든 프레임에서 실행 되야 하는 것은 아니다. 시간이 크리티컬한 것이 아니라면 코드가 번갈아 가면서 사용되는 경우 플립플롭을 사용하라
- 되도록 onEnterFrames 은 적게 사용하라.
- 수학(math) 함수를 사용하지 않고 데이터 테이블을 미리 산출한다.
    - 과도한 수학이 사용된다면 먼저 계산된 값들을 변수배열에 저장할 것을 고려하라. 이들 값을 꺼내는 것이 Scaleform 가 직접 계산하는 것보다 훨씬 빠르다.

3.1.1 루프

- 반복되는 액션들과 루프의 최적화에 중점을 둔다.
- 루프들의 수와 각각의 루프가 포함된 코드의 총량을 제한한다.
- 프레임 기반의 루프는 필요 없어졌을 때 즉시 멈춰라.
- 루프 안에서 여러 번 함수를 호출하는 것은 피한다.
    - 루프 안에 작은 함수를 포함시키는 것이 더 낫다.

3.1.2 함수

- 가능하면 깊이 중첩된 함수는 피한다.
- 함수 내에서 with 구문을 사용하지 않는다. 이 연산자는 최적화가 이루어지지 못하도록 한다.

3.1.3 변수/속성

- 존재하지 않는 변수, 오브젝트 또는 함수의 참조를 피한다.
- 가능한 한 "var" 키워드를 사용한다. "var" 키워드를 함수 안에서 사용하면 지역변수에 접근할 때 이름이나 해쉬 테이블을 사용하지 않고 직접 인덱스를 통해서 내부 레지스터로 접근하기 때문에 컴파일러 최적화에서 매우 중요하다.
- 지역변수로도 충분하다면 클래스 변수 또는 전역 변수를 사용하지 않는다.
- 전역변수의 사용을 제한하는 이유는 그것들이 정의된 무비클립이 삭제될 때 가베지 컬렉션되지 않기 때문이다.
- 더 이상 필요 없는 경우엔 변수를 삭제하거나 null 을 넣어준다. 이렇게 하는 것은 가베지 컬렉션을 위한 것이다. 변수를 삭제하는 것은 실행시 최적화에 도움을 준다. 그 이유는 필요 없는 애셋들이 SWF 파일에서 삭제 되었기 때문이다. 따라서 null 을 넣기보다는 삭제하는 것이 더 좋다.
- getter 와 setter 메소드는 오버헤드가 더 크기 때문에 가능하다면 속성에 직접 접근하도록 한다.

3.2 일반 ActionScript 3 지침

AS3 의 실행 속도를 올리려면 다음과 같이 최적화하십시오.

3.2.1 엄격한 데이터 유형 설정

- 항상 변수 또는 클래스 멤버의 데이터 유형을 선언합니다.
- Object 형식의 변수 및 클래스 멤버는 되도록 선언하지 않습니다. Object 는 AS3 에서 가장 포괄적인 데이터 유형입니다. Object 유형의 변수를 선언하는 것은 아무런 의미가 없습니다.
- Array 클래스는 되도록 사용하지 않는 편이 좋습니다. 대신 Vector 클래스를 사용하십시오. Array 의 데이터 구조는 매우 느슨합니다. 인덱스 4294967295(또는 이와 가까운 인덱스)에서 값을 액세스 또는 설정할 필요가 없다면 Array 를 사용하지 않아도 됩니다.
- Vector 를 사용하여 요소의 유형을 지정할 수 있습니다. Vector<*> 유형은 인스턴스에 모든 형식의 데이터를 저장할 수 있는 벡터입니다. Vector<*>와 같은 포괄적인 유형은 되도록 사용하지 안습니다. 대신 Vector 의 요소로 특정 유형을 사용하십시오. 예를 들어, Vector<int>, Vector<String> 또는 Vector<YourFavoriteClass>는 Vector<*>보다 더 효율적일뿐더러 메모리도 훨씬 더 적게 사용합니다.
- Object 를 해시 테이블로 사용하지 않습니다. 대신 flash.utils.Dictionary 클래스를 사용하십시오.
- 동적 클래스와 동적 속성은 되도록 사용하지 않습니다. 현재 GFx 와 Flash 모두 동적 속성에 대한 액세스를 최적화할 수 없습니다.
- 객체의 원형(prototype)을 사용하거나 변경하지 마십시오. 이 원형은 AS2 의 일부이며 AS3 과의 호환성을 유지하기 위해 존재합니다. AS3 에서 정적 함수와 멤버를 사용하면 같은 기능을 구현할 수 있습니다.

3.3 Advance

만약 advance 를 실행하는데 너무 오래 걸린다면 일곱 가지의 최적화 방법이 있다:

1. 매 프레임 마다 AS 코드를 실행하지 않는다. 매 프레임마다 코드호출이 있는 onEnterFrame /Event.ENTER_FRAME 핸들러를 피한다.
2. 이벤트 전달 방식의 프로그래밍을 사용한다. 텍스트 필드와 UI 상태 값을 변경할 때는 변화가 실제로 발생한 경우에 상태를 알리는 호출을 통해서 처리한다.
3. 보이지 않는 무비클립의 애니메이션은 멈춘다. (_visible (AS2) or visible (AS3) 속성을 true 로 설정하고 애니메이션을 멈추는 stop()함수를 사용한다. 이렇게 하면 Advance 리스트에서 이들 무비클립을 제외시킬 것이다. 유념해야 할 것은 부모 무비클립이 멈췄다 하더라도 포함된 모든 자식들의 무비클립 정지시켜야 한다는 것이다.
4. _global.noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)확장 기능 사용에 대한 한가지 대체 기술이 있다. 이 확장기능은 Advance 리스트에서 중지시키지 않고 보이지 않는 무비클립들을 제외시키는데 유용할 것이다. 만약 이 확장 속성이 "true"로 설정 되어있다면 보이지 않는 무비클립들은 Advance 리스트(그들의 자식을 포함해서)에 삽입되지 않으므로 결국 성능이 향상된다. 이 기술이 완전하게 플래시와 호환되는 것이 아님을 유의하라. 플래시 파일은 숨겨진 무비내에서 프레임별 처리를 하는 것에 의존하지 않는다. _global.gfxExtensions (AS2) or scaleform.gfx.Extensions.enabled (AS3) 를 true 로 설정해서 확장기능을 활성화 시켜야 이러한(그리고 다른) 확장기능을 사용할 수 있음을 명심하라.
5. 스테이지상의 무비클립 수를 줄여준다. 중첩된 클립은 advance 수행에 대한 작은 오버헤드를 발생시키므로 불필요한 무비클립의 중첩을 제한한다.
6. 타임라인 애니메이션과 키프레임의 수와 셰입 트윈을 줄여준다.

3.4 onEnterFrame and Event.ENTER_FRAME

onEnterFrame(AS2) or Event.ENTER_FRAME (AS3)이벤트 핸들러의 사용을 최소화 하고, 필요없을 때는 삭제함으로써 항상 사용되지 않도록 한다. 너무 많은 onEnterFrame/Event.ENTER_FRAME 핸들러를 사용하면 성능이 상당히 떨어질지도 모른다. 그의 대안은 setInterval 과 setTimeout 의 사용을 고려해본다. setInterval 을 사용할때:

- 핸들러가 더 이상 필요 없을 때 clearInterval 을 호출하는 것을 잊지 마라.
- setInterval 과 setTimeout 핸들러는 만약 onEnterFrame/Event.ENTER_FRAME 보다 더 자주 실행되어야 한다면 onEnterFrame/Event.ENTER_FRAME 보다 느릴 수도 있다. 이를 피하려면 시간 인터벌을 아껴 써야 한다.

3.4.1 onEnterFrame 정리

onEnterFrame 핸들러를 삭제하려면 delete 연산자를 사용한다:

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

onEnterFrame 을 null 또는 undefined 로 할당 하지마라. (예: this.onEnterFrame = null;), 이 연산자는 onEnterFrame 핸들러를 완전히 삭제 하지 않는다. Scaleform 는 onEnterFrame 가 계속 존재하는 한 이 핸들러를 계속 찾으려 할 것이다.

3.5 ActionScript 2 최적화

3.5.1 이벤트 상의 onClipEvent

onClipEvent()와 on()이벤트의 사용을 피한다. 대신에 onEnterFrame, onPress 등을 사용한다. 그 이유는 다음과 같다.

- 함수 형태의 이벤트 핸들러들은 런타임 도중에 설치와 삭제가 가능하다.

- 함수 내부의 바이트코드는 구식 스타일인 onClipEvent 과 on 보다 더 최적화 되었다. 그 주된 최적화는 this, _global, arguments, super 등의 전캐싱과, 지역변수를 위한 256 개의 내부 레지스터를 사용하는 것이다. 이 최적화 작업은 오직 함수에서만 작동한다.

유일한 문제는 onLoad 스타일 함수 핸들러를 첫번째 프레임 실행전에 설치할 필요가 있을 때 발생한다. 이 경우에는 문서에 없는 이벤트 핸들러인 onClipEvent(construct)를 사용해서 onEnterFrame 을 설치할 수 있다.

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

아니면 onClipEvent(load)를 사용해서 일반함수를 호출하라. 대신 이 경우 함수호출에 추가 오버헤드가 붙기 때문에 덜 효율적이다.

3.5.2 Var 키워드

가능한 한 "var" 키워드를 사용한다. "var" 키워드를 함수 안에서 사용하면 지역변수에 접근할 때 이름이나 해쉬테이블을 사용하지 않고 직접 인덱스를 통해서 내부 레지스터로 접근하기 때문에 컴파일러 최적화에서 매우 중요하다. var 키워드를 사용하면 AS 함수의 실행속도를 두 배 향상시킬 수 있다.

비 최적화 코드

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

최적화 코드:

```
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 전캐싱(Precaching)

전캐싱은 (var 키워드가 있는)지역변수 중에서 읽기전용 오브젝트인 경우 자주 사용된다.

비 최적화 코드

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

최적화 코드:

```
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;
     }
}
```

전캐싱은 다른 경우에도 효과적으로 사용할 수 있다. 다음 예를 보라.

```
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 긴 경로의 전캐싱

다음과 같은 긴 경로의 반복사용을 피하라

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

지역 변수에서 파일 경로의 일부분을 전캐싱

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

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

3.5.5 복잡한 표현

다음과 같이 C 스타일처럼 복잡한 표현은 피하라

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

더 작은 부분으로 이 표현을 쪼개고, 중간 데이터를 지역 변수에 저장하라

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

다음 반복문처럼 분할되는 경우에 다중 참조가 있다면 특히 중요하다.

```
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;
}
```

다음은 앞서의 루프를 개선한 버전이다.

```
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;
}
```

위의 코드는 더욱 최적화가 가능하다. 만약 가능하다면 배열의 동일 요소에 대한 다중 참조를 제거한다. 얻어낸 객체를 지역변수에 전캐싱한다.

```
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 개발

다음 장은 헤드업 디스플레이(HUD)를 생성하고 반복 적용할 때 Scaleform 이 권하는 고수준 예제들이다. 이후에 구현될 리스트가 반드시 필요한 것은 아니다. 그러나 Scaleform 로 HUD 를 생성할 때 더 나은 성능과 최적의 메모리 사용을 위해서라면 훈련을 하기 바란다.

우리의 추천 방식을 복잡도가 증가하는 순서, 그리고 구현하는데 걸리는 시간 순서로 나열하였다. HUD 를 다중으로 구현하려면 일단 4.1 의 방법으로 시작해서 최종버전으로 더 깊이 진행해 나가는 것이 좋겠다. 이것은 게임에 사용될 HUD 요소를 빠르게 프로토타입화 하는데 좋은 방법이다. HUD 와 필요한 리소스는 여기서 추천하는 방법으로 최적화 및 개선된 것이다.

만약 매우 복잡하고 멀티 레이어인 HUD 를 고성능이면서도 적은 메모리로 구현하려면 C++로 개발하는 것이 효율이 높을 것이다.

4.1 다중 SWF 무비뷰

전반적으로 이 HUD 의 방식은 개발과 반복 적용이 매우 빠르다. 오직 아티스트에 의해 제어되고 더 많은 효과와 빠른 시간 안에 더 나은 그래픽적인 표현을 할 수 있도록 해준다. 이 방식은 프로토타입 구현과 최적화 이전의 반복 디자인 시에 최적이지만, 메모리 사용량은 늘어날 수 있다. 각각의 movie view 들은 약 80K 정도의 메모리 오버헤드를 가지면서 새로운 플레이어 인스턴스를 생성한다. 문제는 더 빠른 HUD 와 개별 무비컨트롤에는 더 많은 메모리가 사용된다는 것이다. 가장 먼저 고려해야 할 것은 메모리와 성능에 대한 이슈, 동적인 확장과 다중 레이어 시스템이다.

다중 SWF 무비뷰는 다음과 같은 장점을 제공한다.

1. *서로 다른 쓰레드들에서 Advance 를 호출하는 능력.* 멀티쓰레드 HUD 인터페이스를 사용하면 다른 쓰레드에서 플래시 무비를 실행하거나 Advance 하는 것이 가능하다(타임라인,

애니메이션, AS 실행, 플래시 처리 등처럼 렌더링이 아닌 처리자체). 이렇게 하면 개별적인 advance 제어가 가능하다. 플래시를 여러 개의 무비로 분할하면 개발자가 정지시키고 특정 요소는 Advance 되지 않게 하거나 다른 쓰레드에서 다른 시간에 Advance 를 호출할 수 있다. HUD 의 어떤 요소들은 더 높은 비율로 Advance 되거나 게임에서 특정 액션과 관련된 이벤트가 발생하기 전까지 HUD 의 정적요소의 Advance 를 호출하지 않도록 정지시키는 것이 가능하다.

주의: Scaleform 에서는 다른 GFx::Movie 객체를 다른 쓰레드에서 Advance 호출하는 것이 가능하다. 하지만 각 무비 인스턴스가 쓰레드 세이프 Display 가 아니기 때문에 명시적으로 동기화를 하지 않으면 안 된다. Input 과 Invoke 호출도 Advance 시에 동기화가 필요하다.

2. *렌더투텍스처 캐싱을 활용하라.* 이는 HUD 요소를 텍스처에 렌더링 해서 캐시에 둠으로써 필요 할 때만 업데이트 하는 것이다. 이렇게 하면 DP 를 줄일 수 있다. 물론 텍스처 버퍼를 HUD 요소와 동일한 크기로 준비해야 하기 때문에 메모리가 더 필요하다. 성능상에서는 장점이지만 메모리상에서는 감점인 셈이다. 이 옵션은 요소가 자주 업데이트 되지 않으면서 그다지 복잡하지 않을 때 고려할 만 하다.

4.2 여러 SWF 를 포함한 단일 무비뷰

이 방법은 여러 플래시 파일을 단일한 전체화면 무비뷰에 loadMovie 명령을 사용해서 로드하는 것이다. 이러한 접근의 장점은 메모리 사용의 효율성과 아주 살짝 출력효율도 개선된다는 것이다.

단일 무비뷰에 여러 플래시 파일을 로드 할 때는 다음 지침을 따르라.

1. 일괄 처리시에 숨겨질 수 있는 객체들을 조심스럽게 그룹화 하고 _global.noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3) 는 true 지만_visible = false (AS2) or visible = false (AS3) = false 로 해서 advance 처리시에 오버헤드를 최소화 한다. HUD 를 생성 할 때는 이것이 가장 중요하다. 숨길 수 있는 객체들을 그룹화하고 이들을 관리할 부모를 추가한다. _visible = false 로 해서 객체가 안보이게 되면

처리를 멈추도록 한다(이는 컨트롤의 보이는 속성을 제어하는 것이 아니라 이미 보이지 않는 객체의 Advance 호출을 막는 것이다). 특정 객체 그룹을 숨겨서 플래시 파일 내의 이들 요소에 대한 로직 수행이 호출되지 않도록 한다. 이는 HUD 의 특정 부분이 숨겨지거나 나타날 때(즉, 잠시정지 메뉴, 맵, 체력 바 등) 유용하다. 또한 HUD 전체를 숨기거나 나타낼 때 Advance 를 함께 부르지 말라. _global.gfxExtensions(AS2) or scaleform.gfx.Extensions.enabled (AS3)을 true 로 해서 Scaleform 확장을 켜놓는 것을 잊지 마라.

2. SetVariableArray 를 사용해서 여러 개의 변수 업데이트를 한번의 Invoke 호출로 처리하도록 그룹화 하라. 이는 (이동하는 요소를 가진 맵처럼) 복잡도가 증가할수록 유용하다. SetVariableArray 를 호출해서 (각 맵요소의 새로운 위치값)데이터 배열을 전달하고나서 한번의 Invoke 를 호출해서 아이템들을 이동시키고 업데이트 하기 위한 데이터 배열을 처리하는 것이다. 하지만, 처리할 데이터가 적다면 이 방법이 성능에 부정적일 수 있으므로 사용하지 않도록 한다.
3. HUD 요소를 생성할 때 onEnterFrame(AS2) or Event.ENTER_FRAME (AS3)은 신중하라. HUD 내에 onEnterFrame/Event.ENTER_FRAME 을 가지는 여러 요소가 있으면 개발자가 Advance 를 호출할 때마다 특정 요소가 변경되지 않아도 무조건 실행될 것이다.
4. 무비의 프래임율을 게임의 프래임율의 절반으로 유지하고 필요 할 때만 AS 를 호출하라. 일반적인 게임 프로그래밍 패러다임에서는 게임엔진에서 모든 프레임마다 tick 을 호출한다. 이는 플래시에 최적화된 것이 절대로 아니다. 개발자가 AS Invoke 를 매 프레임마다 호출하는 것을 피함으로써 메모리 사용을 줄이고 성능을 향상시킬 수 있다. 예를 들어서 게임이 30-40FPS 라면 애니메이션을 15-20FPS 로 업데이트 하라. 하지만 HUD 애니메이션에서 랙이 보일 수도 있고 높은 프레임율의 경우에는 애니메이션이 느리게 보이거나 할 수도 있다.
5. 가능하면 타임라인 애니메이션을 줄여라. 긴 타임라인이 더 많은 메모리를 먹는다. 하지만, 너무 짧게 하면 애니메이션이 뚝뚝 끊길 수 있으니 관리에 주의하라.

4.3 단일 무비뷰

이 방식은 (레이더 스크린같이)복잡한 다중요소 인터페이스처럼 매우 효율이 좋은 플래시에

사용하도록 한다. Flash vs C++이 요소를 어떻게 렌더하는지를 주의 깊게 조사하는 것이 중요하다 다중 플래시 레이어는 분할된 DP 가 되며, 성능을 저하시킨다. 4.2 장에 이어서 다음을 추가한다.

1. 하나의 인터페이스 내부 요소들을 게임엔진으로 그려라. 하지만 테두리(Border)와 틀(Frame)은 플래시를 사용하고 텍스트는 Scaleform 를 사용한다. HUD 에 신속한 갱신이 필요 할 때는 플래시로 정적 요소를 그리고, 자주 변화하는 항목에 C++을 사용할 수 있다.
2. 플래시에 의해 그려지는 요소라도 위치 설정은 C++을 사용한다. 이에 대한 좋은 예는 레이더 스크린이다. 점들의 집합으로 만들어진 단일 플래시를 만들고, 점의 위치 값은 Render::Renderer 내에서 RenderString ID 를 사용해서 태그로 관리한다. C++엔진을 사용해서 렌더되기 전에 재배열 시킨다. 이렇게 하면 AS 오버헤드를 피할 수 있으나 상당히 복잡하고 또한 추가적인 프로그래밍도 필요하다.

4.4 단일 무비뷰(고급)

이 방법은 상당히 고급인 반면 시간을 많이 잡아먹는다. 하지만 추가적인 메모리 절약을 제공한다. HUD 생성을 거의 완료하기 전까지는 이 방법을 추천하지 않는다. 4.3 장 외에 다음의 기술들을 추가로 고려하라.

1. HUD 안의 그래픽이 변화가 있을 때만 Advance 를 호출하거나, 하나의 Invoke 로 모든 것을 갱신하도록 한다. 이는 (텍스트가 포함된 진행 바처럼) HUD 애니메이션이 업데이트 되는 배경 레이어를 가진 단일 무비에 사용될 수 있다. 이에 대한 가장 좋은 예는 일반적으로 애니메이션이 없는 체력바다. 변화가 있기 전에는 Advance 가 호출될 이유가 없다.(첫 프레임 실행, 정지, 캐릭터의 체력이 변경되는 Advance/invoke 됨) 이렇게 하면 렌더에 있어서 더욱 효과적이고 CPU 오버헤드가 더 적지만, 더 복잡한 관리가 필요하다.
2. 정점데이터 관리를 위한 사용자 정의(custom) 정적 버퍼를 활용하고, Render::Renderer 를 재지정(override)하라. 이 방법은 C++에 제한된 방법이며 고급 C++그래픽 프로그래머의 중재가 필요한 어려운 작업이다. Render::Renderer 를 재지정할 때는 다른(custom) 비디오 메모리 벡터

저장소를 사용하고, HUD 요소를 관리할 때 동적 버퍼가 아닌 정적 버퍼를 사용하도록 GFx 시스템을 재지정한다.

3. 쓰레드 렌더링을 사용하고 Render::Renderer 를 재지정한다. 이 방법이 아마도 제일 복잡한 방법이다. 렌더링 코드를 재 작성해야 하고 게임엔진에서 렌더되는 HUD 는 각각 Advance 를 호출해야 한다. 이는 얻게 되는 성능향상과 구현의 복잡도 사이에서 결정해야 한다.

   주의: Scaleform-언리얼 3 통합에 쓰레드 렌더링이 존재하지만 실제 구현하기 위해서는 엄청난 프로그래밍 노력이 필요하다.

4.5 플래시 없이 커스텀 HUD 만들기

이 과정은 대단히 복잡하고 시간이 많이 든다. 결과적으로 이 HUD 는 순수한 C++ 과 비트맵 기반이 될 것이다. Scaleform 와 플래시는 HUD 생성과정에서 활용될 수 있지만 최종과정에서 플래시 인터페이스를 비트맵으로 변환하면서 제거된다. 이 시점에서 Scaleform 는 아무런 advance 나 HUD 요소용 메모리를 필요로 하지 않을 것이다. 단지 Scaleform Player 용 메모리가 필요할 뿐이다.

어디까지 감당할 수 있을지는 모르지만 커스텀 튜닝된 C++렌더링 시스템을 Scaleform 렌더링과 혼용할 수 있다. 외부 커스텀 렌더링에 가장 적절한 것은 미니맵, 인벤토리, 상태창처럼 많은 아이템이 있는 것들이다. 이들 영역은 DP 일괄처리와 멀티아이템 업데이트를 통해서 최적화 가능하다. 경계면, 패널, 상태, 애니메이션 팝업은 Scaleform 에서 사용하도록 하다가 병목현상이 생길 경우에만 교체하면 된다.

우리 입장에서는 개발자가 Scaleform 폰트/텍스트 엔진을 계속 사용할 것을 추천하는데, 이는 Scaleform 에 포함된 DrawText API 가 개발자로 하여금 Scaleform 로 생성한 UI 와 동일한 플래시 폰트 및 텍스트 시스템을 C++에서 사용하게 해주기 때문이다. 이렇게 하면 두 개로 분할된 폰트 시스템이 아니므로 메모리가 절약된다. 자세한 폰트/텍스트 관련 내용은 엔진에 FAQ 의 장, "Font Overview"문서, "DrawText API Reference"를 참고하라.

HUD 를 구현할 때는 다음을 항상 명심하라.

- 무비클립의 수를 최소한으로 사용한다. 반드시 필요한 경우만 내포(nest)시킨다.
- 가능한 한 마스크를 사용하지 않되 한두 개정도만 사용한다. 더 많은 정보를 원한다면 FAQ 의 "Graphics Rendering & Special Effects" 섹션을 참고 하라.
- PC 와 Wii™ 에서 마우스와 키보드를 사용불능 상태로 만들라 (다른 콘솔들은 이미 기본적으로 사용불능화 되어있음.) :
    - GFx::Movie::SetMouseCursorCount(0);
    - 사용불능 상태라면, 입력을 받지 않는다.
- 무비클립 아이템의 보이는 상태와 보이지 않는 상태를 그룹화하라. HUD 패널은 noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)와 _visible (AS2) or visible (AS3)= false 를 사용한다.
- 오직 아이템에 변화가 있을 때만 Invoke 를 호출한다. 2 개 이상의 항목들이 (같은 프레임에서) 함께 변화될 때 단일처리로 Invoke 를 한번만 호출한다. (매 프레임마다) 자주 업데이트 되지 않는 아이템은 처리하지 말라
- 비트맵과 그라디언트의 사용이 최적화 되었는지 확인한다. 더 자세한 내용은 FAQ 의 section 2.1 또는 "Art & Assets" 장을 참조하기 바란다.

5 일반적인 최적화 팁

5.1 플래시 타임라인

타임라인 프레임과 레이어는 플래시 저작 환경의 중요한 부분이다. 이 영역은 애셋들이 작업문서에서 결정되고 어디에 위치하는지 보여준다. 타임라인과 라이브러리가 어떻게 설정되었는지에 따라서 전체 FLA 파일의 모든 사용성과 성능에 영향을 준다.

- 프레임 기반 반복(loop)을 사용한다. 프레임 애니메이션은 어플리케이션의 프레임 속도에 의존적이다. 반대로 시간 기반 모델은 FPS 에 묶여있지 않다.
- 필요 없어지는 순간 즉시 프레임 기반 반복을 멈춰라.
- 가능하면 한 프레임 이상으로 복잡한 코드 블록을 흩어놓는다.
- 몇 백 라인의 코드로 된 스크립트는 몇 백 프레임 기반의 트윈으로 구성된 타임라인과 같다.
- 애니메이션/상호작용이 타임라인에서 손쉽게 이루어질 수 있는지 혹은 AS 를 사용해서 모듈화 할 수 있는지 평가한다.
- (Layer 1, Layer 2 같은)기본 레이어 이름은 피한다. 그 이유는 복잡한 파일상에서 작업할 때 에셋들의 위치로 인한 기억이 혼란스러울 수 있기 때문이다.

5.2 일반적인 성능 최적화

- 성능 향상을 위해 변환 결합이란 방법이 있다. 예를 들면, 세가지 변환을 겹치는 대신 하나의 행렬을 수동으로 계산하는 것이다.
- 만약 시간이 흐를수록 느려진다면 메모리누수를 점검하라. 더 이상 필요 없는 것들은 처분하라.
- 제작 시 많은 양의 trace()구문 이나 텍스트필드의 동적인 갱신은 성능에 영향을 끼치므로 피한다. 가능하다면 가끔 갱신하도록 한다. (지속적이기 보단 변화가 일어났을 때만)

- 가능하다면 AS 를 포함한 레이어를 놓고, 프레임 레이블 레이어를 타임라인의 최상위 레이어 스택에 놓는다. 예를 들어서 AS actions 을 포함한 레이어의 이름을 짓는 것은 좋은 예다.
- 서로 다른 레이어에 액션을 넣지 않고 하나의 레이어에 모아놓는다. 이것은 AS 코드의 관리를 간단하게 해줄 것이고 여러 개의 AS 실행 패스에서 발생하는 오버헤드를 제거해 성능이 개선될 것이다.

6 추가문서

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 학습가이드
http://www.adobe.com/devnet/flash/articles/actionscript_guide.html