ADOBE® FLASH® PLATFORM의 성능 최적화

Adobe

마지막 업데이트 2012 년 5 월 22 일

법적 고지 사항

법적 고지 사항은 http://help.adobe.com/ko_KR/legalnotices/index.html을 참조하십시오.

목차

1장: 소개

런타임 코드 실행 기본 사항 1

인지 성능과 실제 성능 비교 2

최적화 목표 설정 3

2장: 메모리 유지

표시 객체 4

프리미티브 유형 4

객체 다시 사용 5

메모리 해제 10

비트맵 사용 12

필터 및 동적 비트맵 언로드 18

직접 밉매핑 18

3D 효과 사용 19

텍스트 객체와 메모리 20

이벤트 모델과 콜백 21

3장: CPU 사용 최소화

Flash Player 10.1의 CPU 사용 관련 향상 기능 22

절전 모드 24

객체 표시 제거 및 표시 제거 취소 25

이벤트 활성화 및 비활성화 28

마우스 상호 작용 29

타이머와 ENTER_FRAME 이벤트 29

트위닝 신드롬 31

4장: ActionScript 3.0 성능

Vector 클래스와 Array 클래스 32

드로잉 API 33

이벤트 캡처 및 버블링 34

픽셀을 사용한 작업 35

일반 표현식 37

기타 최적화 37

5장: 렌더링 성능

다시 그리기 영역 42

오프 스테이지 내용 43

동영상 품질 44

알파 블렌딩 46

응용 프로그램 프레임 속도 47

비트맵 캐싱 48
수동 비트맵 캐싱 54
텍스트 객체 렌더링 59
GPU 63
비동기 작업 65
투명 윈도우 67
벡터 모양 매끄럽게 하기 67

6장: 네트워크 상호 작용 최적화

네트워크 상호 작용 향상 기능 69
외부 내용 70
입출력 오류 73
Flash Remoting 74
불필요한 네트워크 작업 75

7장: 미디어를 사용한 작업

비디오 76
StageVideo 76
오디오 76

8장: SQL 데이터베이스 성능

데이터베이스 성능을 위한 응용 프로그램 디자인 78
데이터베이스 파일 최적화 80
불필요한 데이터베이스 런타임 처리 81
효율적인 SQL 구문 81
SQL 문 성능 82

9장: 벤치마킹 및 배포

벤치마킹 83
배포 84

1장: 소개

Adobe® AIR® 및 Adobe® Flash® Player 응용 프로그램은 데스크톱, 휴대 장치, 태블릿, 텔레비전 장치를 비롯해 다양한 플랫폼에서 실행됩니다. 이 문서에서는 코드 예제 및 사용 예를 통해 이러한 응용 프로그램을 배포하는 개발자를 위한 최상의 방법을 설명합니다. 다음과 같은 내용이 포함됩니다.

- 메모리 유지
- CPU 사용 최소화
- ActionScript 3.0 성능 개선
- 렌더링 속도 개선
- 네트워크 상호 작용 최적화
- 오디오 및 비디오를 사용한 작업
- SQL 데이터베이스 성능 최적화
- 응용 프로그램 벤치마킹 및 배포

이러한 최적화의 대부분은 모든 장치의 응용 프로그램에 적용되며, AIR 런타임 및 Flash Player 런타임 모두에서 사용할 수 있습니다. 특정 장치에 적용되는 추가 사항과 예외 역시 설명되어 있습니다.

이러한 최적화의 일부는 Flash Player 10.1 및 AIR 2.5에 추가된 기능을 개선하는 데 주력하지만, 대부분의 최적화는 이전 AIR 및 Flash Player 릴리스에도 적용됩니다.

런타임 코드 실행 기본 사항

응용 프로그램 성능 향상을 위한 방법을 이해하는 데 있어서 한 가지 핵심은 Flash Platform 런타임에서 코드가 실행되는 방식을 이해하는 것입니다. 런타임은 각 "프레임"에서 발생하는 특정 작업이 포함된 루프로 작동합니다. 여기서 프레임이란 간단히 말해서 응용 프로그램에서 지정된 프레임 속도로 결정되는 시간 블록을 의미합니다. 프레임 속도는 각 프레임에 할당된 시간을 말합니다. 예를 들어, 프레임 속도를 초당 30프레임으로 지정한 경우, 런타임에서 각 프레임의 길이는 1/30초가 됩니다.

응용 프로그램의 초기 프레임 속도는 제작 시 지정합니다. Adobe® Flash® Builder™ 또는 Flash Professional의 설정을 사용하여 프레임 속도를 설정할 수 있습니다. 또한 코드에서 초기 프레임 속도를 지정할 수도 있습니다. ActionScript 전용 응용 프로그램의 경우 [SWF(frameRate="24"] 메타데이터 태그를 루트 문서 클래스에 적용하여 프레임 속도를 설정합니다. MXML의 경우 응용 프로그램에서 frameRate 특성을 설정하거나 WindowedApplication 태그를 설정합니다.

각 프레임 루프는 두 개의 단계로 구성되며, 이 두 단계는 이벤트, enterFrame 이벤트 및 렌더링의 세 가지 부분으로 나뉩니다.

첫 번째 단계에는 두 부분(이벤트 및 enterFrame 이벤트)이 포함되며, 두 부분 모두 코드 호출을 발생시킬 수 있습니다. 첫 번째 단계의 첫 번째 부분에서는 런타임 이벤트가 수신되고 전달됩니다. 이러한 이벤트는 네트워크를 통한 데이터 로드 작업으로부터의 응답과 같은 비동기 작업의 완료 또는 진행률을 나타낼 수 있습니다. 여기에는 또한 사용자 입력을 통한 이벤트도 포함됩니다. 이벤트가 전달되면 런타임에서 사용자가 등록한 리스너에 있는 코드를 실행합니다. 이벤트가 발생하지 않으면 런타임에서 다른 액션을 수행하지 않고 이 실행 단계가 완료될 때까지 기다립니다. 작업이 없다고 해서 런타임에서 프레임 속도가 빨라지지는 않습니다. 다른 부분의 실행 주기 중에 이벤트가 발생하면 런타임이 이러한 이벤트를 대기열에 추가하고 다음 프레임에서 이를 전달합니다.

첫 번째 단계의 두 번째 부분은 enterFrame 이벤트입니다. 이 이벤트는 항상 프레임마다 한 번씩 전달되기 때문에 다른 이벤트와 구분됩니다.

모든 이벤트가 전달되면 프레임 루프의 렌더링 단계가 시작됩니다. 이때 런타임에서는 화면에 표시되는 모든 가시 요소의 상태를 계산하고 요소들을 화면에 그립니다. 그런 다음 경주장을 도는 선수들처럼 이러한 프로세스가 계속 반복됩니다.

참고: updateAfterEvent 속성을 포함하는 이벤트의 경우 렌더링 단계를 기다리지 않고 즉시 강제로 렌더링을 수행할 수 있습니다. 그러나 이로 인해 성능 문제가 자주 발생하면 updateAfterEvent를 사용하지 마십시오.

프레임 루프의 두 단계에 각각 동일한 시간이 소요된다고 생각하기 쉽습니다. 이 경우 각 프레임 루프의 절반 동안에는 이벤트 핸들러와 응용 프로그램 코드가 실행되고, 나머지 절반 동안에는 렌더링이 수행됩니다. 그러나 실제로는 이와 다른 경우가 많습니다. 때로는 응용 프로그램 코드에서 프레임이 사용할 수 있는 시간이 절반 이상 사용되어 할당된 시간이 더 많이 소요되고 렌더링에 사용할 수 있는 할당 시간이 줄어들 수 있습니다. 또한 필터 및 블렌드 모드와 같이 시각적인 내용이 복잡할 경우에는 렌더링에 필요한 시간이 프레임 시간의 절반 이상일 수 있습니다. 이러한 단계에서 소요되는 실제 시간은 유동적이기 때문에 프레임 루프를 일반적으로 "탄력적 경주장(elastic racetrack)"이라고도 부릅니다.

프레임 루프에서 조합된 작업(코드 실행과 렌더링)의 시간이 너무 오래 걸릴 경우에는 런타임이 프레임 속도를 유지할 수 없습니다. 이 경우에는 할당된 시간보다 많은 시간이 소요되어 프레임이 늘어나므로 다음 프레임이 트리거되기 전에 지연이 발생합니다. 예를 들어 프레임 루프에 1/30초보다 긴 시간이 소요되면 런타임에서 초당 30프레임의 속도로 화면을 업데이트할 수 없습니다. 프레임 속도가 느려지면 성능이 저하됩니다. 그러면 애니메이션 장면이 불규칙하게 전환될 수 있습니다. 심한 경우에는 응용 프로그램이 멈추고 윈도우에 아무 것도 표시되지 않게 됩니다.

Flash Platform 런타임 코드 실행 및 렌더링 모델에 대한 자세한 내용은 다음과 같은 리소스를 참조하십시오.

- Flash Player Mental Model - The Elastic Racetrack(Ted Patrick의 문서)
- Asynchronous ActionScript Execution(Trevor McCauley의 문서)
- http://www.adobe.com/go/learn_fp_air_perf_tv_kr(Sean Christmann의 MAX 컨퍼런스 프레젠테이션 비디오)의 코드 실행, 메모리 및 렌더링을 위한 Adobe AIR 최적화

인지 성능과 실제 성능 비교

응용 프로그램의 수행 성능이 적합한지 여부에 대한 판단은 궁극적으로 응용 프로그램 사용자에게 달려 있습니다. 개발자는 특정 응용 프로그램이 실행되는 데 걸리는 시간이나 생성되는 객체의 인스턴스 수를 기준으로 응용 프로그램의 성능을 측정할 수 있습니다. 하지만 이러한 측정 기준은 최종 사용자에게 중요한 것이 아닙니다. 사용자가 성능을 측정하는 기준이 다른 경우도 있습니다. 예를 들어 응용 프로그램이 빠르고 원활하게 작동하고 입력에 신속하게 응답하는가, 시스템 성능에 부정적인 영향을 주는가 등을 기준으로 삼을 수 있습니다. 인지 성능을 테스트해 볼 수 있는 다음과 같은 질문에 답해 보십시오.

- 애니메이션이 부드럽게 전환됩니까? 아니면 끊어져서 보입니까?
- 비디오 내용이 부드럽게 보입니까? 아니면 끊어져서 보입니까?
- 오디오 클립이 연속으로 재생됩니까? 아니면 멈췄다가 다시 재생됩니까?
- 프로그램을 오랫동안 실행하면 윈도우가 깜박이거나 아무 것도 표시되지 않습니까?
- 내용을 입력할 때 입력하는 텍스트가 바로 입력됩니까? 아니면 입력 시간이 지연됩니까?
- 항목을 클릭하면 바로 무언가 실행됩니까? 아니면 클릭 후 시간이 지연됩니까?
- 응용 프로그램을 실행하면 CPU 팬이 시끄럽게 돌아갑니까?
- 랩톱 컴퓨터 또는 휴대 장치에서 응용 프로그램을 실행하면 배터리가 빨리 소모됩니까?
- 응용 프로그램을 실행하면 다른 응용 프로그램의 응답이 느려집니까?

인지 성능과 실제 성능을 구분하는 것이 중요합니다. 즉, 최상의 인지 성능을 구현하는 방법이 절대적으로 가장 빠른 성능을 구현하는 방법과 항상 같지는 않습니다. 런타임에서 자주 화면을 업데이트하고 사용자 입력을 수집할 수 없을 만큼 많은 코드를 응용 프로그램에서 실행하지 않도록 합니다. 일부 경우에는 이러한 균형점을 찾기 위해 프로그램 작업을 여러 부분으로 분할하고 각 부분 사이에 런타임이 화면을 업데이트하도록 만듭니다. 이에 대한 자세한 내용은 42페이지의 “렌더링 성능”을 참조하십시오.

여기에 설명된 팁과 기술은 실제 코드 실행 성능과 사용자의 인지 성능을 모두 향상시키기 위한 것입니다.

최적화 목표 설정

일부 성능 향상의 경우에는 사용자가 성능 향상을 느낄 수 없습니다. 따라서 특정 응용 프로그램에서 문제가 되는 영역에 집중하여 성능을 최적화하는 것이 중요합니다. 일부 성능 최적화는 일반적으로 유용한 방법이므로 항상 준수하는 것이 좋습니다. 다른 최적화의 경우 응용 프로그램의 요구 사항 및 예상 사용자층에 따라 유용할 수도 있고 유용하지 않을 수도 있습니다. 예를 들어 애니메이션, 비디오 또는 그래픽 필터 및 효과를 사용하지 않으면 응용 프로그램의 성능은 항상 향상됩니다. 그러나 Flash Platform을 사용하여 응용 프로그램을 만드는 이유 중 하나는 표현력이 뛰어난 응용 프로그램을 만들 수 있는 미디어 및 그래픽 성능 때문입니다. 원하는 표현 수준이 응용 프로그램을 실행하는 컴퓨터 및 장치의 성능 특징과 잘 맞는지 여부를 고려하십시오.

이를 위한 한 가지 조언은 "최적화를 너무 빨리 시작하지 말라"는 것입니다. 성능 최적화를 위해서는 이해하기 어렵고 유연성도 떨어지는 방식으로 코드를 작성해야 할 수 있습니다. 최적화가 된 다음에도 이러한 코드는 유지 관리하기가 더 어려울 수 있습니다. 이러한 최적화의 경우 조금 더 기다려 특정 코드 부분의 성능이 떨어지는지 여부를 확인한 후 해당 코드를 최적화하는 것이 좋습니다.

성능 향상을 위해서는 종종 다른 무언가를 포기해야 합니다. 이상적으로는 응용 프로그램에서 사용하는 메모리의 양을 줄이면 응용 프로그램의 작업 수행 속도도 향상됩니다. 하지만 항상 이렇게 이상적인 결과만 얻을 수 있는 것은 아닙니다. 예를 들어, 작업 중에 응용 프로그램이 중단될 경우, 이를 해결하기 위해서는 여러 프레임에 걸쳐서 실행되도록 작업을 분할해야 할 수 있습니다. 작업이 분할되기 때문에 프로세스를 완료하는 데 전체적으로 더 긴 시간이 소요될 수 있습니다. 그러나 응용 프로그램이 계속 입력에 응답하고 중단되지 않으면 사용자가 추가적인 시간을 느끼지 못할 수 있습니다.

최적화할 대상을 확인하고 최적화가 유용한지 여부를 판단하기 위한 한 가지 핵심적인 방법은 성능 테스트를 수행하는 것입니다. 성능 테스트를 위한 몇 가지 기술 및 팁은 83페이지의 “벤치마킹 및 배포”에 설명되어 있습니다.

최적화 대상으로 적합한 응용 프로그램 부분을 확인하는 방법에 대한 자세한 내용은 다음 리소스를 참조하십시오.

- http://www.adobe.com/go/learn_fp_goldman_tv_kr(Oliver Goldman의 MAX 컨퍼런스 프레젠테이션 비디오)의 AIR용 성능 조정 응용 프로그램
- http://www.adobe.com/go/learn_fp_air_perf_devnet_kr(프레젠테이션에 기초한 Oliver Goldman의 Adobe Developer Connection 문서)

2장: 메모리 유지

메모리 유지는 응용 프로그램 개발에서 항상 중요하며 데스크톱 응용 프로그램에서도 마찬가지입니다. 그러나 휴대 장치에서는 메모리 사용이 특히 중요하므로 응용 프로그램에서 사용하는 메모리의 양을 제한하는 것이 좋습니다.

표시 객체


적절한 표시 객체를 선택합니다.

ActionScript 3.0에는 많은 표시 객체 집합이 포함되어 있습니다. 메모리 사용을 제한하기 위한 가장 간단한 최적화 방법 중 하나는 적절한 유형의 표시 객체를 사용하는 것입니다. 대화형이 아닌 간단한 모양의 경우 Shape 객체를 사용하고, 타임라인이 필요하지 않는 대화형 객체의 경우 Sprite 객체를 사용하며, 타임라인을 사용하는 애니메이션의 경우 MovieClip 객체를 사용합니다. 언제나 응용 프로그램에 가장 효율적인 유형의 객체를 선택하십시오.

다음 코드는 여러 가지 표시 객체의 메모리 사용을 보여 줍니다.

```
trace(getSize(new Shape()));
// output: 236

trace(getSize(new Sprite()));
// output: 412

trace(getSize(new MovieClip()));
// output: 440
```

getSize() 메서드는 객체가 사용하는 메모리 바이트 수를 보여 줍니다. MovieClip 객체의 기능이 필요하지 않은 경우 단순한 Shape 객체 대신에 여러 MovieClip 객체를 사용하면 메모리가 낭비될 수 있습니다.

프리미티브 유형


getSize() 메서드를 사용하여 코드를 벤치마킹하고 해당 작업에 가장 효율적인 객체를 결정합니다.

String을 제외한 모든 프리미티브 유형은 4 - 8바이트의 메모리를 사용합니다. 프리미티브에 특정한 유형을 사용하여 메모리를 최적화하는 방법은 없습니다.

```
// Primitive types
var a:Number;
trace(getSize(a));
// output: 8

var b:int;
trace(getSize(b));
// output: 4

var c:uint;
trace(getSize(c));
// output: 4

var d:Boolean;
trace(getSize(d));
// output: 4

var e:String;
trace(getSize(e));
// output: 4
```

64비트 값을 나타내는 Number에 값이 할당되지 않은 경우 AVM(ActionScript Virtual Machine)에서 8바이트를 할당합니다. 다른 모든 프리미티브 유형은 4바이트로 저장됩니다.

```
// Primitive types
var a:Number = 8;
trace(getSize(a));
// output: 4

a = Number.MAX_VALUE;
trace(getSize(a));
// output: 8
```

String 유형의 비헤이비어는 다릅니다. String의 길이에 따라 저장소의 크기가 할당됩니다.

```
var name:String;
trace(getSize(name));
// output: 4

name = "";
trace(getSize(name));
// output: 24

name = "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the
industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and
scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into
electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s with the release
of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like
Aldus PageMaker including versions of Lorem Ipsum.";
trace(getSize(name));
// output: 1172
```

getSize() 메서드를 사용하여 코드를 벤치마킹하고 해당 작업에 가장 효율적인 객체를 결정합니다.

객체 다시 사용


가능하면 객체를 다시 만들지 않고 재사용합니다.

메모리를 최적화하는 또 다른 방법은 가능하면 항상 객체를 재사용하고 다시 만들지 않는 것입니다. 예를 들어 루프에서 다음 코드를 사용하지 않습니다.

```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

var area:Rectangle;

for (var:int = 0; i < MAX_NUM; i++)
{
    // Do not use the following code
    area = new Rectangle(i,0,1,10);
    myBitmapData.fillRect(area,COLOR);
}
```

루프를 반복할 때마다 Rectangle 객체를 다시 만들면 반복할 때마다 새 객체가 만들어지므로 메모리 사용량이 늘어나고 속도가 느려집니다. 다음 방법을 사용합니다.

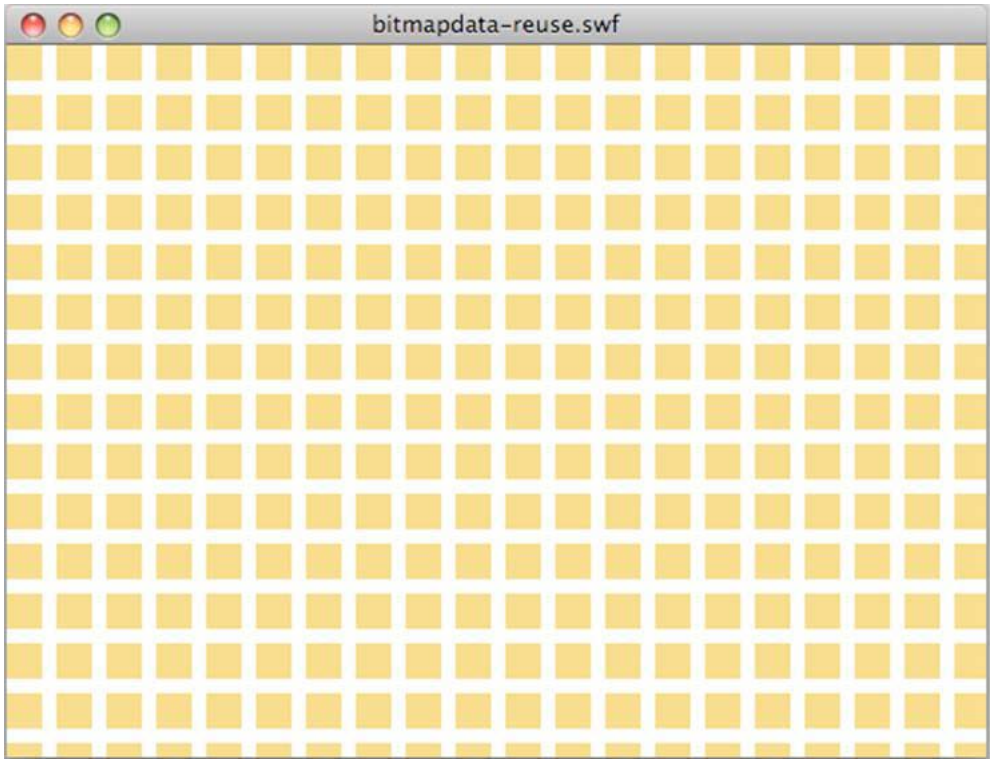
```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

// Create the rectangle outside the loop
var area:Rectangle = new Rectangle(0,0,1,10);

for (var:int = 0; i < MAX_NUM; i++)
{
    area.x = i;
    myBitmapData.fillRect(area,COLOR);
}
```

이전 예제에서는 상대적으로 메모리 영향이 적은 객체를 사용했습니다. 다음 예제에서는 BitmapData 객체를 재사용하여 메모리 사용을 크게 절약하는 것을 보여 줍니다. 바둑판 효과를 만드는 다음 코드는 메모리를 낭비합니다.

```
var myImage:BitmapData;
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a 20 x 20 pixel bitmap, non-transparent
    myImage = new BitmapData(20,20,false,0xF0D062);

    // Create a container for each BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

참고: 양수 값을 사용하고 반올림된 값을 int로 형 변환하는 것이 Math.floor() 메서드를 사용하는 것보다 훨씬 빠릅니다.

다음 이미지는 비트맵 바둑판의 결과를 보여 줍니다.



비트맵 바둑판의 결과

최적화된 버전에서는 여러 Bitmap 인스턴스에서 참조하는 단일 BitmapData 인스턴스를 만들고 같은 결과를 생성합니다.

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

이 방법은 약 700KB의 메모리를 절약하며, 이는 일반적인 휴대 장치에서 상당히 큰 절약 수치입니다. 각 비트맵 컨테이너는 Bitmap 속성을 사용하여 원래 BitmapData 인스턴스를 변경하지 않고도 조작할 수 있습니다.

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the DisplayList
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);

    // Set a specific rotation, alpha, and depth
    myContainer.rotation = Math.random()*360;
    myContainer.alpha = Math.random();
    myContainer.scaleX = myContainer.scaleY = Math.random();
}
```

다음 이미지는 비트맵 변환의 결과를 보여 줍니다.

비트맵 변환의 결과

기타 도움말 항목

48페이지의 “비트맵 캐싱”

객체 풀링


가능한 경우 객체 풀링을 사용합니다.

또 다른 중요한 최적화 방법은 시간 경과에 따라 객체를 다시 사용하는 것과 관련된 객체 풀링 기술입니다. 응용 프로그램 초기화 동안 정의된 수의 객체를 만들어 Array 또는 Vector 객체 등의 풀 내부에 저장합니다. 객체 사용을 마치면 객체가 CPU 리소스를 사용하지 않도록 객체를 비활성화하고 상호 참조를 모두 제거합니다. 그러나 참조를 null로 설정하지는 않습니다. null로 설정하면 객체가 가비지 수집 대상이 될 수 있습니다. 풀에 다시 넣으면 새 객체가 필요할 때 가져올 수 있습니다.

객체를 다시 사용하면 리소스를 많이 사용하는 객체 인스턴스화 작업을 수행해야 할 필요성이 줄어듭니다. 또한 응용 프로그램의 속도를 저하시킬 수 있는 가비지 수집기 실행 빈도도 줄일 수 있습니다. 다음 코드는 객체 풀링 기술을 보여 줍니다.

```
package
{
    import flash.display.Sprite;

    public final class SpritePool
    {
        private static var MAX_VALUE:uint;
        private static var GROWTH_VALUE:uint;
        private static var counter:uint;
        private static var pool:Vector.<Sprite>;
        private static var currentSprite:Sprite;

        public static function initialize( maxPoolSize:uint, growthValue:uint ):void
        {
            MAX_VALUE = maxPoolSize;
            GROWTH_VALUE = growthValue;
            counter = maxPoolSize;

            var i:uint = maxPoolSize;

            pool = new Vector.<Sprite>(MAX_VALUE);
            while( --i > -1 )
                pool[i] = new Sprite();
        }

        public static function getSprite():Sprite
        {
            if ( counter > 0 )
                return currentSprite = pool[--counter];

            var i:uint = GROWTH_VALUE;
            while( --i > -1 )
                    pool.unshift ( new Sprite() );
            counter = GROWTH_VALUE;
            return getSprite();

        }

        public static function disposeSprite(disposedSprite:Sprite):void
        {
            pool[counter++] = disposedSprite;
        }
    }
}
```

SpritePool 클래스는 응용 프로그램 초기화 시 새로운 객체의 풀을 만듭니다. getSprite() 메서드는 이러한 객체의 인스턴스를 반환하고 disposeSprite() 메서드는 이들을 해제합니다. 이 코드에서는 풀이 완전히 사용되었을 때 풀을 확장할 수 있습니다. 또한 풀이 소모되었을 때 새 객체를 할당하지 않는 고정된 크기의 풀을 만들 수도 있습니다. 가능하면 루프에서 새 객체를 만들지 않는 것이 좋습니다. 자세한 내용은 10페이지의 “메모리 해제”를 참조하십시오. 다음 코드에서는 SpritePool 클래스를 사용하여 새 인스턴스를 가져옵니다.

```
const MAX_SPRITES:uint = 100;
const GROWTH_VALUE:uint = MAX_SPRITES >> 1;
const MAX_NUM:uint = 10;

SpritePool.initialize ( MAX_SPRITES,  GROWTH_VALUE );

var currentSprite:Sprite;
var container:Sprite = SpritePool.getSprite();

addChild ( container );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    for ( var j:int = 0; j< MAX_NUM; j++ )
    {
        currentSprite = SpritePool.getSprite();
        currentSprite.graphics.beginFill ( 0x990000 );
        currentSprite.graphics.drawCircle ( 10, 10, 10 );
        currentSprite.x = j * (currentSprite.width + 5);
        currentSprite.y = i * (currentSprite.width + 5);
        container.addChild ( currentSprite );
    }
}
```

다음 코드에서는 마우스 클릭 시 표시 목록에서 모든 표시 객체를 제거하고 나중에 다른 작업에서 이들 객체를 다시 사용합니다.

```
stage.addEventListener ( MouseEvent.CLICK, removeDots );

function removeDots ( e:MouseEvent ):void
{
    while (container.numChildren > 0 )
        SpritePool.disposeSprite (container.removeChildAt(0) as Sprite );
}
```

참고: 풀 벡터는 항상 Sprite 객체를 참조합니다. 객체를 메모리에서 완전히 제거하려면 남아 있는 모든 참조를 제거하는 SpritePool 클래스의 dispose() 메서드가 필요합니다.

메모리 해제

객체에 대한 모든 참조를 삭제하여 가비지 수집기가 트리거되도록 합니다.

Flash Player의 릴리스 버전에서 직접 가비지 수집기를 실행할 수 없습니다. 객체가 가비지 수집되도록 하려면 해당 객체에 대한 모든 참조를 삭제합니다. ActionScript 1.0 및 2.0에서 사용되던 이전의 delete 연산자는 ActionScript 3.0에서 다르게 작동합니다. ActionScript 3.0에서는 동적 객체의 동적 속성을 삭제하기 위해서만 사용할 수 있습니다.

참고: Adobe® AIR® 및 Flash Player의 디버그 버전에서 가비지 수집기를 직접 호출할 수 있습니다.

예를 들어 다음 코드는 Sprite 참조를 null로 설정합니다.

```
var mySprite:Sprite = new Sprite();

// Set the reference to null, so that the garbage collector removes
// it from memory
mySprite = null;
```

객체를 null로 설정해도 메모리에서 객체가 반드시 제거되지는 않습니다. 사용 가능한 메모리가 부족한 것으로 간주되지 않는 경우 가비지 수집기가 실행되지 않기도 합니다. 가비지 수집은 예측 가능하지 않습니다. 가비지 수집은 객체를 삭제할 때가 아니라 메모리를 할당할 때 트리거됩니다. 가비지 수집기가 실행되면 아직 수집되지 않은 객체의 그래프를 찾습니다. 가비지 수집기는 그래프에서 서로 참조하지만 응용 프로그램이 더 이상 사용하지 않는 객체를 찾아 비활성 객체를 검색합니다. 이렇게 검색된 비활성 객체는 삭제됩니다.

대용량 응용 프로그램에서 이러한 프로세스는 CPU를 많이 사용할 수 있으며 성능에 영향을 주고 응용 프로그램의 속도를 크게 저하시킬 수 있습니다. 따라서 가능한 한 많은 객체를 재사용하여 가비지 수집 전달을 제한하는 것이 좋습니다. 또한 가능하면 참조를 null로 설정하여 가비지 수집기에서 객체를 찾는 데 소요되는 처리 시간이 단축되도록 합니다. 가비지 수집은 일종의 보험이라고 생각하고 가능하면 항상 객체 수명을 확실하게 관리해야 합니다.

참고: 표시 객체에 대한 참조를 null로 설정하더라도 해당 객체가 표시 제거되지 않을 수 있습니다. 해당 객체는 가비지 수집될 때까지 계속 CPU 주기를 사용합니다. 객체의 참조를 null로 설정하기 전에 해당 객체를 올바르게 비활성화했는지 확인하십시오.

가비지 수집기는 Adobe AIR 및 Flash Player의 디버그 버전에서 제공되는 System.gc() 메서드를 사용하여 실행할 수 있습니다. 또한 Adobe® Flash® Builder와 함께 번들로 제공되는 프로파일러를 통해 수동으로 시작할 수도 있습니다. 가비지 수집기를 실행하면 응용 프로그램이 응답하는 방식과 객체가 메모리에서 제대로 삭제되었는지 여부를 확인할 수 있습니다.

참고: 객체가 이벤트 리스너로 사용된 경우 다른 객체가 해당 객체를 참조할 수 있습니다. 이 경우 참조를 null로 설정하기 전에 removeEventListener() 메서드를 사용하여 이벤트 리스너를 제거합니다.

다행히 비트맵에 사용되는 메모리 양은 즉시 줄일 수 있습니다. 예를 들어 BitmapData 클래스에는 dispose() 메서드가 포함되어 있습니다. 다음 예제에서는 1.8MB의 BitmapData 인스턴스를 만듭니다. 현재 사용 중인 메모리가 1.8MB로 증가하고 System.totalMemory 속성이 더 작은 값을 반환합니다.

```
trace(System.totalMemory / 1024);
// output: 43100

// Create a BitmapData instance
var image:BitmapData = new BitmapData(800, 600);

trace(System.totalMemory / 1024);
// output: 44964
```

다음으로 BitmapData가 메모리에서 수동으로 제거, 즉 삭제되며 메모리 사용이 다시 한 번 검사됩니다.

```
trace(System.totalMemory / 1024);
// output: 43100

// Create a BitmapData instance
var image:BitmapData = new BitmapData(800, 600);

trace(System.totalMemory / 1024);
// output: 44964

image.dispose();
image = null;

trace(System.totalMemory / 1024);
// output: 43084
```

dispose() 메서드가 메모리에서 픽셀을 제거하지만 참조를 계속 null로 설정해야 메모리가 완전히 해제됩니다. BitmapData 객체가 더 이상 필요하지 않을 때 메모리가 즉시 해제될 수 있도록 항상 dispose()를 호출하고 참조를 null로 설정합니다.

참고: Flash Player 10.1 및 AIR 1.5.2에는 System 클래스에 disposeXML()이라는 새로운 메서드가 도입되었습니다. 이 메서드를 사용하면 XML 트리를 매개 변수로 전달하여 XML 객체를 가비지 수집의 대상으로 즉시 설정할 수 있습니다.

기타 도움말 항목

25페이지의 "객체 표시 제거 및 표시 제거 취소"

비트맵 사용

비트맵 대신에 벡터를 사용하는 것은 메모리를 절약할 수 있는 좋은 방법입니다. 그러나 벡터를 사용하면(특히 많이 사용할 경우) 필요한 CPU 또는 GPU 리소스 양이 크게 증가합니다. 비트맵 사용은 렌더링을 최적화하는 유용한 방법입니다. 런타임에서 벡터 내용을 렌더링할 때보다 화면에 픽셀을 그릴 때 처리 리소스가 더 적게 필요하기 때문입니다.

기타 도움말 항목

54페이지의 "수동 비트맵 캐싱"

비트맵 다운샘플링

Flash Player는 16비트 화면을 감지할 경우 메모리를 보다 효율적으로 사용하기 위해 32비트 불투명 이미지를 16비트 이미지로 줄입니다. 이처럼 다운샘플링을 활용하면 메모리 리소스가 절반만 사용되고 이미지는 더 빨리 렌더링됩니다. 이 기능은 Windows Mobile용 Flash Player 10.1에서만 사용할 수 있습니다.

참고: Flash Player 10.1 이전에는 메모리에 만들어진 모든 픽셀이 32비트(4바이트)로 저장되었습니다. 300 x 300 픽셀의 간단한 로고에 350KB의 메모리(300*300*4/1024)가 사용되었습니다. 이제는 이 새로운 비헤이비어를 통해 동일한 불투명 로고에 175KB만 사용됩니다. 로고가 투명할 경우에는 16비트로 다운샘플링되지 않고 메모리에서 같은 크기로 유지됩니다. 이 기능은 포함된 비트맵 또는 런타임에 로드되는 이미지(PNG, GIF, JPG)에만 적용됩니다.

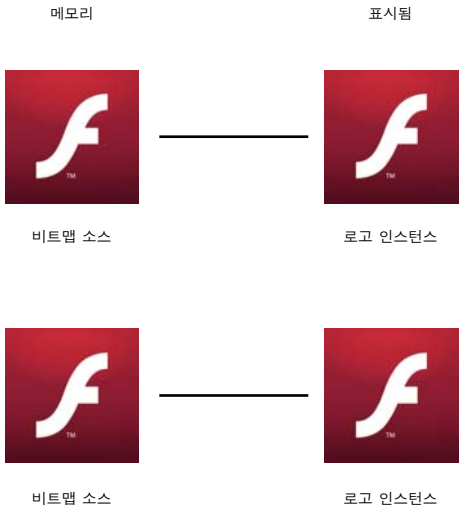
휴대 장치에서는 16비트로 렌더링된 이미지와 32비트로 렌더링된 이미지 간의 차이를 구분하기가 어려우며, 몇 가지 색상만 포함된 간단한 이미지의 경우에는 어떤 차이도 느끼기 어렵습니다. 좀 더 복잡한 이미지인 경우에도 쉽게 그 차이를 느낄 수 없습니다. 하지만 이미지를 확대하면 색상 품질이 약간 저하될 수 있으며 16비트 그래디언트가 32비트 버전보다 덜 매끄럽게 보일 수 있습니다.

BitmapData 단일 참조

가능한 한 많은 인스턴스를 다시 사용하여 BitmapData 클래스의 사용을 최적화하는 것이 중요합니다. Flash Player 10.1 및 AIR 2.5에는 모든 플랫폼에서 사용 가능한 BitmapData 단일 참조라는 새 기능이 추가되었습니다. 포함된 이미지에서 BitmapData 인스턴스를 만들 때 모든 BitmapData 인스턴스에 대해 단일 버전의 비트맵이 사용됩니다. 나중에 비트맵이 수정되면 메모리에 고유한 비트맵이 제공됩니다. 포함된 이미지는 라이브러리 또는 [Embed] 태그에서 가져올 수 있습니다.

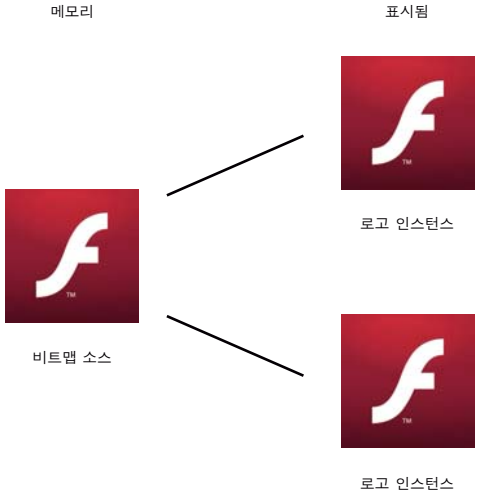
참고: Flash Player 10.1 및 AIR 2.5에서는 비트맵을 자동으로 재사용하기 때문에 기존 내용도 이 새로운 기능으로 인해 이점을 얻을 수 있습니다.

포함된 이미지를 인스턴스화할 때 연결된 비트맵이 메모리에 만들어집니다. Flash Player 10.1 및 AIR 2.5 이전에는 다음 다이어그램에서처럼 인스턴스별로 각각의 비트맵이 메모리에 제공되었습니다.



Flash Player 10.1 및 AIR 2.5 이전의 경우 메모리의 비트맵

Flash Player 10.1 및 AIR 2.5에서는 같은 이미지에 대한 인스턴스를 여러 개 만들 경우 단일 버전의 비트맵이 모든 BitmapData 인스턴스에 사용됩니다. 다음 다이어그램에서는 이러한 개념을 보여 줍니다.

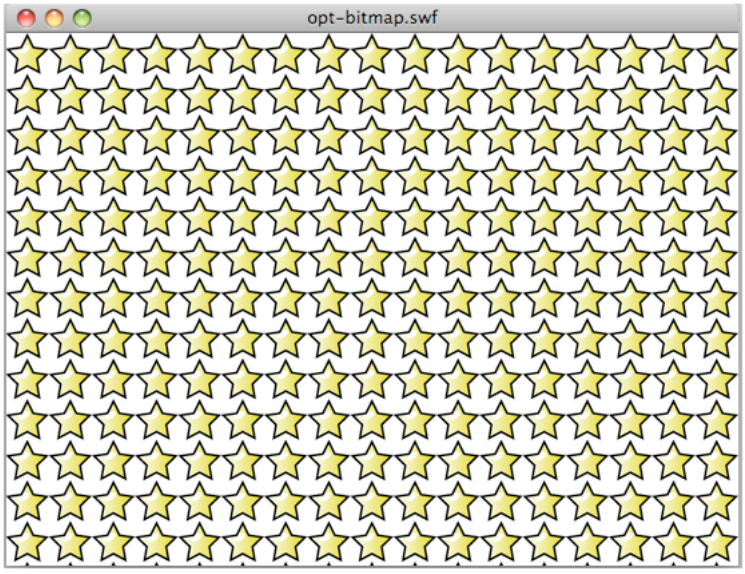


Flash Player 10.1 및 AIR 2.5의 경우 메모리의 비트맵

이 방법은 비트맵이 많은 응용 프로그램에서 사용하는 메모리의 양을 크게 줄여줍니다. 다음 코드는 Star 기호에 대한 인스턴스를 여러 개 만듭니다.

```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}
```

다음 이미지는 코드의 결과를 보여 줍니다.



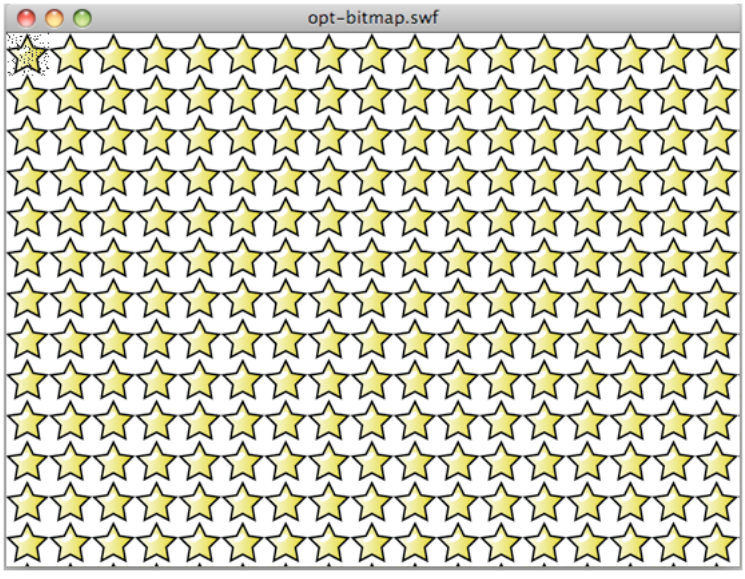
심볼의 인스턴스를 여러 개 만드는 코드의 결과

예를 들어 위 애니메이션은 Flash Player 10에서 약 1008KB의 메모리를 사용하지만 Flash Player 10.1에서는 데이크톱이나 휴대 장치에서 단지 4KB만 사용합니다.

다음 코드에서는 한 BitmapData 인스턴스를 수정합니다.

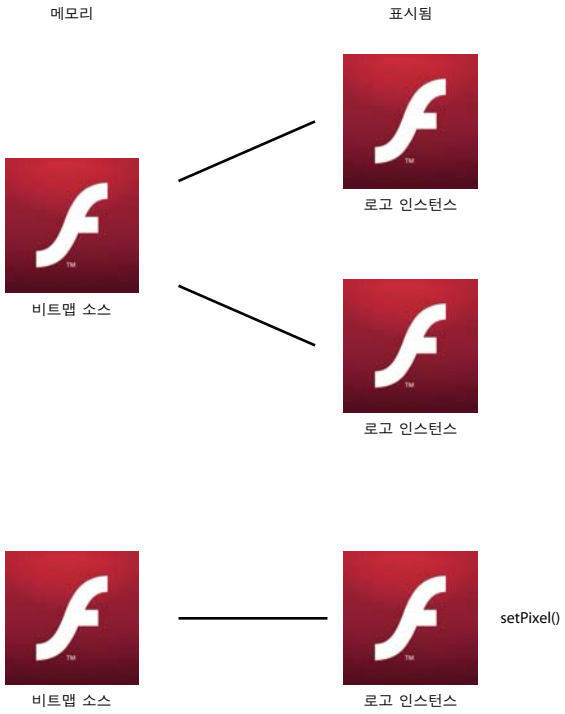
```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}

var ref:Bitmap = getChildAt(0) as Bitmap;

ref.bitmapData.pixelDissolve(ref.bitmapData, ref.bitmapData.rect, new
Point(0,0),Math.random()*200,Math.random()*200, 0x990000);
```

다음 이미지는 Star 인스턴스 하나를 수정한 결과를 보여 줍니다.



인스턴스 하나를 수정한 결과

내부적으로 런타임에서 비트맵을 메모리에 자동으로 할당하고 만들어 픽셀 수정을 처리합니다. BitmapData 클래스의 메서드가 호출되면 픽셀이 수정되어 새 인스턴스가 메모리에 만들어지며 다른 모든 인스턴스는 업데이트되지 않습니다. 다음 그림에서는 이 개념을 보여 줍니다.



메모리에서 비트맵 하나를 수정한 결과

별 하나가 수정되면 메모리에 새로운 사본이 만들어집니다. 결과적으로 Flash Player 10.1 및 AIR 2.5에서 애니메이션은 메모리 8KB를 사용합니다.

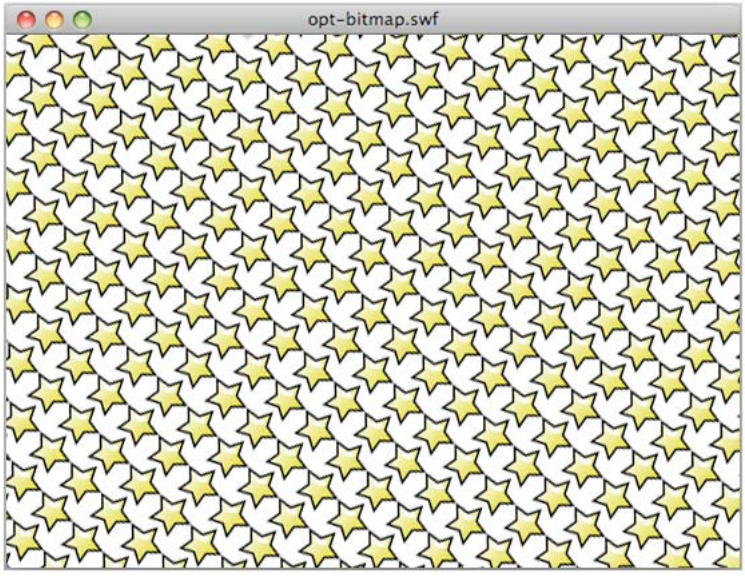
이전 예제에서 변형에 각 비트맵을 개별적으로 사용할 수 있습니다. 타일링 효과만 만들려면 beginBitmapFill() 메서드가 가장 적절한 메서드입니다.

```
var container:Sprite = new Sprite();

var source:BitmapData = new Star(0,0);

// Fill the surface with the source BitmapData
container.graphics.beginBitmapFill(source);
container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);

addChild(container);
```

이 방법은 단일 BitmapData 인스턴스만 만들었을 때와 같은 결과를 생성합니다. 별을 지속적으로 회전하려면 각 Star 인스턴스에 액세스하는 대신에 각 프레임에서 회전되는 Matrix 객체를 사용하십시오. 다음과 같이 Matrix 객체를 beginBitmapFill() 메서드에 전달합니다.

```
var container:Sprite = new Sprite();

container.addEventListener(Event.ENTER_FRAME, rotate);

var source:BitmapData = new Star(0,0);
var matrix:Matrix = new Matrix();

addChild(container);

var angle:Number = .01;

function rotate(e:Event):void
{
    // Rotate the stars
    matrix.rotate(angle);

    // Clear the content
    container.graphics.clear();

    // Fill the surface with the source BitmapData
    container.graphics.beginBitmapFill(source,matrix,true,true);
    container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);
}
```

이 기법을 사용하면 효과를 만드는 데 ActionScript 루프가 필요하지 않습니다. 런타임은 모든 작업을 내부적으로 수행합니다. 다음 이미지는 별을 변환한 결과를 보여 줍니다.



별을 회전한 결과

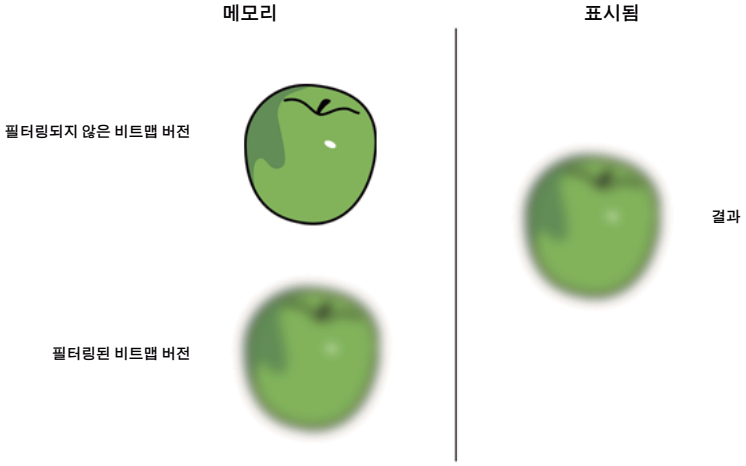
이 방법을 사용하면 원래 소스 BitmapData 객체를 업데이트할 경우 스테이지에서 사용된 다른 모든 곳에서 자동으로 업데이트되므로 이 기술은 매우 유용하게 사용할 수 있습니다. 그러나 이 방식을 사용하면 이전 예제에서처럼 각 별의 크기를 개별적으로 조절할 수 없습니다.

참고: 같은 이미지에 대한 인스턴스를 여러 개 사용할 경우 클래스가 메모리의 원래 비트맵과 연관되는지 여부에 따라 그리기가 달라집니다. 클래스가 비트맵과 연관되지 않으면 이미지는 비트맵 채우기가 포함된 Shape 객체로 그려집니다.

필터 및 동적 비트맵 언로드

Pixel Bender를 통해 처리하는 필터를 비롯하여 필터 사용을 피합니다.

Pixel Bender를 통해 휴대 장치에서 처리되는 필터를 비롯하여 필터와 같은 효과의 사용을 최소화하십시오. 필터 효과가 표시 객체에 적용되면 런타임은 메모리에 두 개의 비트맵을 만듭니다. 이러한 비트맵은 각각 표시 객체의 크기를 가집니다. 첫 번째 비트맵은 표시 객체의 래스터화된 버전으로 만들어져 필터가 적용된 두 번째 비트맵을 생성하는 데 다시 사용됩니다.



필터가 적용되었을 때 메모리의 두 비트맵

필터의 속성 중 하나를 수정하면 두 비트맵이 메모리에서 업데이트되어 결과 비트맵이 만들어집니다. 이 프로세스에는 약간의 CPU 처리가 수반되며 두 비트맵에 상당한 양의 메모리가 사용될 수 있습니다.

Flash Player 10.1 및 AIR 2.5는 모든 플랫폼에서 작동하는 새 필터링 비헤이비어를 도입했습니다. 필터가 30초 내에 수정되지 않거나, 숨겨지거나 화면에서 벗어나는 경우 필터가 적용되지 않은 비트맵에 사용된 메모리가 해제됩니다.

이 기능은 모든 플랫폼에서 필터에 사용되는 메모리를 절반으로 줄입니다. 예를 들어 흐림 필터가 적용된 텍스트 객체를 고려해 봅니다. 이 예에서 텍스트는 간단한 장식에 사용되고 수정되지 않습니다. 30초 후 메모리에서 필터가 적용되지 않은 비트맵이 해제됩니다. 텍스트가 30초 동안 숨겨지거나 화면에서 벗어나는 경우에도 똑같은 결과가 발생합니다. 필터 속성 중 하나가 수정되는 경우 메모리에서 필터가 적용되지 않은 비트맵이 다시 만들어집니다. 이 기능을 동적 비트맵 언로드라고 합니다. 이러한 최적화를 사용하는 경우에도 필터 사용 시에는 주의해야 합니다. 필터를 수정할 경우 여전히 많은 양의 CPU 또는 GPU 처리가 필요합니다.

가장 좋은 방법은 가능한 경우 제작 도구(예: Adobe® Photoshop®)를 통해 만든 비트맵을 사용하여 필터를 에뮬레이션하는 것입니다. ActionScript에서 런타임에 만든 동적 비트맵은 사용하지 마십시오. 외부에서 제작된 비트맵을 사용하면 런타임에서 CPU 또는 GPU 로드를 줄이는 데 도움이 되며, 시간이 지나도 필터 속성이 변경되지 않는 경우에 특히 유용합니다. 가능한 경우 제작 도구에서 비트맵에 대해 필요한 모든 효과를 만드십시오. 그러면 아무런 처리도 수행하지 않고 런타임에서 비트맵을 표시할 수 있으므로 속도가 훨씬 더 빨라질 수 있습니다.

직접 밉매핑

필요한 경우 밉매핑을 사용하여 큰 이미지의 크기를 조절합니다.

모든 플랫폼에서 사용할 수 있는 Flash Player 10.1 및 AIR 2.5의 또 다른 새 기능은 밉매핑과 관련됩니다. Flash Player 9 및 AIR 1.0에는 크기가 작게 조절된 비트맵의 품질과 성능을 향상시키는 밉매핑 기능이 새로 추가되었습니다.

참고: 밉매핑 기능은 동적으로 로드된 이미지 또는 포함된 비트맵에만 적용됩니다. 필터가 적용되거나 캐시된 표시 객체에는 적용되지 않습니다. 밉매핑은 비트맵의 폭과 높이가 짝수인 경우에만 처리할 수 있습니다. 홀수인 폭 또는 높이가 발견되면 밉매핑이 중지됩니다. 예를 들어 250 x 250 이미지를 125 x 125로 밉매핑할 수 있지만 그 이하로 밉매핑할 수는 없습니다. 이 경우 크기 중 적어도 하나가 홀수입니다. 256 x 256, 512 x 512, 1024 x 1024와 같이 비트맵 크기가 2의 거듭제곱일 때 최상의 결과를 얻을 수 있습니다.

예를 들어 1024 x 1024 이미지를 로드하고 개발자가 이미지의 크기를 조절하여 갤러리에 축소판을 만들려고 한다고 가정해 보겠습니다. 중간 크기로 다운샘플링된 버전의 비트맵을 텍스처로 사용하여 크기를 조절하는 경우 밉매핑 기능이 이미지를 올바르게 렌더링합니다. 이전 버전의 런타임에서는 중간 크기로 조절된 버전의 비트맵을 메모리에 만들었습니다. 1024 x 1024 이미지를 로드하고 64 x 64에서 표시하는 경우 이전 버전의 런타임에서는 절반 크기의 비트맵을 차례로 모두 만들었습니다. 예를 들어 이 예에서는 512 x 512, 256 x 256, 128 x 128, 및 64 x 64 비트맵이 만들어집니다.

이제 Flash Player 10.1 및 AIR 2.5에서는 원래 소스에서 필요한 대상 크기로 바로 밉매핑을 만들 수 있습니다. 이전 예제에서 4MB(1024 x 1024)의 원래 비트맵과 16KB(64 x 64)의 밉매핑된 비트맵만 만들어집니다.

밉매핑 논리는 동적 비트맵 언로드 기능과도 작동합니다. 64 x 64 비트맵만 사용하는 경우 4MB의 원래 비트맵이 메모리에서 해제됩니다. 밉매핑을 다시 만들어야 하는 경우에는 원래 비트맵이 다시 로드됩니다. 또한 밉매핑된 다양한 크기의 다른 비트맵이 필요할 경우 비트맵의 밉매핑 체인을 사용하여 비트맵을 만들 수 있습니다. 예를 들어 1:8 비트맵을 만들어야 하는 경우 1:4/1:2/1:1 비트맵을 검사하여 메모리에 처음으로 로드되는 비트맵을 확인합니다. 다른 버전이 발견되지 않으면 1:1의 원래 비트맵이 리소스에서 로드되어 사용됩니다.

JPEG 압축 해제 프로그램은 자체의 고유한 형식 내에서 밉매핑을 수행할 수 있습니다. 이러한 직접 밉매핑을 사용하면 압축되지 않은 전체 이미지를 로드하지 않고 큰 비트맵을 밉맵 형식으로 바로 압축 해제할 수 있습니다. 밉맵 생성은 상당히 빠르며 큰 비트맵에 사용되는 메모리가 할당되지 않고 해제됩니다. JPEG 이미지 품질은 일반 밉매핑 기술과 비슷한 수준입니다.

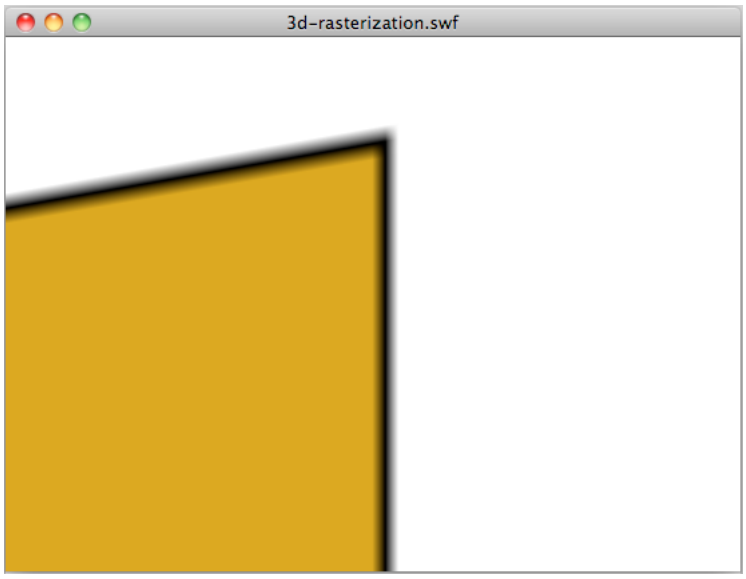
참고: 밉매핑은 가능한 한 사용하지 마십시오. 밉매핑은 다운스케일 비트맵의 품질을 향상시키지만 대역폭, 메모리 및 속도에 영향을 줍니다. 일부 경우, 외부 도구에서 비트맵의 크기가 미리 조절된 버전을 사용하고 해당 버전을 응용 프로그램으로 가져오는 것이 더 나을 수 있습니다. 비트맵의 크기를 줄이려는 경우 큰 비트맵으로 시작하지 마십시오.

3D 효과 사용

3D 효과를 수동으로 만들어 봅니다.

Flash Player 10 및 AIR 1.5에는 표시 객체에 원근 변형을 적용할 수 있는 3D 엔진이 새로 추가되었습니다. 이러한 원근 변형은 rotationX 및 rotationY 속성을 사용하거나 Graphics 클래스의 drawTriangles() 메서드를 사용하여 적용할 수 있습니다. z 속성을 사용하여 깊이를 적용할 수도 있습니다. 하지만 원근 변형된 각 표시 객체는 비트맵으로 래스터화되므로 메모리를 더 많이 요구한다는 점에 유의하십시오.

다음 그림에서는 원근 변형을 사용할 때 래스터화를 통해 생성된 앤티앨리어싱을 보여 줍니다.



원근 변형의 결과로 생성된 앤티앨리어싱

앤티앨리어싱은 벡터 내용을 비트맵으로 동적으로 래스터화한 결과입니다. 이러한 앤티앨리어싱은 데스크톱 버전의 AIR 및 Flash Player와 휴대 장치용 AIR 2.0.1 및 AIR 2.5에서 3D 효과를 사용하는 경우에 발생합니다. 하지만 휴대 장치용 Flash Player에는 앤티앨리어싱이 적용되지 않습니다.

기본 API를 사용하지 않고 3D 효과를 수동으로 만들면 메모리 사용을 줄일 수 있습니다. 하지만 Flash Player 10 및 AIR 1.5에 도입된 새로운 3D 기능을 사용하면 손쉽게 텍스처 매핑을 수행할 수 있습니다. 텍스처 매핑을 기본적으로 처리하는 drawTriangles() 등의 메서드가 지원되기 때문입니다.

개발자는 만들고자 하는 3D 효과가 기본 API를 통해 처리할 때 더 나은 성능을 발휘하는지, 아니면 수동으로 처리할 때 더 나은 성능을 발휘하는지를 결정해야 합니다. ActionScript 실행 및 렌더링 성능과 메모리 사용을 함께 고려합니다.

renderMode 응용 프로그램 속성을 GPU로 설정한 AIR 2.0.1 및 AIR 2.5 모바일 응용 프로그램에서 GPU는 3D 변형을 수행하지 않습니다. 하지만 renderMode가 CPU이면 CPU가 3D 변형을 수행합니다(GPU 아님). Flash Player 10.1 응용 프로그램에서 CPU는 3D 변형을 수행합니다.

CPU가 3D 변형을 수행하는 경우 표시 객체에 3D 변형을 적용하면 메모리에 두 개의 비트맵이 필요하다는 점을 고려하십시오. 비트맵 하나는 소스 비트맵용이고 두 번째 비트맵은 원근감 변형 버전용입니다. 이와 같이 3D 변형은 필터와 비슷한 방식으로 작동합니다. 그러므로 CPU가 3D 변형을 수행하는 경우에는 3D 속성을 조금만 사용하십시오.

텍스트 객체와 메모리

읽기 전용 텍스트에 Adobe® Flash® Text Engine을 사용하고 입력 텍스트에 TextField 객체를 사용하십시오.

Flash Player 10 및 AIR 1.5는 시스템 메모리를 절약할 수 있는 강력한 차세대 엔진인 Adobe Flash Text Engine(FTE)의 사용을 도입했습니다. 하지만 FTE는 하위 수준 API로서 추가 ActionScript 3.0 레이어가 필요하며 이는 flash.text.engine 패키지에 제공됩니다.

읽기 전용 텍스트의 경우 메모리 사용이 적고 렌더링 품질이 나은 Flash Text Engine을 사용하는 것이 가장 적합합니다. 입력 텍스트의 경우 입력 처리 및 줄 바꿈과 같은 일반적인 비헤이비어를 만드는 데 ActionScript 코드가 더 적게 필요하기 때문에 TextField 객체를 사용하는 것이 좋습니다.

기타 도움말 항목

59페이지의 “텍스트 객체 렌더링”

이벤트 모델과 콜백


이벤트 모델 대신에 단순 콜백을 사용하는 것이 좋습니다.

ActionScript 3.0 이벤트 모델은 객체 전달의 개념에 기반합니다. 이벤트 모델은 객체 지향적이며 코드 재사용을 위해 최적화되어 있습니다. dispatchEvent() 메서드는 리스너 목록을 반복하며 등록된 각 객체에 대한 이벤트 핸들러 메서드를 호출합니다. 그러나 이벤트 모델의 단점 중 하나는 응용 프로그램의 수명 주기 동안 많은 객체를 만들 가능성이 높다는 것입니다.

타임라인에서 이벤트를 전달하여 애니메이션 시퀀스의 끝을 나타내야 한다고 가정해 보겠습니다. 알림을 수행하기 위해 다음 코드와 같이 타임라인의 특정 프레임에서 이벤트를 전달할 수 있습니다.

```
dispatchEvent( new Event ( Event.COMPLETE ) );
```

Document 클래스는 다음과 같은 코드 행을 사용하여 이 이벤트를 수신할 수 있습니다.

```
addEventListener( Event.COMPLETE, onAnimationComplete );
```

이 방식이 올바르긴 하지만, 기본 이벤트 모델을 사용하면 기존 콜백 함수를 사용하는 경우보다 속도가 느려지고 메모리 사용량이 많아질 수 있습니다. Event 객체를 만들어 메모리에 할당해야 하므로 성능이 저하됩니다. 예를 들어 Event.ENTER_FRAME 이벤트를 수신할 때 이벤트 핸들러에 대한 각 프레임에 새 이벤트 객체가 만들어집니다. 표시 객체의 경우 표시 목록이 복잡하면 리소스 사용량이 많은 캡처 및 버블링 단계로 인해 특히 성능이 저하될 수 있습니다.

3장: CPU 사용 최소화

최적화에서 중점적으로 설명할 또 다른 중요 영역은 CPU 사용입니다. CPU 처리를 최적화하면 성능이 향상되고, 그 결과 휴대 장치의 배터리 수명도 늘어납니다.

Flash Player 10.1의 CPU 사용 관련 향상 기능

Flash Player 10.1에는 CPU 처리를 줄이는 데 유용한 두 가지 새 기능이 추가되었습니다. 이 기능은 SWF 내용이 화면에서 사라졌을 때 일시 정지하고 다시 시작하는 작업과 더불어, 한 페이지에 표시할 수 있는 Flash Player 인스턴스를 제한하는 것과 관련이 있습니다.

일시 정지, 제한 및 다시 시작

참고: 일시 정지, 제한 및 다시 시작 기능은 Adobe® AIR® 응용 프로그램에 적용되지 않습니다.

CPU 및 배터리 사용을 최적화하기 위해 Flash Player 10.1에는 비활성 인스턴스와 관련된 새로운 기능이 추가되었습니다. 이 기능을 사용하면 내용이 화면에서 사라졌다가 표시될 때 SWF 파일을 일시 정지했다가 다시 시작하여 CPU 사용을 제한할 수 있습니다. 이 기능을 사용하면 Flash Player가 내용 재생을 다시 시작할 때 다시 만들 수 있는 객체를 제거하여 메모리를 가능한 한 많이 확보할 수 있습니다. 내용은 전체 내용이 화면을 벗어날 때 화면을 벗어나는 것으로 간주됩니다.

SWF 내용이 화면을 벗어나는 이유는 다음 두 가지가 있습니다.

- 사용자가 페이지를 스크롤하여 SWF 내용이 화면을 벗어납니다.

  이 경우 재생 중인 오디오 또는 비디오가 있으면 내용은 계속 재생되지만 렌더링은 중지됩니다. 재생 중인 오디오 또는 비디오가 없으면 재생이나 ActionScript 실행이 일시 정지되지 않도록 hasPriority HTML 매개 변수를 true로 설정합니다. 그러나 SWF 내용이 화면을 벗어나거나 숨겨지면 hasPriority HTML 매개 변수의 값에 상관없이 해당 내용 렌더링이 일시 정지됩니다.

- 브라우저에 탭이 열려 있어 SWF 내용이 백그라운드로 이동합니다.

  이 경우 hasPriority HTML 태그의 값에 상관없이 SWF 내용의 속도가 2fps에서 8fps 사이로 느려지거나 제한됩니다. SWF 내용이 다시 표시되지 않는 경우 오디오 및 비디오 재생이 중지되고 내용 렌더링이 처리되지 않습니다.

Windows 및 Mac 데스크톱 브라우저에서 실행되는 Flash Player 11.2 이상 버전의 경우 응용 프로그램에서 ThrottleEvent를 사용할 수 있습니다. Flash Player는 재생을 일시 정지, 제한 또는 다시 시작할 때 ThrottleEvent를 전달합니다.

ThrottleEvent는 브로드캐스트 이벤트입니다. 즉, 이 이벤트에 등록된 리스너가 있는 모든 EventDispatcher 객체를 통해 전달됩니다. 브로드캐스트 이벤트에 대한 자세한 내용은 DisplayObject 클래스를 참조하십시오.

인스턴스 관리

참고: 인스턴스 관리 기능은 Adobe® AIR® 응용 프로그램에 적용되지 않습니다.


hasPriority HTML 매개 변수를 사용하여 SWF 파일의 로드를 지연합니다.

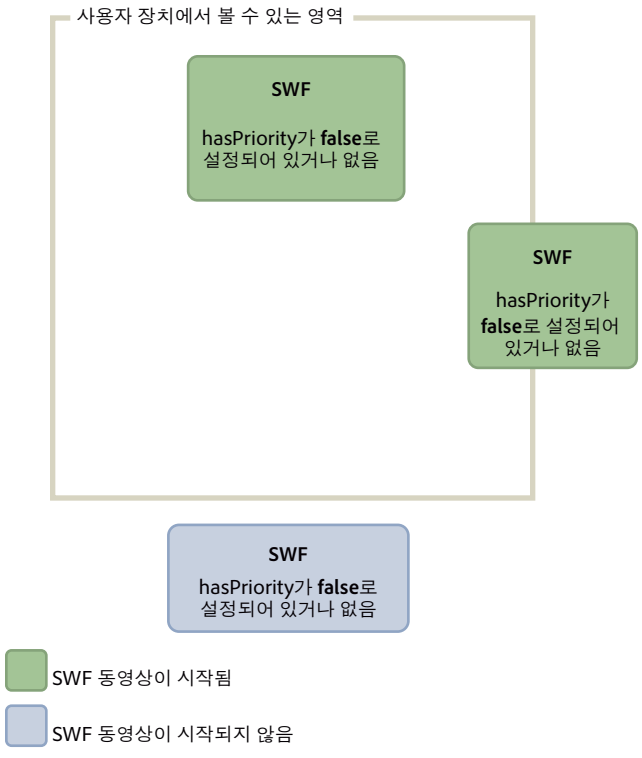
Flash Player 10.1에는 hasPriority라는 새로운 HTML 매개 변수가 도입되었습니다.

```
<param name="hasPriority" value="true" />
```

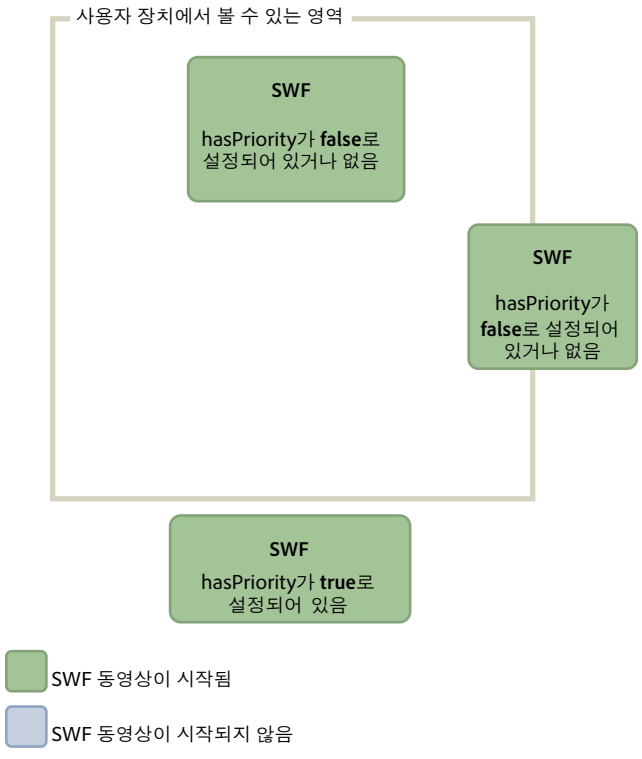
이 기능은 페이지에서 시작되는 Flash Player 인스턴스의 수를 제한합니다. 인스턴스 수를 제한하면 CPU 및 배터리 리소스를 절약하는 데 도움이 됩니다. 이 매개 변수는 특정 우선 순위를 SWF 내용에 할당하여 페이지에서 내용의 우선 순위를 지정하기 위해 만들어졌습니다. 사용자가 웹 사이트를 탐색하고 인덱스 페이지에서 세 가지 다른 SWF 파일을 호스트하는 간단한 예를 고려해 봅니다. 세 파일 중 하나는 완전히 표시되고 또 다른 하나는 화면에 부분적으로 표시되며 마지막 하나는 화면을 벗어나기 때문에 스크롤이 필요합니다. 처음 두 애니메이션은 정상적으로 시작되지만 마지막 하나는 화면에 나타날 때까지 시작되지 않습니다. 이 시나리오는 hasPriority 매개 변수가 없거나 false로 설정되어 있는 경우의 기본 비헤이비어입니다. SWF 파일이 화면에서 벗어난 경우에도 시작되도록 하려면 hasPriority 매개 변수를 true로 설정합니다. 그러나 hasPriority 매개 변수의 값에 상관없이 사용자에게 표시되지 않는 SWF 파일의 렌더링은 항상 일시 정지됩니다.

참고: 사용 가능한 CPU 리소스가 부족해지면 hasPriority 매개 변수가 true로 설정된 경우에도 Flash Player 인스턴스가 더 이상 자동으로 시작되지 않습니다. 페이지가 로드된 후 JavaScript를 통해 만들어진 새 인스턴스는 hasPriority 플래그를 무시합니다. 웹 마스터가 hasPriority 플래그를 포함시키지 않은 경우 1x1 픽셀 또는 0x0 픽셀 내용이 시작되어 도우미 SWF 파일이 지연되지 않도록 합니다. 그러나 클릭할 경우에는 SWF 파일이 시작될 수 있습니다. 이 비헤이비어를 "클릭하여 재생(click to play)"이라고 합니다.

다음 다이어그램에서는 hasPriority 매개 변수를 다른 값으로 설정할 경우의 효과를 보여 줍니다.



hasPriority 매개 변수의 값에 따른 다양한 효과



hasPriority 매개 변수의 값에 따른 다양한 효과

절전 모드

Flash Player 10.1 및 AIR 2.5에는 CPU 처리를 줄이고, 결과적으로 배터리 수명을 늘리는 데 도움이 되는 휴대 장치용 새 기능이 도입되었습니다. 이 기능은 많은 휴대 장치에서 볼 수 있는 백라이트 기능과 관련됩니다. 예를 들어 사용자가 모바일 응용 프로그램 실행을 중단하고 장치의 사용을 중지하는 경우 런타임에서 백라이트가 절전 모드로 전환되었음을 감지하여 프레임 속도를 4프레임/초(fps)로 줄이고 렌더링을 일시 정지합니다. AIR 응용 프로그램의 경우 응용 프로그램이 백그라운드로 이동할 때 절전 모드가 시작됩니다.

ActionScript 코드는 절전 모드에서 계속 실행됩니다. 이것은 Stage.frameRate 속성을 4fps로 설정하는 것과 비슷하지만, 렌더링 단계를 건너뛰므로 사용자는 플레이어가 4fps에서 실행 중임을 인식할 수 없습니다. 프레임 속도를 0이 아닌 4fps로 선택한 것은 모든 연결(NetStream, Socket 및 NetConnection)을 열려 있는 상태로 유지할 수 있기 때문입니다. 프레임 속도를 0으로 설정하면 열려 있는 연결이 끊깁니다. 새로 고침 속도를 250ms(4fps)로 선택한 이유는 대부분의 장치 제조업체에서 이 프레임 속도를 새로 고침 속도로 사용하기 때문입니다. 이 값을 사용하면 런타임의 프레임 속도가 장치 자체의 속도와 거의 비슷하게 유지됩니다.

참고: 런타임에서 절전 모드인 경우 Stage.frameRate 속성은 4fps가 아닌 원본 SWF 파일의 프레임 속도를 반환합니다.

백라이트가 정상 모드로 다시 전환되면 렌더링이 다시 시작되고 프레임 속도도 원래 값으로 돌아갑니다. 사용자가 음악을 재생하고 있는 미디어 플레이어 응용 프로그램을 고려해 봅니다. 화면이 절전 모드로 전환되면 런타임은 재생 중인 내용의 유형에 따라 응답합니다. 다음 목록에는 여러 가지 경우와 해당 런타임 비헤이비어가 나열되어 있습니다.

- 백라이트가 절전 모드로 전환되고 A/V 내용이 재생되고 있지 않은 경우 렌더링이 일시 정지되고 프레임 속도가 4fps로 설정됩니다.
- 백라이트가 절전 모드로 전환되고 A/V 내용이 재생되고 있는 경우 런타임에서 백라이트를 항상 켜진 상태로 두고 사용자 환경을 계속 유지합니다.

- 백라이트가 절전 모드에서 정상 모드로 전환되는 경우 런타임에서 프레임 속도를 원래 SWF 파일 프레임 속도 설정으로 지정하고 렌더링을 다시 시작합니다.
- A/V 내용이 재생되는 동안 Flash Player가 일시 정지되는 경우 A/V가 더 이상 재생되지 않기 때문에 Flash Player는 백라이트 상태를 기본 시스템 비헤이비어로 다시 설정합니다.
- A/V 내용을 재생하는 동안 휴대 장치에서 전화를 수신하는 경우 렌더링이 일시 정지되고 프레임 속도가 4fps로 설정됩니다.
- 휴대 장치에서 백라이트 절전 모드가 해제된 경우 런타임에서 정상적으로 동작합니다.

백라이트가 절전 모드로 전환되는 경우 렌더링이 일시 정지되고 프레임 속도가 느려집니다. 이 기능은 CPU 처리를 줄이지만 게임 응용 프로그램에서처럼 실제 일시 정지를 만드는 데 사용할 수 없습니다.

참고: 런타임에서 절전 모드가 시작되거나 끝날 때 ActionScript 이벤트가 전달됩니다.

객체 표시 제거 및 표시 제거 취소


REMOVED_FROM_STAGE 및 ADDED_TO_STAGE 이벤트를 사용하여 객체를 올바르게 표시 제거하거나 표시 제거 취소합니다.

코드를 최적화하려면 항상 객체를 표시 제거 및 표시 제거 취소해야 합니다. 표시 제거 및 표시 제거 취소는 모든 객체에 대해 중요하지만 특히 표시 객체의 경우 중요합니다. 표시 객체는 표시 목록에 더 이상 없고 가비지 수집되기 위해 대기 중인 경우에도 CPU 소모가 많은 코드를 계속 사용할 수 있습니다. 예를 들어 Event.ENTER_FRAME을 계속 사용할 수 있습니다. 따라서 Event.REMOVED_FROM_STAGE 및 Event.ADDED_TO_STAGE 이벤트를 사용하여 객체를 올바르게 표시 제거 및 표시 제거 취소하는 것이 중요합니다. 다음 예제에서는 키보드를 사용하여 상호 작용하는 스테이지에서 재생되는 동영상 클립을 보여 줍니다.

```
// Listen to keyboard events
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyIsDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyIsUp);

// Create object to store key states
var keys:Dictionary = new Dictionary(true);

function keyIsDown(e:KeyboardEvent):void
{
    // Remember that the key was pressed
    keys[e.keyCode] = true;

    if (e.keyCode==Keyboard.LEFT || e.keyCode==Keyboard.RIGHT)
    {
        runningBoy.play();
    }
}

function keyIsUp(e:KeyboardEvent):void
{
    // Remember that the key was released
    keys[e.keyCode] = false;

    for each (var value:Boolean in keys)
        if ( value ) return;
    runningBoy.stop();
```

```
}

runningBoy.addEventListener(Event.ENTER_FRAME, handleMovement);
runningBoy.stop();

var currentState:Number = runningBoy.scaleX;
var speed:Number = 15;

function handleMovement(e:Event):void
{
    if (keys[Keyboard.RIGHT])
    {
        e.currentTarget.x += speed;
        e.currentTarget.scaleX = currentState;
    } else if (keys[Keyboard.LEFT])
    {
        e.currentTarget.x -= speed;
        e.currentTarget.scaleX = -currentState;
    }
}
```

키보드와 상호 작용하는 동영상 클립

[Remove] 버튼을 클릭할 경우 동영상 클립이 표시 목록에서 제거됩니다.

```
// Show or remove running boy
showBtn.addEventListener (MouseEvent.CLICK,showIt);
removeBtn.addEventListener (MouseEvent.CLICK,removeIt);

function showIt (e:MouseEvent):void
{
    addChild (runningBoy);
}

function removeIt(e:MouseEvent):void
{
    if (contains(runningBoy)) removeChild(runningBoy);
}
```

동영상 클립은 표시 목록에서 제거된 경우에도 Event.ENTER_FRAME 이벤트를 계속 전달합니다. 동영상 클립은 여전히 실행되지만 렌더링되지는 않습니다. 이 상황을 올바르게 처리하려면 올바른 이벤트를 수신하고 이벤트 리스너를 제거하여 CPU를 많이 사용하는 코드가 실행되지 않도록 해야 합니다.

```
// Listen to Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE
runningBoy.addEventListener(Event.ADDED_TO_STAGE,activate);
runningBoy.addEventListener(Event.REMOVED_FROM_STAGE,deactivate);

function activate(e:Event):void
{
    // Restart everything
    e.currentTarget.addEventListener(Event.ENTER_FRAME,handleMovement);
}

function deactivate(e:Event):void
{
    // Freeze the running boy - consumes fewer CPU resources when not shown
    e.currentTarget.removeEventListener(Event.ENTER_FRAME,handleMovement);
    e.currentTarget.stop();
}
```

[표시] 버튼을 누르면 동영상 클립이 다시 시작되고, Event.ENTER_FRAME 이벤트를 다시 수신하며, 키보드로 동영상 클립을 올바르게 제어할 수 있습니다.

참고: 표시 객체를 표시 목록에서 제거하는 경우 객체를 제거한 후 해당 참조를 null로 설정해도 객체가 표시 제거되지 않을 수 있습니다. 가비지 수집기가 실행되지 않으면 해당 객체는 더 이상 표시되지 않는 경우에도 계속해서 메모리와 CPU 처리를 소모합니다. 객체가 가능한 한 최소 CPU 처리를 사용하도록 하려면 객체를 표시 목록에서 제거할 때 완전히 표시 제거해야 합니다.

Flash Player 10 및 AIR 1.5부터는 다음 비헤이비어도 발생합니다. 재생 헤드가 빈 프레임을 발견할 경우 사용자가 표시 제거 비헤이비어를 구현하지 않았더라도 해당 표시 객체가 자동으로 표시 제거됩니다.

표시 제거 개념은 Loader 클래스를 사용하여 원격 내용을 로드하는 경우에도 중요합니다. Flash Player 9 및 AIR 1.0에서 Loader 클래스를 사용하는 경우 LoaderInfo 객체에서 전달한 Event.UNLOAD 이벤트를 수신하여 내용을 수동으로 표시 제거해야 했습니다. 모든 객체를 수동으로 표시 제거해야 했으며, 이는 매우 번거로운 작업이었습니다. Flash Player 10 및 AIR 1.5에서는 Loader 클래스에 unloadAndStop()이라는 중요한 새 메서드가 추가되었습니다. 이 메서드를 사용하면 SWF 파일을 언로드하고, 로드된 SWF 파일의 모든 객체를 자동으로 표시 제거하며, 가비지 수집기를 강제로 실행할 수 있습니다.

다음 코드에서는 SWF 파일이 로드된 다음 unload() 메서드를 사용하여 언로드되는데, 이 작업에는 더 많은 처리와 수동 표시 제거 작업이 필요합니다.

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with no automatic object deactivation
    // All deactivation must be processed manually
    loader.unload();
}
```

가장 좋은 방법은 기본적으로 표시 제거를 처리하고 가비지 수집 프로세스를 강제로 실행하는 unloadAndStop() 메서드를 사용하는 것입니다.

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with automatic object deactivation
    // All deactivation is handled automatically
    loader.unloadAndStop();
}
```

unloadAndStop() 메서드를 호출하면 다음 작업이 수행됩니다.

- 사운드가 중지됩니다.
- SWF 파일의 기본 타임라인에 등록된 리스너가 제거됩니다.
- 타이머 객체가 중지됩니다.
- 하드웨어 주변 장치(예: 카메라 및 마이크)가 해제됩니다.
- 모든 동영상 클립이 중지됩니다.
- Event.ENTER_FRAME, Event.FRAME_CONSTRUCTED, Event.EXIT_FRAME, Event.ACTIVATE 및 Event.DEACTIVATE 전달이 중지됩니다.

이벤트 활성화 및 비활성화

Event.ACTIVATE 및 Event.DEACTIVATE 이벤트를 사용하면 백그라운드의 활동이 중단된 것을 감지하여 응용 프로그램을 그에 맞게 최적화할 수 있습니다.

두 이벤트(Event.ACTIVATE 및 Event.DEACTIVATE)는 응용 프로그램을 미세하게 조정할 수 있도록 지원하므로 가급적 CPU 주기를 거의 사용하지 않습니다. 이들 이벤트를 사용하면 런타임이 포커스를 얻거나 잃을 때를 감지합니다. 그러므로 컨텍스트의 변화에 맞게 코드를 최적화할 수 있습니다. 다음은 이 두 이벤트를 수신하고 응용 프로그램이 포커스를 잃으면 동적으로 프레임 속도를 0으로 변경하는 코드입니다. 예를 들어 사용자가 다른 탭으로 전환하거나 응용 프로그램을 백그라운드에 배치하면 애니메이션은 포커스를 잃을 수 있습니다.

```
var originalFrameRate:uint = stage.frameRate;
var standbyFrameRate:uint = 0;

stage.addEventListener ( Event.ACTIVATE, onActivate );
stage.addEventListener ( Event.DEACTIVATE, onDeactivate );

function onActivate ( e:Event ):void
{
    // restore original frame rate
    stage.frameRate = originalFrameRate;
}

function onDeactivate ( e:Event ):void
{
    // set frame rate to 0
    stage.frameRate = standbyFrameRate;
}
```

응용 프로그램이 포커스를 다시 획득하면 프레임 속도는 원래 값으로 재설정됩니다. 프레임 속도를 동적으로 변경하는 대신, 객체를 표시 제거하거나 표시 제거 취소 등의 다른 최적화 방법을 사용할 수도 있습니다.

이벤트를 활성화하거나 비활성화하면 휴대 장치 및 넷북에 종종 있는 "일시 정지 및 다시 시작" 기능과 유사한 메커니즘을 구현할 수 있습니다.

기타 도움말 항목

47페이지의 “응용 프로그램 프레임 속도”

25페이지의 “객체 표시 제거 및 표시 제거 취소”

마우스 상호 작용

가능한 경우 마우스 상호 작용을 비활성화해 봅니다.

대화형 객체(예: MovieClip 또는 Sprite 객체)를 사용하는 경우 런타임에서는 기본 코드를 실행하여 마우스 상호 작용을 감지하고 처리합니다. 많은 대화형 객체가 화면에 표시되는 경우, 특히 서로 겹치는 경우 마우스 상호 작용을 감지하는 데 CPU가 많이 사용될 수 있습니다. 이러한 처리를 방지하는 쉬운 방법은 마우스 상호 작용이 필요하지 않은 객체에 대해 마우스 상호 작용을 비활성화하는 것입니다. 다음 코드에서는 mouseEnabled 및 mouseChildren 속성의 사용을 보여 줍니다.

```
// Disable any mouse interaction with this InteractiveObject
myInteractiveObject.mouseEnabled = false;
const MAX_NUM:int = 10;

// Create a container for the InteractiveObjects
var container:Sprite = new Sprite();

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    // Add InteractiveObject to the container
    container.addChild( new Sprite() );
}

// Disable any mouse interaction on all the children
container.mouseChildren = false;
```

가능한 경우 마우스 상호 작용을 비활성화하는 것이 좋습니다. 그러면 응용 프로그램에서 사용하는 CPU 처리가 줄어들고, 그 결과 휴대 전화의 배터리 사용도 줄어듭니다.

타이머와 ENTER_FRAME 이벤트

내용에 애니메이션을 적용하는지 여부에 따라 타이머 또는 ENTER_FRAME 이벤트를 선택합니다.

오랜 시간 동안 실행되는, 애니메이션이 적용되지 않은 내용의 경우 Event.ENTER_FRAME 이벤트보다는 타이머를 사용하는 것이 좋습니다.

ActionScript 3.0에는 특정 간격으로 한 함수를 호출할 수 있는 두 가지 방법이 있습니다. 첫 번째 방식은 표시 객체(DisplayObject)가 전달하는 Event.ENTER_FRAME 이벤트를 사용하는 것입니다. 두 번째 방식은 타이머를 사용하는 것입니다. ActionScript 개발자는 ENTER_FRAME 이벤트 방식을 자주 사용합니다. ENTER_FRAME 이벤트는 모든 프레임에서 전달됩

니다. 따라서 함수가 호출되는 간격은 현재 프레임 속도와 관련됩니다. 프레임 속도에는 Stage.frameRate 속성을 통해 액세스할 수 있습니다. 그러나 일부 경우에는 ENTER_FRAME 이벤트를 사용하는 것보다 타이머를 사용하는 것이 나을 수 있습니다. 예를 들어 애니메이션은 사용하지 않지만 코드가 특정 간격으로 호출되도록 하려는 경우 타이머를 사용하는 것이 낫습니다.

타이머는 ENTER_FRAME 이벤트와 비슷한 방식으로 작동할 수 있지만 프레임 속도에 구애받지 않고 이벤트를 전달할 수 있습니다. 이 비헤이비어를 사용하면 상당한 최적화를 실현할 수 있습니다. 비디오 플레이어 응용 프로그램을 예로 들어 생각해 보겠습니다. 이 경우 응용 프로그램 컨트롤만 이동하므로 높은 프레임 속도를 사용할 필요가 없습니다.

참고: 비디오는 타임라인에 포함되지 않으므로 프레임 속도가 비디오에 영향을 주지 않습니다. 대신, 비디오는 점진적 다운로드 또는 스트리밍을 통해 동적으로 로드됩니다.

이 예제에서는 프레임 속도가 10fps의 낮은 값으로 설정되어 있습니다. 타이머는 초당 1개의 업데이트 속도로 컨트롤을 업데이트합니다. TimerEvent 객체에서 사용 가능한 updateAfterEvent() 메서드를 통해 보다 높은 업데이트 속도를 구현할 수 있습니다. 이 메서드에서는 필요한 경우 타이머가 이벤트를 전달할 때마다 화면을 강제로 업데이트합니다. 다음 코드에서는 이러한 아이디어를 보여 줍니다.

```
// Use a low frame rate for the application
stage.frameRate = 10;

// Choose one update per second
var updateInterval:int = 1000;
var myTimer:Timer = new Timer(updateInterval,0);

myTimer.start();
myTimer.addEventListener( TimerEvent.TIMER, updateControls );

function updateControls( e:TimerEvent ):void
{
    // Update controls here
    // Force the controls to be updated on screen
    e.updateAfterEvent();
}
```

updateAfterEvent() 메서드를 호출해도 프레임 속도가 수정되지는 않습니다. 이 메서드는 단지 런타임에서 변경된 화면의 내용을 강제로 업데이트합니다. 타임라인은 여전히 10fps로 실행됩니다. 타이머 및 ENTER_FRAME 이벤트는 성능이 낮은 장치에서 또는 이벤트 핸들러 함수에 처리량이 많은 코드가 포함된 경우에는 완벽히 정확하게 작동하지 않습니다. SWF 파일 프레임 속도와 마찬가지로 타이머의 업데이트 프레임 속도는 일부 상황에서 달라질 수 있습니다.


응용 프로그램에서 Timer 객체 및 등록된 enterFrame 핸들러의 수를 최소화하십시오.

런타임에서는 프레임마다 표시 목록에 있는 각 표시 객체에 enterFrame 이벤트를 전달합니다. 여러 표시 객체를 사용하여 enterFrame 이벤트에 대한 리스너를 등록할 수도 있지만 이렇게 하면 결국 프레임마다 더 많은 코드가 실행됩니다. 대신 각 프레임을 실행하기 위한 코드를 모두 실행하는 중앙화된 단일 enterFrame 핸들러를 사용하는 것이 좋습니다. 이렇게 코드를 중앙화하면 자주 실행되는 모든 코드를 쉽게 관리할 수 있습니다.

또한 Timer 객체를 사용하는 경우 여러 Timer 객체로부터 이벤트를 만들고 전달하기 위해 오버헤드가 발생할 수 있습니다. 서로 다른 간격으로 여러 작업을 트리거해야 하는 경우에는 다음 몇 가지 대안을 고려해 보십시오.

- 작업의 수행 빈도에 따라 Timer 객체 및 그룹 작업 수를 최소한으로 사용하십시오..

  예를 들어, 자주 수행되는 작업에 대해 하나의 Timer를 사용하고 100밀리초마다 트리거되도록 설정합니다. 자주 수행되지 않거나 백그라운드로 수행되는 작업에 대해서도 다른 Timer를 만들고 2000밀리초마다 트리거되도록 설정합니다.

- 단일 Timer 객체를 사용하고 Timer 객체의 delay 속성 간격의 배수로 작업이 트리거되도록 하십시오.

  예를 들어, 100밀리초마다 실행되어야 하는 작업이 있고, 200밀리초마다 실행되어야 하는 작업이 있다고 가정해 보십시오. 이 경우에는 delay 값이 100밀리초로 설정된 단일 Timer 객체를 사용합니다. 그리고 timer 이벤트 핸들러에서 매 다음 회수마다 200밀리초 작업만 실행하도록 하는 조건문을 추가합니다. 다음 예제에서는 이 방법을 보여 줍니다.

```
var timer:Timer = new Timer(100);
timer.addEventListener(TimerEvent.Timer, timerHandler);
timer.start();

var offCycle:Boolean = true;

function timerHandler(event:TimerEvent):void
{
    // Do things that happen every 100 ms

    if (!offCycle)
    {
        // Do things that happen every 200 ms
    }

    offCycle = !offCycle;
}
```


사용 중이 아닐 때는 Timer 객체를 중단하십시오..

Timer 객체의 timer 이벤트 핸들러가 특정 조건에서만 작업을 수행하는 경우 true인 조건이 없으면 Timer 객체의 stop() 메서드를 호출합니다.


enterFrame 이벤트 또는 Timer 핸들러에서 화면을 다시 그려야 하는 표시 객체의 모양 변경 횟수를 최소화하십시오.

각 프레임마다 렌더링 단계에서는 해당 프레임 중에 변경된 스테이지의 부분을 새로 그립니다. 새로 그릴 영역이 크거나, 영역이 작지만 대량 또는 복잡한 표시 객체가 포함되는 경우 런타임에서 렌더링하는 데 더 많은 시간이 필요합니다. 다시 그리는데 필요한 시간을 테스트하려면 Flash Player 또는 AIR 디버깅의 "다시 그리기 영역 표시" 기능을 사용하십시오.

반복되는 액션의 성능 향상에 대한 자세한 내용은 다음 문서를 참조하십시오.

- Writing well-behaved, efficient, AIR applications(Arno Gourdol의 문서 및 샘플 코드 응용 프로그램)

기타 도움말 항목

56페이지의 “비헤이비어 격리”

트위닝 신드롬


CPU 용량을 절약하려면 트위닝 사용을 제한하십시오. 그러면 CPU 처리, 메모리 및 배터리 수명이 절약됩니다.

데스크톱에서 Flash용 내용을 만드는 디자이너 및 개발자는 응용 프로그램에서 많은 모션 트윈을 사용하는 경향이 있습니다. 성능이 낮은 휴대 장치용 내용을 만드는 경우 모션 트윈 사용을 최소화해 보십시오. 모션 트윈 사용을 제한하면 성능이 낮은 장치에서도 내용을 보다 빠르게 실행하는 데 도움이 됩니다.

4장: ActionScript 3.0 성능

Vector 클래스와 Array 클래스


가능한 경우 Array 클래스보다 Vector 클래스를 사용합니다.

Vector 클래스를 사용하면 Array 클래스보다 읽기 및 쓰기 액세스가 빠릅니다.

간단한 벤치마크를 실행해 보면 Array 클래스 대비 Vector 클래스의 이점이 크다는 것을 알 수 있습니다. 다음 코드는 Array 클래스에 대한 벤치마크를 보여 줍니다.

```
var coordinates:Array = new Array();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 107
```

다음 코드는 Vector 클래스에 대한 벤치마크를 보여 줍니다.

```
var coordinates:Vector.<Number> = new Vector.<Number>();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 72
```

이 예제는 벡터에 특정 길이를 할당하고 해당 길이를 고정된 값으로 설정하여 좀 더 최적화할 수 있습니다.

```
// Specify a fixed length and initialize its length
var coordinates:Vector.<Number> = new Vector.<Number>(300000, true);

var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 48
```

벡터의 크기를 사전에 지정하지 않은 경우 벡터의 공간이 부족해지면 크기가 증가합니다. 벡터의 크기가 증가할 때마다 새로운 메모리 블록이 할당됩니다. 벡터의 현재 내용은 새로운 메모리 블록에 복사됩니다. 이러한 추가 할당 및 데이터 복사는 성능에 좋지 않은 영향을 줍니다. 위 코드는 벡터의 초기 크기를 지정하여 성능을 위해 최적화되어 있습니다. 그러나 유지 관리 면에서는 이 코드가 최적화되어 있지 않습니다. 유지 관리 편의성도 향상시키려면 다음과 같이 다시 사용되는 값을 상수로 저장하십시오.

```
// Store the reused value to maintain code easily
const MAX_NUM:int = 300000;

var coordinates:Vector.<Number> = new Vector.<Number>(MAX_NUM, true);
var started:Number = getTimer();

for (var i:int = 0; i< MAX_NUM; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 47
```

Vector 객체 API의 실행 속도가 더 빠를 가능성이 높으므로 가능하면 이 API를 사용해 보십시오.

드로잉 API


드로잉 API를 사용하면 코드 실행 속도가 빨라집니다.

Flash Player 10 및 AIR 1.5에서는 실행 성능이 더 나은 새로운 드로잉 API를 제공합니다. 이 새 API는 렌더링 성능을 향상시키지는 않지만 작성해야 하는 코드의 줄 수를 대폭 줄일 수 있습니다. 코드 줄이 감소되면 ActionScript 실행 성능이 향상될 수 있습니다.

새 드로잉 API에는 다음 메서드가 포함됩니다.

- drawPath()
- drawGraphicsData()
- drawTriangles()

참고: 여기서는 3D와 관련된 drawTriangles()는 자세하게 다루지 않습니다. 하지만 이 메서드는 기본 텍스처 매핑을 처리하기 때문에 ActionScript 성능을 향상시킬 수 있습니다.

다음 코드는 그려지는 각 줄에 대해 해당 메서드를 호출합니다.

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.moveTo ( coords[0], coords[1] );
container.graphics.lineTo ( coords[2], coords[3] );
container.graphics.lineTo ( coords[4], coords[5] );
container.graphics.lineTo ( coords[6], coords[7] );
container.graphics.lineTo ( coords[8], coords[9] );

addChild( container );
```

다음 코드는 실행하는 코드 줄이 더 적기 때문에 이전 예제보다 빠르게 실행됩니다. 경로가 복잡할수록 drawPath() 메서드를 사용하여 성능을 좀더 향상시킬 수 있습니다.

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var commands:Vector.<int> = Vector.<int>([1,2,2,2,2]);
var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.drawPath(commands, coords);

addChild( container );
```

drawGraphicsData() 메서드를 사용해도 비슷한 정도의 성능 개선을 이끌어 낼 수 있습니다.

이벤트 캡처 및 버블링


이벤트 캡처 및 버블링을 사용하여 이벤트 핸들러를 최소화할 수 있습니다.

ActionScript 3.0의 이벤트 모델에는 이벤트 캡처 및 이벤트 버블링의 개념이 도입되었습니다. 이벤트 버블링을 활용하면 ActionScript 코드 실행 시간을 최적화하는 데 도움이 될 수 있습니다. 성능을 향상시키기 위해 여러 객체 대신에 한 객체에 대해 이벤트 핸들러를 등록할 수 있습니다.

예를 들어 사용자가 가능한 한 빠르게 사과를 클릭하여 없애야 하는 게임을 만든다고 가정해 보겠습니다. 이 게임에서는 사과를 클릭할 때마다 화면에서 사과가 없어지고 사용자의 점수가 올라갑니다. 각 사과가 전달하는 MouseEvent.CLICK 이벤트를 수신하려면 다음과 같은 코드를 작성할 수 있습니다.

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    addChild ( currentApple );

    // Listen to the MouseEvent.CLICK event
    currentApple.addEventListener ( MouseEvent.CLICK, onAppleClick );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.currentTarget as InteractiveObject;
    currentAppleClicked.removeEventListener(MouseEvent.CLICK, onAppleClick );
    removeChild ( currentAppleClicked );
}
```

이 코드는 각 Apple 인스턴스에 대해 addEventListener() 메서드를 호출합니다. 또한 사과를 클릭할 때 removeEventListener() 메서드를 사용하여 각 리스너를 제거합니다. 그러나 ActionScript 3.0의 이벤트 모델에서는 일부 이벤트에 대해 캡처 및 버블링 단계를 제공하므로 부모 InteractiveObject에서 이벤트를 수신할 수 있습니다. 따라서 위 코드를 최적화하고 addEventListener() 및 removeEventListener() 메서드에 대한 호출 수를 최소화할 수 있습니다. 다음 코드에서는 캡처 단계를 사용하여 부모 객체에서 이벤트를 수신합니다.

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;
var container:Sprite = new Sprite();

addChild ( container );

// Listen to the MouseEvent.CLICK on the apple's parent
// Passing true as third parameter catches the event during its capture phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, true );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    container.addChild ( currentApple );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

이 코드는 부모 컨테이너에 대해 addEventListener() 메서드를 한 번만 호출하여 단순해지고 훨씬 더 최적화됩니다. 리스너는 더 이상 Apple 인스턴스에 등록되지 않으므로 사과를 클릭할 때 제거할 필요가 없습니다. 이벤트 전파를 중지하여 더 이상 진행되지 않도록 함으로써 onAppleClick() 핸들러를 보다 최적화할 수 있습니다.

```
function onAppleClick ( e:MouseEvent ):void
{
    e.stopPropagation();
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

addEventListener() 메서드에 대한 세 번째 매개 변수로 false를 전달하여 이벤트를 포착하는 데 버블링 단계를 사용할 수도 있습니다.

```
// Listen to the MouseEvent.CLICK on apple's parent
// Passing false as third parameter catches the event during its bubbling phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, false );
```

캡처 단계 매개 변수의 기본값은 false이므로 생략할 수 있습니다.

```
container.addEventListener ( MouseEvent.CLICK, onAppleClick );
```

픽셀을 사용한 작업


setVector() 메서드를 사용하여 픽셀을 페인트합니다.

픽셀을 칠할 때 BitmapData 클래스의 적절한 메서드를 사용하기만 하면 몇 가지 간단한 최적화를 수행할 수 있습니다. 픽셀을 칠하는 빠른 방법은 setVector() 메서드를 사용하는 것입니다.

```
// Image dimensions
var wdth:int = 200;
var hght:int = 200;
var total:int = wdth*hght;

// Pixel colors Vector
var pixels:Vector.<uint> = new Vector.<uint>(total, true);

for ( var i:int = 0; i< total; i++ )
{
    // Store the color of each pixel
    pixels[i] = Math.random()*0xFFFFFF;
}

// Create a non-transparent BitmapData object
var myImage:BitmapData = new BitmapData ( wdth, hght, false );
var imageContainer:Bitmap = new Bitmap ( myImage );

// Paint the pixels
myImage.setVector ( myImage.rect, pixels );
addChild ( imageContainer );
```

setPixel() 또는 setPixel32() 같은 느린 메서드를 사용하는 경우 lock() 및 unlock() 메서드를 사용하여 실행 속도를 높일 수 있습니다. 다음 코드에서는 lock() 및 unlock() 메서드를 사용하여 성능을 향상시킵니다.

```
var buffer:BitmapData = new BitmapData(200,200,true,0xFFFFFFFF);
var bitmapContainer:Bitmap = new Bitmap(buffer);
var positionX:int;
var positionY:int;

// Lock update
buffer.lock();
var starting:Number=getTimer();

for (var i:int = 0; i<2000000; i++)
{
    // Random positions
    positionX = Math.random()*200;
    positionY = Math.random()*200;
    // 40% transparent pixels
    buffer.setPixel32( positionX, positionY, 0x66990000 );
}

// Unlock update
buffer.unlock();
addChild( bitmapContainer );

trace( getTimer () - starting );
// output : 670
```

BitmapData 클래스의 lock() 메서드는 이미지를 잠가 BitmapData 객체가 변경될 때 해당 이미지를 참조하는 객체가 업데이트되지 않도록 합니다. 예를 들어 Bitmap 객체가 BitmapData 객체를 참조하는 경우 BitmapData 객체를 잠그고 변경한 다음 잠금을 해제할 수 있습니다. Bitmap 객체는 BitmapData 객체의 잠금이 해제될 때까지 변경되지 않습니다. 성능을 높이려면 setPixel() 또는 setPixel32() 메서드를 여러 차례 호출하기 전과 후에 이 메서드를 unlock() 메서드와 함께 사용합니다. lock() 및 unlock() 호출은 화면이 불필요하게 업데이트되지 않도록 방지합니다.

참고: 표시 목록이 아니라 비트맵에서 픽셀을 처리할 때(이중 버퍼링) 이 기법으로 성능이 향상되지 않는 경우가 있습니다. 비트맵 객체에서 비트맵 버퍼를 참조하지 않는 경우 lock() 및 unlock()을 사용해도 성능이 향상되지 않습니다. Flash Player는 버퍼가 참조되지 않는지와 비트맵이 화면에 렌더링되지 않는지를 감지합니다.

픽셀을 반복하는 메서드(예: getPixel(), getPixel32(), setPixel() 및 setPixel32())는 특히 휴대 장치에서 속도가 느려질 수 있습니다. 가능한 경우 한 번의 호출로 모든 픽셀을 가져오는 메서드를 사용하십시오. 픽셀 읽기에는 getPixels() 메서드보다 빠른 getVector() 메서드를 사용합니다. 또한 Vector 객체가 더 빨리 실행될 가능성이 높으므로 가능하면 Vector 객체를 사용하는 API를 사용하는 것이 좋습니다.

일반 표현식

기본 문자열 찾기 및 추출을 위해 일반 표현식 대신 indexOf(), substr() 또는 substring()과 같은 String 클래스 메서드를 사용합니다.

일반 표현식을 사용하여 수행할 수 있는 일부 작업은 String 클래스의 메서드를 사용해서도 수행할 수 있습니다. 예를 들어, 문자열에 다른 문자열이 포함되는지 여부를 확인하려는 경우 String.indexOf() 메서드나 일반 표현식을 사용할 수 있습니다. 하지만 가능하다면 String 클래스 메서드를 사용하는 것이 상응하는 일반 표현식보다 속도도 빠르며 다른 객체를 만들 필요가 없습니다.

결과에서 그룹의 내용을 격리하지 않으면서 요소를 그룹화하려면 일반 표현식에서 그룹(“(xxxx)”) 대신 캡처하지 않는 그룹(“(?:xxxx)”)을 사용합니다.

복잡성이 중간 정도인 일반 표현식에서는 종종 표현식의 일부를 그룹화합니다. 예를 들어, 다음 일반 표현식 패턴에서 괄호는 "ab" 텍스트 주위에 그룹을 만듭니다. 따라서 "+" 한정 기호는 단일 문자가 아닌 그룹에 적용됩니다.

```
/(ab)+/
```

기본적으로 각 그룹의 내용은 "캡처"됩니다. 일반 표현식을 실행한 결과의 일부로 패턴에서 각 그룹의 내용을 가져올 수 있습니다. 이러한 그룹 결과를 캡처하면 그룹 객체를 포함하기 위한 객체가 생성되기 때문에 시간이 더 오래 걸리고 메모리도 더 많이 필요합니다. 이에 대한 대안으로 시작 괄호 다음에 물음표와 콜론을 포함하여 비캡처 그룹 구문을 사용할 수 있습니다. 이 구문은 문자가 그룹으로 동작하지만 결과가 캡처되지 않도록 지정합니다.

```
/(?:ab)+/
```

비캡처 그룹 구문을 사용하는 것이 표준 그룹 구문을 사용하는 것보다 속도도 빠르고 메모리도 덜 사용합니다.

일반 표현식의 성능이 낮을 때는 다른 일반 표현식 패턴을 사용하는 것이 좋습니다..

일부 경우에는 동일 텍스트 패턴을 테스트하거나 식별하기 위해 두 개 이상의 일반 표현식 패턴을 사용할 수 있습니다. 여러 가지 이유로 인해 여러 패턴들 중 실행 속도가 빠른 패턴이 존재합니다. 일반 표현식으로 인해 코드 실행 속도가 필요한 것보다 더 느리다고 판단되는 경우에는 동일한 결과를 얻을 수 있는 대체 일반 표현식 패턴을 고려하십시오. 이러한 대체 패턴을 테스트하여 속도가 가장 빠른 패턴을 확인합니다.

기타 최적화

TextField 객체의 경우 += 연산자 대신 appendText() 메서드를 사용합니다.

TextField 클래스의 text 속성을 사용하여 작업하는 경우 += 연산자 대신 appendText()를 사용합니다. appendText() 메서드를 사용하면 성능이 향상됩니다.

다음 코드 예제에서는 += 연산자를 사용하고 루프가 완료되는 데 1120ms 걸립니다.

```
addChild ( myTextField );

myTextField.autoSize = TextFieldAutoSize.LEFT;
var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.text += "ActionScript 3";
}

trace( getTimer() - started );
// output : 1120
```

다음 예제에서는 += 연산자가 appendText() 메서드로 대체되었습니다.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.appendText ( "ActionScript 3" );
}

trace( getTimer() - started );
// output : 847
```

이 경우 코드가 완료되는 데 847ms가 걸립니다.


가능하면 루프 외부에서 텍스트 필드를 업데이트합니다.

이 코드는 간단한 기술을 사용하여 훨씬 더 최적화할 수 있습니다. 각 루프에서 텍스트 필드를 업데이트하면 내부 프로세스가 많이 사용됩니다. 간단히 문자열을 결합하여 루프 외부의 텍스트 필드에 할당하므로 코드 실행 시간이 크게 줄어듭니다. 이 경우 코드가 완료되는 데 2ms 걸립니다.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.text;

for (var i:int = 0; i< 1500; i++ )
{
    content += "ActionScript 3";
}

myTextField.text = content;

trace( getTimer() - started );
// output : 2
```

HTML 텍스트를 사용하여 작업하는 경우 이전 방식은 너무 느리기 때문에 일부 경우에 Flash Player에서 Timeout 예외가 발생할 수 있습니다. 예를 들어 기본 하드웨어가 너무 느린 경우 예외가 발생할 수 있습니다.

참고: Adobe® AIR®에서는 이 예외가 발생하지 않습니다.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.htmlText += "ActionScript <b>2</b>";
}

trace( getTimer() - started );
```

값을 루프 외부의 문자열에 할당하면 코드가 완료되는 데 29ms밖에 걸리지 않습니다.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.htmlText;

for (var i:int = 0; i< 1500; i++ )
{
    content += "<b>ActionScript<b> 3";
}

myTextField.htmlText = content;

trace ( getTimer() - started );
// output : 29
```

참고: Flash Player 10.1 및 AIR 2.5에서는 문자열에 더 적은 메모리가 사용되도록 String 클래스가 향상되었습니다.


가능하면 대괄호 연산자를 사용하지 않습니다.

대괄호 연산자를 사용하면 성능이 저하될 수 있습니다. 참조를 지역 변수에 저장하여 대괄호 연산자 사용을 피할 수 있습니다. 다음 코드 예제에서는 대괄호 연산자의 비효율적인 사용을 보여 줍니다.

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();

for ( i = 0; i< lng; i++ )
{
    arraySprite[i].x = Math.random()*stage.stageWidth;
    arraySprite[i].y = Math.random()*stage.stageHeight;
    arraySprite[i].alpha = Math.random();
    arraySprite[i].rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 16
```

다음과 같은 최적화된 버전에서는 대괄호 연산자의 사용이 줄어듭니다.

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();
var currentSprite:Sprite;

for ( i = 0; i< lng; i++ )
{
    currentSprite = arraySprite[i];
    currentSprite.x = Math.random()*stage.stageWidth;
    currentSprite.y = Math.random()*stage.stageHeight;
    currentSprite.alpha = Math.random();
    currentSprite.rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 9
```


가능하면 코드를 인라인으로 이동하여 코드 내 함수 호출 수를 줄입니다.

함수 호출은 리소스를 많이 소모할 수 있습니다. 코드를 인라인으로 이동하여 함수 호출 수를 줄여 봅니다. 코드를 인라인으로 이동하는 것은 최상의 성능을 얻는 데 유용한 최적화 방법입니다. 하지만 인라인 코드는 코드의 재사용을 어렵게 하므로 SWF 파일의 크기가 늘어날 수도 있다는 점에 유의해야 합니다. Math 클래스 메서드와 같은 일부 함수 호출을 사용하면 손쉽게 인라인으로 이동할 수 있습니다. 다음 코드에서는 Math.abs() 메서드를 사용하여 절대값을 계산합니다.

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = Math.abs ( currentValue );
}

trace( getTimer() - started );
// output : 70
```

Math.abs()에서 수행한 계산은 수동으로 수행 가능하며 인라인으로 이동할 수 있습니다.

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = currentValue > 0 ? currentValue : -currentValue;
}

trace( getTimer() - started );
// output : 15
```

함수 호출을 인라인으로 이동하면 코드가 4배 더 빨라질 수 있습니다. 이 방법은 많은 경우에 유용하지만 코드의 재사용 및 관리 측면에서 미치는 영향을 잘 알고 있어야 합니다.

참고: 코드 크기는 전반적인 플레이어 실행에 큰 영향을 미칩니다. 응용 프로그램에 많은 양의 ActionScript 코드가 포함된 경우 가상 머신에서 코드 확인 및 JIT 컴파일에 상당한 시간이 소요됩니다. 상속 계층 구조가 더 깊어지고 내부 캐시가 보다 과도하게 사용되는 경향이 있으므로 속성 조회 속도가 느려질 수 있습니다. 코드 크기를 줄이려면 Adobe® Flex® 프레임워크, TLF 프레임워크 라이브러리 또는 타사 대용량 ActionScript 라이브러리를 사용하지 마십시오.

루프의 명령문을 평가하지 않습니다.

루프 내부의 명령문을 평가하지 않으면 한층 더 최적화할 수 있습니다. 다음 코드는 배열에 대해 반복되지만 배열 길이가 각 반복에서 평가되기 때문에 최적화되어 있지 않습니다.

```
for (var i:int = 0; i< myArray.length; i++)
{
}
```

값을 저장하여 다시 사용하는 것이 좋습니다.

```
var lng:int = myArray.length;

for (var i:int = 0; i< lng; i++)
{
}
```

while 루프에 역순을 사용합니다.

역순 while 루프는 정방향 루프보다 더 빠릅니다.

```
var i:int = myArray.length;

while (--i > -1)
{
}
```

이러한 팁은 ActionScript를 최적화할 수 있는 몇 가지 방법을 제공하며, 한 줄의 코드가 성능 및 메모리에 어떤 영향을 줄 수 있는지 보여 줍니다. 이 외에도 가능한 ActionScript 최적화 방법은 많이 있습니다. 자세한 내용은 http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/ 링크를 참조하십시오.

5장: 렌더링 성능

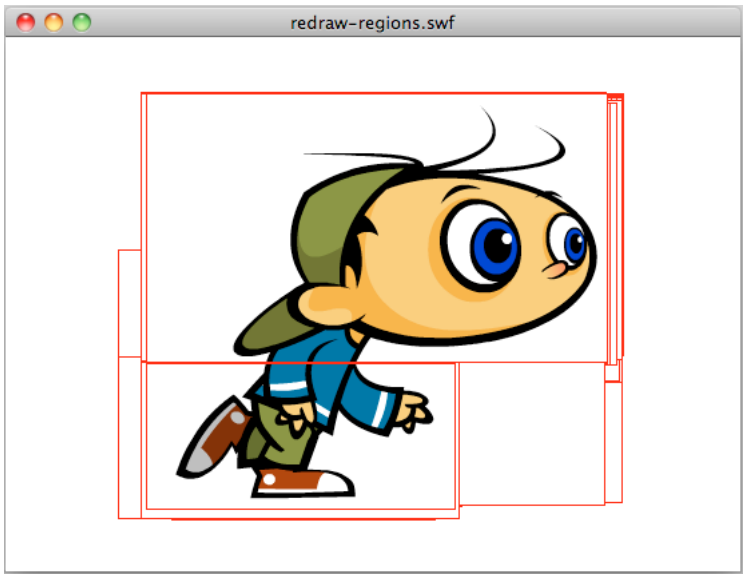
다시 그리기 영역

프로젝트를 만들 때는 항상 다시 그리기 영역 옵션을 사용하십시오.

렌더링을 향상시키려면 프로젝트를 만들 때 다시 그리기 영역 옵션을 사용하는 것이 중요합니다. 이 옵션을 사용하면 Flash Player에서 렌더링하고 처리하는 영역을 확인할 수 있습니다. 디버그 버전의 Flash Player에서 컨텍스트 메뉴의 [다시 그리기 영역 표시]를 선택하여 이 옵션을 활성화할 수 있습니다.

참고: [다시 그리기 영역 표시] 옵션은 Adobe AIR나 릴리스 버전의 Flash Player에서는 사용할 수 없습니다. Adobe AIR에서 컨텍스트 메뉴는 데스크톱 응용 프로그램에서만 사용할 수 있으며, [다시 그리기 영역 표시]와 같이 기본으로 제공되는 항목이나 표준 항목은 없습니다.

아래 이미지는 타임라인의 간단한 애니메이션 MovieClip에서 이 옵션이 활성화된 모습을 보여 줍니다.



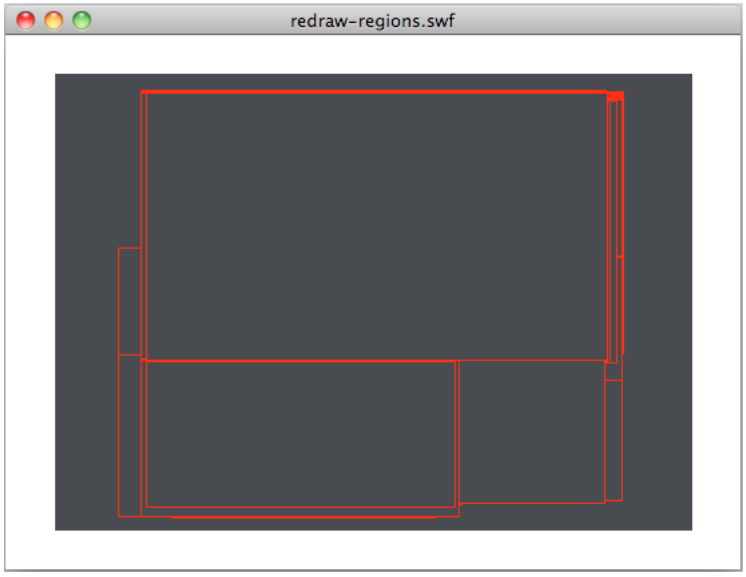
다시 그리기 영역 옵션이 활성화되어 있습니다

다음과 같이 flash.profiler.showRedrawRegions() 메서드를 사용하여 프로그래밍 방식으로 이 옵션을 활성화할 수도 있습니다.

```
// Enable Show Redraw Regions
// Blue color is used to show redrawn regions
flash.profiler.showRedrawRegions ( true, 0x0000FF );
```

Adobe AIR 응용 프로그램에서 이 메서드가 영역 다시 그리기 옵션을 사용할 수 있는 유일한 방법입니다.

영역 다시 그리기 기능을 사용하면 최적화할 수 있는 기회를 포착할 수 있습니다. 일부 표시 객체는 표시되지 않더라도 계속 렌더링 중이기 때문에 여전히 CPU 주기를 사용할 수 있다는 점을 기억하십시오. 다음 이미지는 이러한 아이디어를 보여 줍니다. 검정 벡터 모양이 애니메이션이 적용된 움직이는 문자를 가립니다. 이미지는 표시 객체가 표시 목록에서 제거되지 않았으며 계속 렌더링되고 있음을 보여 줍니다. 이로 인해 CPU 주기가 낭비됩니다.



다시 그려진 영역입니다

성능을 향상시키려면 숨겨진 실행 문자의 visible 속성을 false로 설정하거나 표시 목록에서 모두 제거해야 합니다. 또한 해당 타임라인도 중지해야 합니다. 이렇게 하면 해당 표시 객체가 중단되고 최소한의 CPU 전원만 사용하게 됩니다.

전체 개발 주기 동안 다시 그리기 영역 옵션을 사용하십시오. 이 옵션을 사용하면 프로젝트가 끝날 때 이전에 놓쳤을 수도 있는 불필요한 다시 그리기 영역 및 최적화 영역을 방지할 수 있습니다.

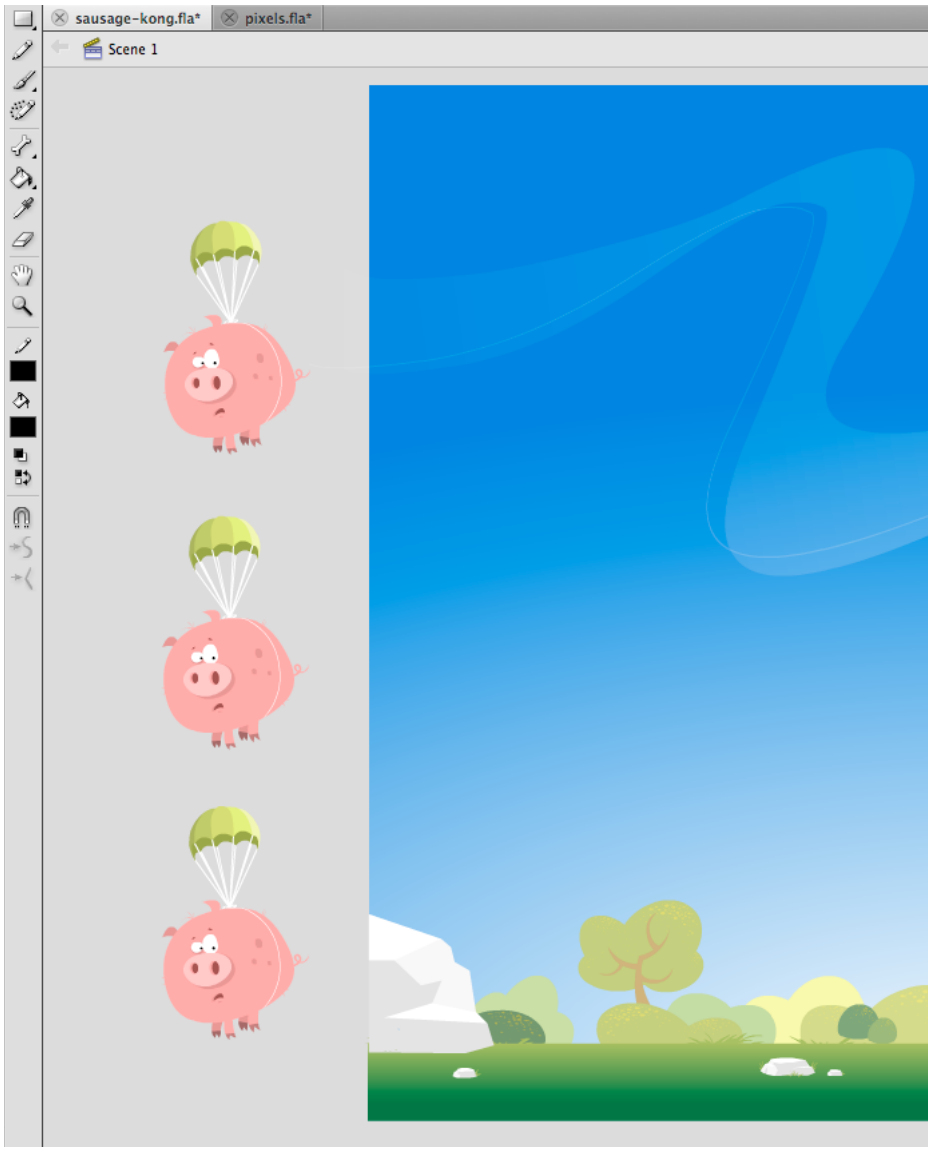
기타 도움말 항목

25페이지의 “객체 표시 제거 및 표시 제거 취소”

오프 스테이지 내용

오프 스테이지 내용을 배치하지 않도록 하십시오. 대신, 필요한 경우 객체를 표시 목록에 배치하십시오.

가능한 경우 오프 스테이지의 그래픽 내용을 배치하지 않도록 하십시오. 디자이너와 개발자는 응용 프로그램의 수명 내에 에셋을 다시 사용할 목적으로 오프 스테이지 요소를 배치하는 것이 일반적입니다. 이러한 일반적인 기법이 다음 그림에 잘 나타나 있습니다.



오프 스테이지 내용

오프 스테이지 요소는 화면에 표시되지도 않고 렌더링되지 않은 경우라도 표시 목록에 계속 존재합니다. 런타임에서는 해당 요소가 계속 오프 스테이지 내용이며 사용자가 상호 작용하지 않는다는 것을 확인하기 위해 해당 요소에 대한 내부 테스트를 계속 수행합니다. 그러므로 가능한 한 오프 스테이지 객체를 배치하지 말고 표시 목록에서 제거하십시오.

동영상 품질

적절한 스테이지 품질 설정을 사용하여 렌더링을 향상시킵니다.

휴대 전화와 같이 화면이 작은 휴대 장치용 내용을 개발할 때는 데스크톱 응용 프로그램을 개발할 때보다 이미지 품질이 중요하지 않습니다. 스테이지 품질을 적절한 설정으로 구성하면 렌더링 성능을 향상시킬 수 있습니다.

스테이지 품질에 대해 사용할 수 있는 설정은 다음과 같습니다.

- StageQuality.LOW: 모양보다 재생 속도에 중점을 두어 앤티앨리어싱을 사용하지 않습니다. 이러한 설정은 데스크톱용이나 TV용 Adobe AIR에서 지원되지 않습니다.
- StageQuality.MEDIUM: 앤티앨리어싱을 일부 적용하지만 크기가 조절된 비트맵을 다듬지는 않습니다. 이 설정은 휴대 장치용 AIR에서 기본값이지만 데스크톱 또는 TV용 AIR에서는 지원되지 않습니다.
- StageQuality.HIGH: (데스크톱의 기본값) 재생 속도보다 모양을 우선적으로 고려하며 항상 앤티앨리어싱을 사용합니다. SWF 파일에 애니메이션이 포함되어 있지 않으면 크기가 조절된 비트맵이 다듬어지지만 애니메이션이 포함되어 있으면 비트맵이 다듬어지지 않습니다.
- StageQuality.BEST: 최상의 품질로 표시하고 재생 속도는 고려하지 않습니다. 모든 출력이 앤티앨리어싱되며 크기가 조절된 비트맵은 항상 다듬어집니다.

StageQuality.MEDIUM을 사용하면 대개 휴대 장치의 응용 프로그램에 적절한 품질이 제공되지만 경우에 따라서는 StageQuality.LOW를 사용하여 적절한 품질을 제공할 수 있습니다. Flash Player 8부터는 스테이지 품질이 LOW로 설정된 경우에도 앤티앨리어싱이 적용된 텍스트가 정확하게 렌더링될 수 있습니다.

참고: 일부 휴대 장치에서는 품질이 HIGH로 설정된 경우에도 Flash Player 응용 프로그램의 더 나은 성능을 위해 MEDIUM이 사용되기도 합니다. 하지만 품질을 HIGH로 설정해도 대부분 큰 차이가 나타나지 않습니다. 일반적으로 휴대 장치 화면의 dpi가 더 높기 때문입니다. dpi는 장치에 따라 달라질 수 있습니다.

다음 그림에서 동영상 품질은 MEDIUM으로 설정되고 텍스트 렌더링은 애니메이션을 위한 앤티앨리어싱으로 설정되었습니다.

중간 스테이지 품질과 애니메이션을 위한 앤티앨리어싱으로 설정된 텍스트 렌더링

적절한 텍스트 렌더링 설정이 사용되고 있지 않기 때문에 스테이지 품질 설정이 텍스트 품질에 영향을 줍니다.

런타임에서는 가독성을 높이기 위해 텍스트 렌더링을 앤티앨리어싱으로 설정할 수 있습니다. 이 설정은 사용하는 스테이지 품질 설정에 관계없이 앤티앨리어싱된 텍스트의 완벽한 품질을 유지합니다.

낮은 스테이지 품질과 가독성을 위한 앤티앨리어싱으로 설정된 텍스트 렌더링

텍스트 렌더링을 비트맵 텍스트(앤티앨리어싱 없음)로 설정하여 같은 렌더링 품질을 얻을 수 있습니다.

낮은 스테이지 품질과 비트맵 텍스트(앤티앨리어싱 없음)로 설정된 텍스트 렌더링

마지막 두 예제에서는 사용하는 스테이지 품질 설정에 관계없이 고품질 텍스트를 구현할 수 있음을 보여 줍니다. 이 기능은 Flash Player 8부터 사용 가능했으며 이제 휴대 장치에서도 사용할 수 있습니다. Flash Player 10.1에서는 일부 장치에서 StageQuality.MEDIUM으로 자동 전환하여 성능을 향상시킵니다.

알파 블렌딩

가능하면 alpha 속성을 사용하지 않습니다.

alpha 속성(예: 페이드 효과)을 사용하는 경우 알파 블렌딩이 필요한 효과를 사용하지 마십시오. 표시 객체가 알파 블렌딩을 사용하는 경우 런타임에서는 겹쳐진 모든 표시 객체의 색상 값을 배경 색상과 결합하여 최종 색상을 결정해야 합니다. 그러므로 알파 블렌딩은 불투명 색상을 그리는 것보다 프로세서를 더 많이 소모할 수 있습니다. 이렇게 추가적인 계산 작업이 필요하므로 속도가 느린 장치에서 성능이 저하될 수 있습니다. 가능하면 alpha 속성을 사용하지 마십시오.

기타 도움말 항목

48페이지의 “비트맵 캐싱”

59페이지의 “텍스트 객체 렌더링”

응용 프로그램 프레임 속도

일반적으로 성능 향상을 위해서는 가능한 한 가장 낮은 프레임 속도를 사용하십시오..

1페이지의 “런타임 코드 실행 기본 사항”에 설명된 것처럼 응용 프로그램의 프레임 속도에 따라 각 "응용 프로그램 코드 및 렌더링" 주기에 사용할 수 있는 시간이 결정됩니다. 프레임 속도가 높을수록 매끄러운 애니메이션이 만들어집니다. 그러나 애니메이션 또는 기타 시각적 변경이 발생하지 않을 경우에는 대개 프레임 속도를 높게 설정할 이유가 없습니다. 프레임 속도가 높을수록 CPU 주기를 더 많이 사용하고 배터리 소모량도 더 많습니다.

다음은 응용 프로그램에 적합한 기본 프레임 속도를 결정하기 위한 몇 가지 일반적인 지침입니다.

- Flex 프레임워크를 사용 중인 경우 시작 프레임 속도를 기본값 그대로 두십시오..
- 응용 프로그램에 애니메이션이 포함되는 경우 적절한 프레임 속도는 최소 초당 20프레임입니다. 일반적으로는 초당 30프레임 이상으로 설정할 필요가 없습니다.
- 응용 프로그램에 애니메이션이 포함되지 않는 경우에는 일반적으로 프레임 속도가 초당 12프레임 정도면 충분합니다.

"가능한 한 가장 낮은 프레임 속도"는 응용 프로그램의 현재 작업에 따라 달라질 수 있습니다. 자세한 내용은 아래에서 설명하는 "응용 프로그램의 프레임 속도를 동적으로 변경"을 참조하십시오.

응용 프로그램에서 비디오가 유일한 동적 내용인 경우 낮은 프레임 속도를 사용하십시오..

로드된 비디오 내용은 런타임에서 응용 프로그램의 프레임 속도와 관계없이 고유한 프레임 속도로 재생됩니다. 응용 프로그램에 애니메이션이나 기타 빠르게 변경되는 시각적인 내용이 없는 경우 낮은 프레임 속도를 사용해도 사용자 인터페이스의 환경을 저해하지 않습니다.

응용 프로그램의 프레임 속도를 동적으로 변경하십시오.

프로젝트 또는 컴파일러 설정에서 응용 프로그램의 초기 프레임 속도를 정의하지만, 프레임 속도는 이 값으로 고정되지 않습니다. 프레임 속도는 Stage.frameRate 속성(또는 Flex의 경우 WindowedApplication.frameRate 속성)을 설정하여 변경할 수 있습니다.

응용 프로그램의 현재 요구 사항에 따라 프레임 속도를 변경하십시오. 예를 들어, 응용 프로그램에서 애니메이션이 수행되지 않을 때는 프레임 속도를 낮춥니다. 애니메이션이 전환이 시작될 때는 프레임 속도를 높입니다. 마찬가지로 응용 프로그램이 포커스를 잃은 후 백그라운드에서 실행되는 경우 보통 프레임 속도를 훨씬 더 낮출 수 있습니다. 따라서 사용자는 다른 응용 프로그램이나 작업에 집중할 수 있습니다.

다음은 서로 다른 작업 유형에 대해 적절한 프레임 속도를 결정하는 데 기초로 사용할 수 있는 몇 가지 일반적인 지침입니다.

- Flex 프레임워크를 사용 중인 경우 시작 프레임 속도를 기본값 그대로 두십시오..
- 애니메이션이 재생 중인 경우 프레임 속도를 최소 초당 20프레임으로 설정하십시오. 일반적으로는 초당 30프레임 이상으로 설정할 필요가 없습니다.
- 애니메이션이 실행 중이 아닌 경우 프레임 속도는 초당 12 프레임 정도면 충분합니다.
- 로드된 비디오는 응용 프로그램의 프레임 속도와 관계없이 고유한 프레임 속도로 재생됩니다. 응용 프로그램에서 움직이는 내용이 비디오 뿐인 경우 프레임 속도는 초당 12 프레임 정도면 충분합니다.
- 응용 프로그램에 입력 포커스가 없는 경우 프레임 속도는 초당 5 프레임 정도면 충분합니다.
- AIR 응용 프로그램이 표시되지 않는 경우 초당 2프레임 이하의 프레임 속도가 적절할 수 있습니다. 예를 들어 이러한 지침은 응용 프로그램을 최소화하는 경우에 적용됩니다. 또한 기본 윈도우의 visible 속성이 false인 경우 데스크톱 장치에도 적용됩니다.

Flex에서 만든 응용 프로그램의 경우 spark.components. WindowedApplication 클래스는 응용 프로그램의 프레임 속도에 대한 동적 변경을 기본적으로 지원합니다. backgroundFrameRate 속성은 응용 프로그램이 활성 상태가 아닐 때 응용 프로그램의 프레임 속도를 정의합니다. 기본값은 1으로, Spark 프레임워크를 사용하여 만든 응용 프로그램의 프레임 속도를 초당 1프레임으로 변경합니다. backgroundFrameRate 속성을 설정하여 백그라운드 프레임 속도를 변경할 수 있습니다. 이 속성을 다른 값으로 설정하거나, -1로 설정하여 자동 프레임 속도 제한을 해제합니다.

응용 프로그램의 프레임 속도를 동적으로 변경하는 데 대한 자세한 내용은 다음 문서들을 참조하십시오.

- Reducing CPU usage in Adobe AIR(Jonnie Hallman의 Adobe 개발자 센터 문서 및 샘플 코드)
- Writing well-behaved, efficient, AIR applications(Arno Gourdol의 문서 및 샘플 코드 응용 프로그램)

Grant Skinner가 프레임 속도 제한자 클래스를 만들었습니다. 응용 프로그램에서 이 클래스를 사용하여 응용 프로그램이 백그라운드에 있을 때 프레임 속도를 자동으로 줄일 수 있습니다. 자세한 정보를 확인하고 FramerateThrottler 클래스의 소스 코드를 다운로드하려면 http://gskinner.com/blog/archives/2009/05/idle_cpu_usage.html에서 Grant가 작성한 Idle CPU Usage in Adobe AIR and Flash Player 문서를 참조하십시오.

적응 프레임 속도

SWF 파일을 컴파일할 때 해당 동영상의 특정한 프레임 속도를 설정합니다. CPU 속도가 낮은 제한된 환경에서는 프레임 속도가 재생 중에 떨어지기도 합니다. 사용자의 프레임 속도를 적절히 유지하기 위해 런타임에서 일부 프레임의 렌더링을 건너뜁니다. 일부 프레임의 렌더링을 건너뛰면 프레임 속도가 적절한 값 이하로 낮아지지 않도록 유지됩니다.

참고: 이 경우 런타임은 프레임을 건너뛰지 않으며 프레임의 내용 렌더링만 건너뜁니다. 코드가 계속 실행되고 표시 목록이 업데이트되지만 업데이트가 화면에 표시되지는 않습니다. 런타임에서 프레임 속도를 안정적으로 유지할 수 없는 경우 건너뛸 프레임 수를 나타내는 fps 임계값을 지정할 수 있는 방법은 없습니다.

비트맵 캐싱


적절한 경우 복잡한 벡터 내용에 대해 비트맵 캐싱 기능을 사용합니다.

비트맵 캐싱 기능을 사용해야 최적화 작업을 효과적으로 수행할 수 있습니다. 이 기능은 벡터 객체를 캐시하고 내부적으로 비트맵으로 렌더링하며 이 비트맵을 렌더링에 사용합니다. 그 결과 렌더링 성능이 크게 향상될 수 있지만 상당한 양의 메모리가 필요할 수 있습니다. 복잡한 그래디언트 또는 텍스트와 같은 복잡한 벡터 내용에 비트맵 캐싱 기능을 사용합니다.

복잡한 벡터 그래픽(예: 텍스트 또는 그래디언트)이 포함된, 애니메이션이 적용된 객체에 대해 비트맵 캐싱을 설정하면 성능이 향상됩니다. 그러나 자체 타임라인 재생이 포함된 동영상 클립과 같은 표시 객체에 비트맵 캐싱이 활성화되어 있는 경우 반대의 결과가 나타납니다. 런타임이 각 프레임에서 캐시된 비트맵을 업데이트한 다음 화면에 다시 그려야 하는데, 이 작업에는 CPU 주기가 많이 필요합니다. 비트맵 캐싱 기능은 캐시된 비트맵을 한 번 생성한 다음 업데이트할 필요 없이 사용할 수 있는 경우에만 도움이 됩니다.

Sprite 객체에 대해 비트맵 캐싱을 설정하는 경우 런타임에서 캐시된 비트맵을 다시 생성하지 않고도 객체를 이동할 수 있습니다. 객체의 x 및 y 속성을 변경해도 다시 생성되지 않습니다. 그러나 객체를 회전하거나, 객체 크기를 조절하거나, 객체의 알파 값을 변경하면 런타임에서 캐시된 비트맵을 다시 생성하므로 성능이 저하됩니다.

참고: AIR 및 Packager for iPhone에 사용 가능한 DisplayObject.cacheAsBitmapMatrix 속성에는 이러한 제한이 없습니다. cacheAsBitmapMatrix 속성을 사용하면 비트맵을 다시 생성하지 않고도 표시 객체의 알파 값을 회전하고 크기 조절하고 기울이고 변경할 수 있습니다.

캐시된 비트맵은 일반 동영상 클립 인스턴스보다 더 많은 메모리를 소비할 수도 있습니다. 예를 들어 스테이지의 동영상 클립이 250 x 250 픽셀인 경우 캐시하지 않으면 1KB를 사용하지만 캐시하면 약 250KB를 사용합니다.

다음 예제에는 사과 이미지를 포함하는 Sprite 객체가 관련됩니다. 다음 클래스는 사과 기호에 연결됩니다.

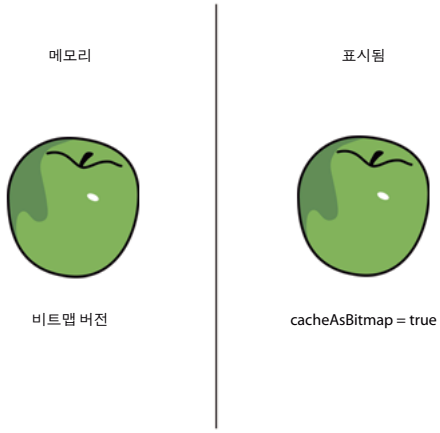
```
package org.bytearray.bitmap
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Apple extends Sprite
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function Apple ()
        {
            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener (Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

코드에서는 각 사과에 타임라인이 필요하지 않기 때문에 MovieClip 클래스 대신 Sprite 클래스를 사용합니다. 최상의 성능을 위해 가능하면 가장 리소스 소모가 적은 객체를 사용합니다. 다음으로 클래스는 아래 코드를 사용하여 인스턴스화됩니다.

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(MouseEvent.CLICK,createApples);
stage.addEventListener(KeyboardEvent.KEY_DOWN,cacheApples);

const MAX_NUM:int = 100;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

function createApples(e:MouseEvent):void
{
    for (var i:int = 0; i< MAX_NUM; i++)
    {
        apple = new Apple();

        holder.addChild(apple);
    }
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holder.numChildren;

        for (var i:int = 0; i < lng; i++)
        {
            apple = holder.getChildAt (i) as Apple;

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

사용자가 마우스를 클릭하면 사과가 캐시되지 않고 만들어집니다. 사용자가 C키(키 코드 67)를 누르면 사과 벡터가 비트맵으로 캐시되어 화면에 표시됩니다. 이 기술은 CPU가 느릴 경우 데스크톱 및 휴대 장치 모두에서 렌더링 성능을 크게 향상시킵니다.

하지만 비트맵 캐싱 기능을 사용하여 렌더링 성능이 향상되긴 해도 많은 양의 메모리가 빠르게 소모될 수 있습니다. 다음 다이어그램과 같이 객체가 캐시되는 즉시 객체의 표면이 투명 비트맵으로 캡처되어 메모리에 저장됩니다.

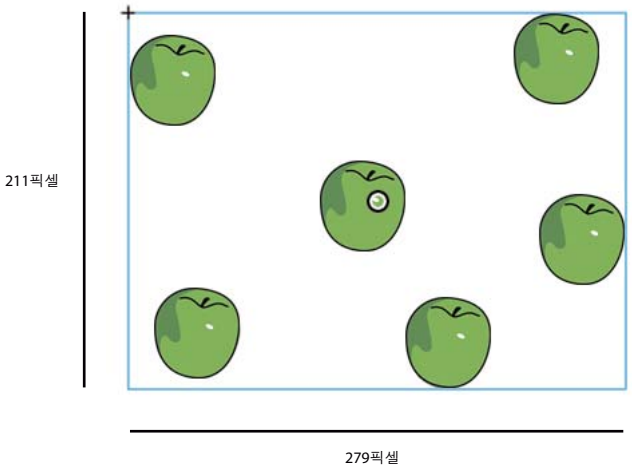


객체와 메모리에 저장된 객체의 표면 비트맵

Flash Player 10.1 및 AIR 2.5에서는 18페이지의 "필터 및 동적 비트맵 언로드"에 설명된 것과 동일한 방식으로 메모리 사용을 최적화합니다. 캐시된 표시 객체가 숨겨지거나 화면에 표시되지 않는 경우 잠시 사용되지 않는 동안 메모리의 비트맵이 해제됩니다.

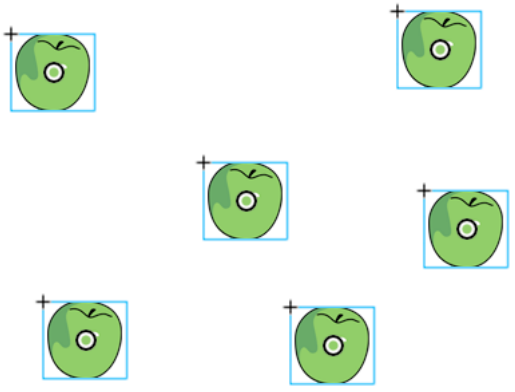
참고: 표시 객체의 opaqueBackground 속성이 특정 색상으로 설정되면 런타임은 표시 객체가 불투명한 것으로 간주하고, cacheAsBitmap 속성과 함께 사용되는 경우 런타임에서는 메모리에 투명하지 않은 32비트 비트맵을 만듭니다. 비트맵을 화면에 그리는 데 투명도가 필요하지 않으므로 성능 향상을 위해 알파 채널이 0xFF로 설정됩니다. 알파 블렌딩을 사용하지 않으면 렌더링이 훨씬 빨라집니다. 현재 화면 깊이가 16비트로 제한된 경우 메모리의 비트맵이 16비트 이미지로 저장됩니다. opaqueBackground 속성을 사용하더라도 비트맵 캐싱이 암시적으로 활성화되지는 않습니다.

메모리를 절약하려면 컨테이너가 아닌 각 표시 객체에서 cacheAsBitmap 속성을 사용하고 활성화합니다. 컨테이너에서 비트맵 캐싱을 활성화하면 메모리에서 최종 비트맵의 크기가 훨씬 더 커져 211 x 279 픽셀 크기의 투명 비트맵이 만들어집니다. 이미지에 약 229KB의 메모리가 사용됩니다.



컨테이너에서 비트맵 캐싱 활성화

또한 컨테이너를 캐싱하면 프레임에서 이동하는 사과가 있을 경우 메모리에서 전체 비트맵이 업데이트될 위험이 있습니다. 개별 인스턴스에서 비트맵 캐싱을 활성화하면 메모리에 7KB 표면 6개가 캐시되며 42KB의 메모리만 사용됩니다.



인스턴스에서 비트맵 캐싱 활성화

표시 목록을 통해 각 사과 인스턴스에 액세스하고 getChildAt() 메서드를 호출하면 참조가 Vector 객체에 저장되어 더 쉽게 액세스할 수 있습니다.

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(KeyboardEvent.KEY_DOWN, cacheApples);

const MAX_NUM:int = 200;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<Apple> = new Vector.<Apple>(MAX_NUM, true);

for (var i:int = 0; i< MAX_NUM; i++)
{
    apple = new Apple();

    holder.addChild(apple);

    holderVector[i] = apple;
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holderVector.length

        for (var i:int = 0; i < lng; i++)
        {
            apple = holderVector[i];

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

비트맵 캐싱은 캐시된 내용이 각 프레임에서 회전, 크기 조절 또는 변경되지 않는 경우에 렌더링을 향상시킵니다. 하지만 x축 및 y축 평행 이동이 아닌 변환의 경우 렌더링이 향상되지 않습니다. 이러한 경우 Flash Player에서는 표시 객체에서 발생하는 모든 변형의 캐시된 비트맵 복사본을 업데이트합니다. 캐시된 복사본을 업데이트하면 CPU 사용이 증가되고 성능이 저하되며 배터리 사용이 늘어날 수 있습니다. 다시 한 번 말하지만 AIR 또는 Packager for iPhone의 cacheAsBitmapMatrix 속성에는 이러한 제한이 없습니다.

다음 코드에서는 이동 메서드의 알파 값을 변경하며 이로 인해 모든 프레임에서 사과의 불투명도가 변경됩니다.

```
private function handleMovement(e:Event):void
{
        alpha = Math.random();
        x -= (x - destinationX)*.5;
        y -= (y - destinationY)*.5;

        if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
            initPos();
}
```

비트맵 캐싱을 사용하면 성능이 저하될 수 있습니다. 알파 값을 변경하면 알파 값이 수정될 때마다 메모리에서 캐시된 비트맵이 강제로 업데이트됩니다.

캐시된 동영상 클립의 재생 헤드가 이동할 때마다 업데이트되는 비트맵이 필터에 사용됩니다. 따라서 필터를 사용하면 cacheAsBitmap 속성이 true로 자동 설정됩니다. 다음 그림에서는 애니메이션이 적용된 동영상 클립을 보여 줍니다.

애니메이션이 적용된 동영상 클립

필터를 사용하면 성능 문제가 발생할 수 있으므로 애니메이션이 적용된 내용에는 필터를 사용하지 않습니다. 다음 그림에서 디자이너는 그림자 필터를 추가합니다.

그림자 필터로 애니메이션이 적용된 동영상 클립

따라서 동영상 클립 내의 타임라인이 재생 중인 경우 비트맵이 다시 생성되어야 합니다. 또한 단순 x 또는 y 변형 이외의 방식으로 내용이 수정되는 경우에도 비트맵이 다시 생성되어야 합니다. 각 프레임에서는 비트맵이 강제로 다시 그려지는데, 이로 인해 필요한 CPU 리소스가 늘어나고 성능이 저하되며 배터리가 더 많이 소모됩니다.

다음 교육 비디오에서는 Paul Trani가 Flash Professional 및 ActionScript를 통해 비트맵이 사용된 그래픽을 최적화하는 예를 보여 줍니다.

- 그래픽 최적화
- ActionScript를 사용하여 그래픽 최적화

AIR의 캐시된 비트맵 변형 행렬

모바일 AIR 응용 프로그램에서 캐시된 비트맵을 사용하는 경우 cacheAsBitmapMatrix 속성을 설정하십시오.

AIR 모바일 프로파일에서 표시 객체의 cacheAsBitmapMatrix 속성에 Matrix 객체를 할당할 수 있습니다. 이 속성을 설정하면 캐시된 비트맵을 다시 생성하지 않고도 객체에 2D 변형을 적용할 수 있습니다. 또한 캐시된 비트맵을 다시 생성하지 않고도 알파 속성을 변경할 수도 있습니다. cacheAsBitmap 속성은 true로 설정되어야 하고, 객체에는 3D 속성이 설정되면 안 됩니다.

cacheAsBitmapMatrix 속성을 설정하면 표시 객체가 화면에 없거나 숨겨져 있거나 visible 속성이 false로 설정된 경우라도 캐시된 비트맵이 생성됩니다. 다른 변형을 포함한 행렬 객체를 사용하여 cacheAsBitmapMatrix 속성을 다시 설정해도 캐시된 비트맵이 다시 생성됩니다.

cacheAsBitmapMatrix 속성에 적용한 행렬 변형은 비트맵 캐시에 렌더링되는 것처럼 표시 객체에 적용됩니다. 그러므로 변형에 2배의 크기 조절이 포함되는 경우 비트맵 렌더링은 벡터 렌더링 크기의 두 배가 됩니다. 렌더러는 캐시된 비트맵에 반전 변형을 적용하여 최종적으로 동일하게 표시되도록 합니다. 렌더링 품질은 낮아질지라도 메모리 사용을 줄이기 위해 캐시된 비트맵을 더 작

은 크기로 조절할 수 있습니다. 또한 메모리 사용을 감수하면서 렌더링 품질을 높이기 위해 비트맵을 더 큰 크기로 조절하는 경우도 있습니다. 하지만 다음 예에 나타나는 것처럼 일반적으로 모양이 변경되는 것을 피하기 위해 변형을 적용하지 않는 행렬인 단위 행렬을 사용합니다.

```
displayObject.cacheAsBitMap = true;
displayObject.cacheAsBitmapMatrix = new Matrix();
```

cacheAsBitmapMatrix 속성을 설정하면 비트맵을 다시 생성하지 않고도 객체의 크기를 조절하거나 객체를 기울이고, 회전하고, 변환할 수 있습니다.

또한 0과 1 사이의 범위에서 알파 값을 변경할 수도 있습니다. 색상 변형과 함께 transform.colorTransform 속성을 통해 알파 값을 변경하는 경우 변형 객체에 사용하는 알파는 0에서 255 사이여야 합니다. 다른 방식으로 색상 변형을 변경하면 캐시된 비트맵이 다시 생성됩니다.

휴대 장치용으로 만든 내용에서 cacheAsBitmap을 true로 설정할 때마다 항상 cacheAsBitmapMatrix 속성을 설정해야 합니다. 하지만 이에 대한 부작용으로, 일반 벡터 렌더링에 비해 객체를 회전하거나 크기 조절하거나 기울인 후 최종 렌더링에 비트맵 크기 조절 또는 앨리어싱 아티팩트가 나타날 수 있다는 점을 고려하십시오.

수동 비트맵 캐싱


BitmapData 클래스를 통해 사용자 정의 비트맵 캐싱 비헤이비어를 만듭니다.

다음 예제에서는 표시 객체의 래스터화된 단일 비트맵 버전을 사용하고 같은 BitmapData 객체를 참조합니다. 각 표시 객체의 크기를 조절하는 경우 메모리의 원래 BitmapData 객체가 업데이트되지 않고 다시 그려지지 않습니다. 이 방법은 CPU 리소스를 절약하고 응용 프로그램 실행 속도를 향상시킵니다. 표시 객체의 크기를 조절하면 포함된 비트맵이 확대됩니다.

다음은 업데이트된 BitmapApple 클래스입니다.

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super(buffer);

            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
```

```
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            alpha = Math.random();

            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if ( Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

알파 값은 각 프레임에서 계속 수정됩니다. 다음 코드에서는 각 BitmapApple 인스턴스에 원본 소스 버퍼를 전달합니다.

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x,-bounds.y);

var buffer:BitmapData = new BitmapData(source.width+1, source.height+1, true, 0);
buffer.draw(source,mat);

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}
```

메모리에서 하나의 캐시된 비트맵만 사용되고 모든 BitmapApple 인스턴스에서 공유되므로 이 기법에는 적은 양의 메모리만 사용됩니다. 또한 알파, 회전 또는 크기 조절 등 BitmapApple 인스턴스가 변경되더라도 원본 소스 비트맵이 업데이트되지 않습니다. 이 기술을 사용하면 결과적으로 성능이 저하되는 것을 막을 수 있습니다.

매끄러운 최종 비트맵을 얻으려면 smoothing 속성을 true로 설정합니다.

```
public function BitmapApple(buffer:BitmapData)
{
    super (buffer);

    smoothing = true;

    addEventListener(Event.ADDED_TO_STAGE, activation);
    addEventListener(Event.REMOVED_FROM_STAGE, deactivation);
}
```

스테이지 품질을 조정하면 성능도 향상시킬 수 있습니다. 래스터화 전에 스테이지 품질을 HIGH로 설정한 후 나중에 LOW로 전환합니다.

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild ( holder );

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds ( source );

var mat:Matrix = new Matrix();
mat.translate ( -bounds.x, -bounds.y );

var buffer:BitmapData = new BitmapData ( source.width+1, source.height+1, true, 0 );

stage.quality = StageQuality.HIGH;

buffer.draw ( source, mat );

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++ )
{
    bitmapApple = new BitmapApple( buffer );

    holderVector[i] = bitmapApple;

    holder.addChild ( bitmapApple );
}
```

벡터를 비트맵에 그리기 전과 후에 스테이지 품질을 전환하면 화면에 앤티앨리어싱된 내용을 효과적으로 표시할 수 있습니다. 이 기법은 최종 스테이지 품질과 상관없이 효과적일 수 있습니다. 예를 들어 스테이지 품질이 LOW로 설정된 경우에도 앤티앨리어싱된 텍스트가 포함된 앤티앨리어싱 비트맵을 표시할 수 있습니다. 이 기술은 cacheAsBitmap 속성과 함께 사용할 수 없습니다. 이 경우 스테이지 품질을 LOW로 설정하면 벡터 품질이 업데이트되고, 이에 따라 비트맵 표면이 메모리에서 업데이트되고 최종 품질이 업데이트됩니다.

비헤이비어 격리


가능하면 단일 핸들러에서 Event.ENTER_FRAME 이벤트와 같은 이벤트를 분리합니다.

Apple 클래스의 Event.ENTER_FRAME 이벤트를 단일 핸들러에 격리하여 코드를 더욱 최적화할 수 있습니다. 이 기법을 사용하면 CPU 리소스가 절약됩니다. 다음 예제에서는 BitmapApple 클래스에서 더 이상 이동 비헤이비어를 처리하지 않는 이러한 다른 방식을 보여 줍니다.

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super (buffer);

            smoothing = true;
        }
}
```

다음 코드는 단일 핸들러에서 사과를 인스턴스화하고 사과의 이동을 처리합니다.

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x,-bounds.y);

stage.quality = StageQuality.BEST;

var buffer:BitmapData = new BitmapData(source.width+1,source.height+1, true,0);
buffer.draw(source,mat);

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    bitmapApple.destinationX = Math.random()*stage.stageWidth;
    bitmapApple.destinationY = Math.random()*stage.stageHeight;

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}

stage.addEventListener(Event.ENTER_FRAME,onFrame);
```

```
var lng:int = holderVector.length

function onFrame(e:Event):void
{
        for (var i:int = 0; i < lng; i++)
        {
            bitmapApple = holderVector[i];
            bitmapApple.alpha = Math.random();

            bitmapApple.x -= (bitmapApple.x - bitmapApple.destinationX) *.5;
            bitmapApple.y -= (bitmapApple.y - bitmapApple.destinationY) *.5;

            if (Math.abs(bitmapApple.x - bitmapApple.destinationX ) < 1 &&
                Math.abs(bitmapApple.y - bitmapApple.destinationY ) < 1)
            {
                    bitmapApple.destinationX = Math.random()*stage.stageWidth;
                    bitmapApple.destinationY = Math.random()*stage.stageHeight;
            }
        }
}
```

각 사과를 이동하는 200개의 핸들러 대신 이동을 처리하는 단일 Event.ENTER_FRAME 이벤트가 생성됩니다. 전체 애니메이션을 쉽게 일시 정지할 수 있으므로 게임에서 유용할 수 있습니다.

예를 들어 간단한 게임에서 다음 핸들러를 사용할 수 있습니다.

```
stage.addEventListener(Event.ENTER_FRAME, updateGame);
function updateGame (e:Event):void
{
    gameEngine.update();
}
```

다음 단계는 사과가 마우스 또는 키보드와 상호 작용하도록 하는 것입니다. 여기에는 BitmapApple 클래스의 수정이 필요합니다.

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;

    public class BitmapApple extends Sprite
    {
        public var destinationX:Number;
        public var destinationY:Number;
        private var container:Sprite;
        private var containerBitmap:Bitmap;

        public function BitmapApple(buffer:BitmapData)
        {
            container = new Sprite();
            containerBitmap = new Bitmap(buffer);
            containerBitmap.smoothing = true;
            container.addChild(containerBitmap);
            addChild(container);
        }
}
```

일반적인 Sprite 객체와 같이 대화형의 BitmapApple 인스턴스가 생성됩니다. 그러나 인스턴스는 표시 객체가 변형되는 경우 다시 샘플링되지 않는 단일 비트맵에 연결됩니다.

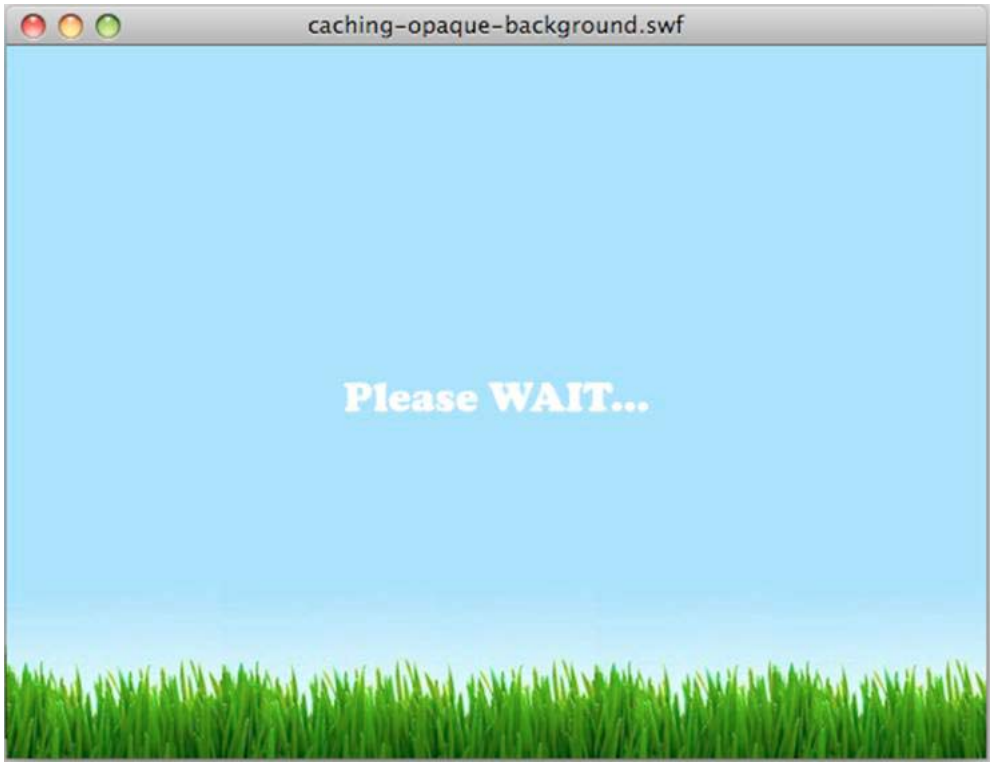
텍스트 객체 렌더링

비트맵 캐싱 기능과 opaqueBackground 속성을 사용하여 텍스트 렌더링 성능을 향상시킵니다.

Flash Text Engine을 사용하면 상당한 최적화를 달성할 수 있습니다. 하지만 텍스트 한 줄을 표시하기 위해 많은 클래스가 필요합니다. 이로 인해 TextLine 클래스를 사용하여 편집 가능한 텍스트 필드를 만들면 대량의 메모리와 많은 줄의 ActionScript 코드가 요구됩니다. TextLine 클래스는 더 빨리 렌더링되고 더 적은 메모리를 요구하는 정적 텍스트와 편집 가능하지 않은 텍스트에 가장 적절합니다.

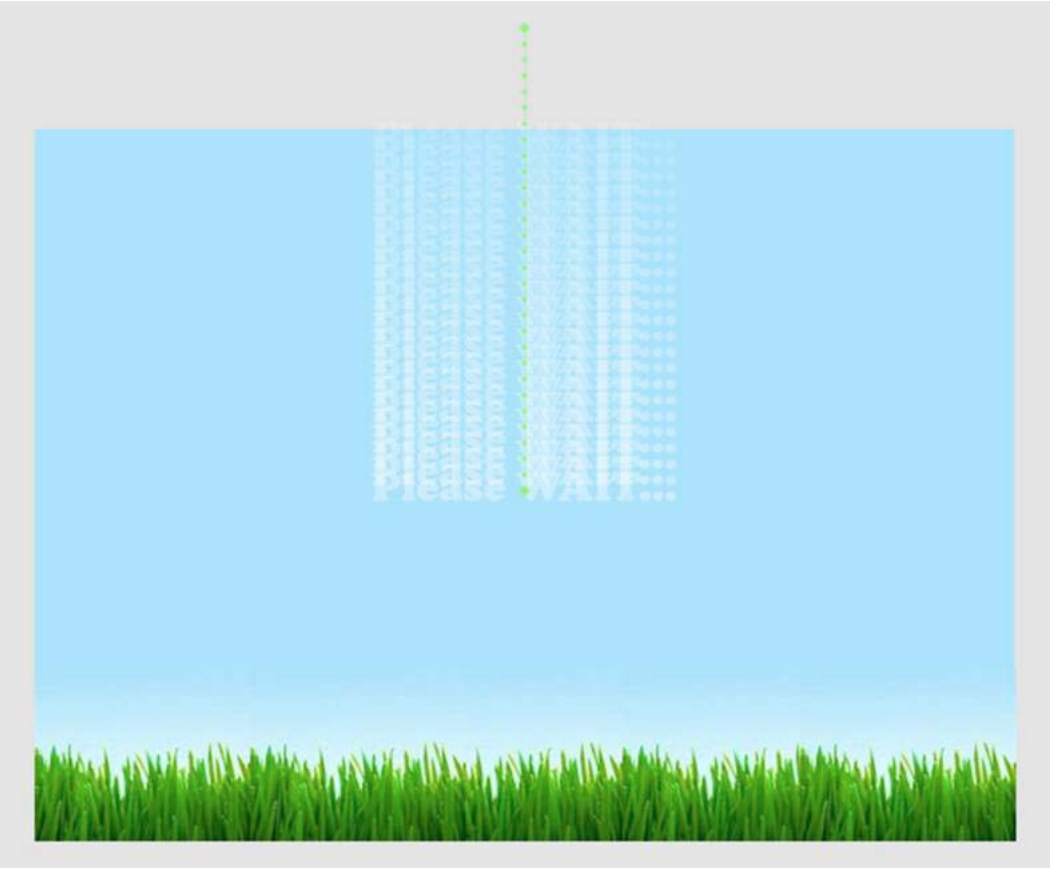
비트맵 캐싱 기능을 사용하면 벡터 내용을 비트맵으로 캐시하여 렌더링 성능을 높일 수 있습니다. 이 기능은 복잡한 벡터 내용에 유용하며 렌더링을 위해 처리가 필요한 텍스트 내용과 함께 사용하는 경우에도 도움이 됩니다.

다음 예제는 비트맵 캐싱 기능과 opaqueBackground 속성을 사용하여 렌더링 성능을 향상시키는 방법을 보여 줍니다. 다음 그림에서는 사용자가 무언가를 로드하는 동안 표시될 수 있는 일반적인 시작 화면을 보여 줍니다.



시작 화면

다음 그림에서는 TextField 객체에 프로그래밍 방식으로 적용되는 부드럽게 효과를 보여 줍니다. 텍스트가 장면 위쪽에서 가운데까지 부드럽게 천천히 내려옵니다.



텍스트를 부드럽게

다음 코드는 부드럽게 효과를 만듭니다. preloader 변수는 현재 대상 객체를 저장하여 속성 조회를 최소화하므로 성능이 저하될 수 있습니다.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

var destX:Number=stage.stageWidth/2;
var destY:Number=stage.stageHeight/2;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(preloader.y-destY)<1)
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
}
```

여기에서 Math.abs() 함수를 인라인으로 이동하여 함수 호출 수를 줄이고 성능을 더욱 향상시킬 수 있습니다. 가장 좋은 방법은 destX 및 destY 속성에 int 유형을 사용하여 고정 소수점 값을 사용하는 것입니다. int 유형을 사용하면 Math.ceil() 또는 Math.round()와 같이 느린 메서드를 통해 값을 수동으로 반올림할 필요 없이 완벽한 픽셀 물리기를 구현할 수 있습니다. 값을 지속적으로 반올림하면 객체가 매끄럽게 이동하지 않으므로 이 코드에서는 좌표를 int로 반올림하지 않습니다. 모든 프레임에서 좌표가 가장 가까운 반올림된 정수로 물려지므로 객체가 매끄럽지 않게 이동할 수 있습니다. 그러나 이 기법은 표시 객체의 최종 위치를 설정할 때 유용할 수 있습니다. 다음 코드를 사용하지 마십시오.

```
// Do not use this code
var destX:Number = Math.round ( stage.stageWidth / 2 );
var destY:Number = Math.round ( stage.stageHeight / 2);
```

다음 코드는 훨씬 빠릅니다.

```
var destX:int = stage.stageWidth / 2;
var destY:int = stage.stageHeight / 2;
```

비트 시프트 연산자를 사용하여 값을 분할하면 이전 코드를 훨씬 더 최적화할 수 있습니다.

```
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
```

비트맵 캐싱 기능은 런타임에서 동적 비트맵을 사용하여 객체를 손쉽게 렌더링하도록 합니다. 현재 예제에서는 TextField 객체가 포함된 동영상 클립이 캐시됩니다.

```
wait_mc.cacheAsBitmap = true;
```

성능을 향상시키는 다른 한 방법은 알파 투명도를 제거하는 것입니다. 알파 투명도는 이전 코드에서처럼 투명 비트맵 이미지를 그릴 때 런타임의 로드를 증가시킵니다. opaqueBackground 속성을 통해 특정 색상을 배경색으로 지정하여 이 문제를 피할 수 있습니다.

opaqueBackground 속성을 사용하는 경우 메모리에 만들어진 비트맵 표면에 여전히 32비트가 사용됩니다. 그러나 알파 오프셋이 255로 설정되고 투명도가 사용되지 않습니다. 따라서 opaqueBackground 속성은 메모리 사용을 줄이지 않지만 비트맵 캐싱 기능을 사용하는 경우 렌더링을 향상시킵니다. 다음 코드에는 모든 최적화가 포함되어 있습니다.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;

// Set the background to the color of the scene background
wait_mc.opaqueBackground = 0x8AD6FD;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;

function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
        e.currentTarget.removeEventListener ( Event.ENTER_FRAME, movePosition );
}
```

이제 애니메이션이 최적화되었으며 투명도를 제거하여 비트맵 캐싱이 최적화되었습니다. 휴대 장치에서 비트맵 캐싱 기능을 사용하는 동안 애니메이션의 다양한 상태에서 스테이지 품질을 LOW 및 HIGH로 전환하는 것을 고려하십시오.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;
wait_mc.opaqueBackground = 0x8AD6FD;

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth>>1;
var destY:int = stage.stageHeight>>1;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(e.currentTarget.y-destY)<1)
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
    }
}
```

하지만 이 경우에 스테이지 품질을 변경하면 런타임에서 현재 스테이지 품질에 맞춰 TextField 객체의 비트맵 표면을 강제로 다시 생성하게 됩니다. 이러한 이유로 비트맵 캐싱 기능을 사용할 때는 스테이지 품질을 변경하지 않는 것이 가장 좋습니다.

여기서 수동 비트맵 캐싱 방법을 사용할 수도 있습니다. 동영상 클립을 투명하지 않은 BitmapData 객체로 그리면 런타임에서 비트맵 표면을 강제로 다시 생성하는 일 없이 opaqueBackground 속성을 시뮬레이션할 수 있습니다.

이 방법은 시간이 지나도 변경되지 않는 내용에 매우 효과적입니다. 하지만 텍스트 필드의 내용이 변경될 수 있는 경우에는 다른 전략을 사용해야 합니다. 예를 들어 응용 프로그램이 로드된 정도를 나타내는 비율에 따라 계속해서 업데이트되는 텍스트 필드를 생각해 봅니다. 텍스트 필드 또는 이 필드에 포함된 표시 객체가 비트맵으로 캐시된 경우 내용이 변경될 때마다 해당 표면이 생성되어야 합니다. 표시 객체 내용이 지속적으로 변경되므로 여기에서 수동 비트맵 캐싱을 사용할 수 없습니다. 이러한 상수 변경으로 인해 BitmapData.draw() 메서드를 수동으로 호출하여 캐시된 비트맵을 업데이트해야 할 수 있습니다.

Flash Player 8(및 AIR 1.0)부터는 스테이지 품질 값에 관계없이 렌더링이 가독성을 위한 앤티앨리어싱으로 설정된 텍스트 필드가 완벽하게 앤티앨리어싱이 적용된 상태로 유지됩니다. 이 방법은 메모리를 더 적게 소모하지만 CPU 처리를 더 많이 요구하며 비트맵 캐싱 기능보다 약간 더 느리게 렌더링됩니다.

다음 코드에서는 이 방법을 사용합니다.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;
function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener ( Event.ENTER_FRAME, movePosition );
    }
}
```

모션 상태의 텍스트에는 이 옵션(가독성을 위한 앤티앨리어싱)을 사용하지 않는 것이 좋습니다. 텍스트의 크기를 조절할 때 이 옵션으로 인해 텍스트가 정렬된 상태로 유지되도록 시도하므로 이동 효과가 생성됩니다. 하지만 표시 객체의 내용이 지속적으로 변경되고 크기가 조절된 텍스트가 필요한 경우 품질을 LOW로 설정하여 모바일 응용 프로그램의 성능을 향상시킬 수 있습니다. 모션이 끝나면 품질을 HIGH로 다시 전환합니다.

GPU

Flash Player 응용 프로그램의 GPU 렌더링

Flash Player 10.1의 중요한 새 기능은 GPU를 사용하여 휴대 장치에서 그래픽 내용을 렌더링할 수 있다는 것입니다. 과거에는 CPU를 통해서만 그래픽이 렌더링되었습니다. GPU를 사용하면 필터, 비트맵, 비디오 및 텍스트 렌더링이 최적화됩니다. GPU 렌더링이 항상 소프트웨어 렌더링만큼 정확하지는 않습니다. 하드웨어 렌더러를 사용하는 경우 내용이 약간 어색해 보일 수 있습니다. 또한 Flash Player 10.1에는 화면의 Pixel Bender 효과가 렌더링되지 않도록 할 수 있는 제한이 있습니다. 이러한 효과는 하드웨어 가속을 사용하는 경우 검정색 정사각형으로 렌더링될 수 있습니다.

Flash Player 10에는 GPU 가속 기능이 있었지만 GPU가 그래픽을 계산하는 데 사용되지 않았습니다. GPU는 모든 그래픽을 화면에 전송하는 데만 사용되었습니다. Flash Player 10.1에서는 GPU가 또한 그래픽을 계산하는 데 사용되므로 렌더링 속도를 크게 향상시킬 수 있습니다. CPU 작업 로드도 감소하므로 리소스가 제한된 장치(예: 휴대 장치)에서 유용합니다.

가능한 최고 성능을 얻기 위해 휴대 장치에서 내용을 실행할 때 GPU 모드가 자동으로 설정됩니다. GPU 렌더링을 구현하기 위해 더 이상 wmode를 gpu로 설정할 필요는 없지만 wmode를 opaque 또는 transparent로 설정하면 GPU 가속이 비활성화됩니다.

참고: 데스크톱의 Flash Player에서는 소프트웨어 렌더링에 CPU을 계속 사용합니다. 드라이버는 데스크톱에서 매우 다양하며 렌더링 차이를 강조할 수 있으므로 소프트웨어 렌더링이 사용됩니다. 데스크톱과 일부 휴대 장치 간의 렌더링 차이도 있을 수 있습니다.

모바일 AIR 응용 프로그램의 GPU 렌더링

AIR 응용 프로그램 설명자에 <renderMode>gpu</renderMode>를 포함하여 응용 프로그램에 하드웨어 그래픽 가속을 사용하십시오. 런타임에서는 렌더링 모드를 변경할 수 없습니다. 데스크톱 컴퓨터에서 renderMode 설정은 무시되고 GPU 그래픽 가속은 현재 지원되지 않습니다.

GPU 렌더링 모드 제한 사항

AIR 2.5에서 GPU 렌더링 모드를 사용하는 경우 다음과 같은 제한이 있습니다.

- GPU가 객체를 렌더링할 수 없는 경우 아무 것도 표시되지 않으며, CPU 렌더링으로 대체되지도 않습니다.
- 혼합 모드 중 layer, alpha, erase, overlay, hardlight, lighten 및 darken이 지원되지 않습니다.
- 필터가 지원되지 않습니다.
- PixelBender가 지원되지 않습니다.
- 대부분 GPU 장치의 최대 텍스처 크기는 1024x1024입니다. ActionScript에서 이는 모든 변형을 거친 후 표시 객체의 최종 렌더링 최대 크기로 변환됩니다.
- 비디오를 재생하는 AIR 응용 프로그램에서 GPU 렌더링 모드를 사용하지 않는 것이 좋습니다.
- GPU 렌더링 모드에서 가상 키보드가 열릴 때 텍스트 필드가 항상 보이는 위치로 이동하는 것은 아닙니다. 사용자가 텍스트를 입력하는 동안 텍스트 필드가 보이도록 하려면 다음 중 하나를 수행합니다. 텍스트 필드에 포커스가 놓이면 화면의 위쪽 절반에 텍스트 필드를 배치하거나, 화면 위쪽 절반으로 텍스트 필드를 이동하십시오.
- GPU 렌더링 모드는 일부 장치에서 안정적으로 작동하지 않으므로 그러한 장치에서 비활성화됩니다. 최신 정보는 AIR 개발자 릴리스 정보를 참조하십시오.

최상의 GPU 렌더링 모드 사용 방법

다음 지침에 따르면 GPU 렌더링 속도를 높일 수 있습니다.

- 스테이지에 표시되는 항목 수를 줄입니다. 항목마다 렌더링하고 주위의 다른 항목과 합성하는 데 시간이 걸립니다. 표시 객체를 더 이상 표시하지 않으려면 visible 속성을 false로 설정합니다. 단순히 스테이지 밖으로 이동하거나 다른 객체 뒤에 숨기거나 alpha 속성을 0으로 설정하는 방법을 쓰지 마십시오. 표시 객체가 더 이상 필요하지 않으면 removeChild()를 사용하여 스테이지에서 제거하십시오.
- 객체를 만들어 삭제하기보다 객체를 재사용하십시오.
- 비트맵 크기를 2^n x 2^m 비트보다 작지만 이 크기에 가깝게 만듭니다. 크기가 반드시 정확한 2의 거듭제곱일 필요는 없지만, 2의 거듭제곱을 초과하지 않으면서 2의 거듭제곱에 가까워야 합니다. 예를 들어 31 x 15 픽셀 이미지는 33 x 17 픽셀 이미지보다 빠르게 렌더링됩니다. 31과 15는 2: 32 및 16보다 조금 작습니다.
- Graphic.beginBitmapFill() 메서드를 호출할 때 가능하면 repeat 매개 변수를 false로 설정합니다.
- 오버드로우하지 않습니다. 배경색을 배경으로 사용합니다. 큰 모양을 서로 겹치지 않습니다. 그려야 하는 모든 픽셀마다 리소스가 소모됩니다.
- 길고 가늘게 튀어나온 부분이 있거나, 가장자리가 서로 교차하거나, 가장자리를 따라 세부 요소가 많은 모양을 사용하지 않습니다. 이러한 모양은 가장자리가 매끄러운 표시 객체보다 렌더링하는 데 오래 걸립니다.
- 표시 객체의 크기를 제한합니다.
- 그래픽이 자주 업데이트되지 않는 객체를 표시하려면 cacheAsBitMap 및 cacheAsBitmapMatrix를 사용합니다.
- ActionScript 드로잉 API(Graphics 클래스)를 사용하여 그래픽을 제작하지 않는 것이 좋습니다. 가급적 제작 시 정적으로 해당 객체를 만듭니다.
- 비트맵 에셋을 가져오기 전에 최종 크기로 조정합니다.

모바일 AIR 2.0.3의 GPU 렌더링 모드

GPU 렌더링은 Packager for iPhone로 만든 모바일 AIR 응용 프로그램에서 좀 더 제한적입니다. GPU는 cacheAsBitmap 속성이 설정된 비트맵, 단색 모양 및 표시 객체에만 효과적입니다. 또한 cacheAsBitmap 및 cacheAsBitmapMatrix가 설정된 객체의 경우 GPU로 회전 또는 크기 조절되는 객체를 효과적으로 렌더링할 수 있습니다. GPU를 다른 표시 객체와 함께 사용하면 렌더링 성능이 일반적으로 낮아집니다.

GPU 렌더링 성능을 최적화하기 위한 팁

GPU 렌더링으로 SWF 내용의 성능을 크게 향상시킬 수 있지만 이는 내용의 디자인에 상당히 좌우됩니다. 기존 소프트웨어 렌더링에서 잘 작동하는 설정이 GPU 렌더링에서는 제대로 작동하지 않는 경우가 있습니다. 다음과 같은 팁을 활용하면 소프트웨어 렌더링의 성능 손실을 방지하면서 GPU 렌더링에서 우수한 성능을 얻을 수 있습니다.

참고: 하드웨어 렌더링을 지원하는 휴대 장치는 대부분 웹에서 SWF 내용을 액세스합니다. 따라서 모든 종류의 화면에서 최상의 성능을 낼 수 있도록 SWF 내용을 제작할 때 항상 다음과 같은 팁을 고려하는 것이 좋습니다.

- HTML 포함 매개 변수에 wmode=transparent 또는 wmode=opaque를 사용하지 마십시오. 이러한 모드를 사용하면 성능이 저하될 수 있습니다. 또한 소프트웨어 및 하드웨어 렌더링 시에 모두 오디오-비디오 동기화에 약간의 손실이 발생할 수 있습니다. 게다가 많은 플랫폼에서 이러한 모드를 사용할 때 GPU 렌더링을 지원하지 않으므로 성능이 크게 저하됩니다.
- 보통 모드와 알파 블렌드 모드만 사용하고, 다른 블렌드 모드, 특히 레이어 블렌드 모드는 사용하지 마십시오. 일부 블렌드 모드는 GPU 렌더링 시 정확하게 재현되지 않습니다.
- GPU에서는 벡터 그래픽을 렌더링할 때 그래픽을 그리기 전에 작은 삼각형으로 이루어진 메쉬로 분할합니다. 이 프로세스를 테셀레이션이라고 합니다. 테셀레이션은 약간의 성능 저하를 일으키며, 모양이 복잡할수록 성능 저하 현상이 심해집니다. 성능에 미치는 영향을 최소화하려면 변형 모양을 사용하지 마십시오. 변형 모양을 사용하면 GPU 렌더링 시 모든 프레임에 대해 테셀레이션이 발생합니다.
- 자체 교차 곡선이나 매우 가느다란 곡선 영역(예: 가느다란 초승달)을 되도록 사용하지 않고 모양의 가장자리에 복잡한 세부 표현을 가급적 적용하지 않아야 합니다. 이러한 모양은 GPU에서 삼각형 메시로 테셀레이션하기가 복잡합니다. 이해를 돕기 위해 500 × 500 정사각형과 100 × 10 초승달의 두 가지 벡터를 예로 들어 생각해 볼 수 있습니다. 큰 사각형은 단지 삼각형 두 개로 분할되기 때문에 GPU가 쉽게 렌더링할 수 있습니다. 반면, 초승달 곡선을 표현하려면 많은 삼각형이 필요합니다. 따라서 이 모양은 픽셀 수가 작은데도 모양을 렌더링하기가 더욱 복잡합니다.
- 크기를 많이 변경하지 마십시오. 이 경우 GPU가 그래픽을 다시 테셀레이션할 수 있습니다.
- 가급적 오버드로우하지 마십시오. 오버드로우는 여러 그래픽 요소를 겹쳐서 서로 흐리게 하는 기법입니다. 소프트웨어 렌더러를 사용할 때는 각 픽셀을 한 번만 그리므로 소프트웨어 렌더링의 경우 같은 픽셀에서 서로 겹치는 그래픽 요소가 얼마나 많든지 관계없이 응용 프로그램에 성능 손실이 발생하지 않습니다. 반면에 하드웨어 렌더러는 다른 요소가 해당 영역을 흐리게 하는지 여부에 관계없이 각 픽셀을 요소마다 하나씩 그립니다. 예를 들어 두 직사각형이 서로 겹치는 경우 소프트웨어는 렌더러는 겹치는 영역을 한 번만 그리는데 반해 하드웨어 렌더러는 겹치는 영역을 두 번 그립니다.

  따라서 소프트웨어 렌더러를 사용하는 데스크톱에서는 오버드로우로 인한 성능 영향이 거의 없지만 GPU 렌더링을 사용하는 장치에서는 겹치는 모양이 많으면 성능이 저하될 수 있습니다. 그러므로 객체를 가리는 방법을 사용하기보다는 아예 표시하지 않는 편이 좋습니다.
- 채워진 큰 직사각형을 배경으로 사용하지 마십시오. 대신 스테이지의 배경색을 설정합니다.
- 비트맵 반복의 기본 비트맵 채우기 모드를 가급적 사용하지 마십시오. 대신 비트맵 클램프 모드를 사용하면 더 나은 성능을 얻을 수 있습니다.

비동기 작업

가능한 경우 동기 작업보다는 비동기 버전의 작업을 사용하는 것이 좋습니다.

동기 작업은 코드가 지시하는 대로 즉시 실행되고 코드가 다음으로 진행하기 전에 이 작업이 완료될 때까지 기다립니다. 따라서 프레임 루프의 응용 프로그램 코드 단계에서 실행됩니다. 동기 작업의 시간이 너무 오래 걸리면 프레임 루프의 길이가 늘어나서 디스플레이가 멈추는 현상이 발생할 수 있습니다.

코드에서 비동기 작업을 실행할 때는 반드시 즉시 실행할 필요는 없습니다. 응용 프로그램의 코드 및 현재 실행 스레드에 있는 다른 응용 프로그램 코드는 계속해서 실행됩니다. 그런 다음 런타임에서는 렌더링 문제가 발생하지 않도록 하면서 가능한 한 빨리 작업을 수행합니다. 일부 경우에는 백그라운드에서 실행되고 런타임 프레임 루프의 일부로는 전혀 실행되지 않습니다. 마지막으로 작업이 완료되면 런타임에서 이벤트가 전달되고 코드에서는 이 이벤트를 수신하여 추가 작업을 수행할 수 있습니다.

렌더링 문제를 방지하기 위해 비동기 작업을 예약 및 분할합니다. 따라서 응답성이 뛰어난 응용 프로그램에서는 작업을 비동기 방식으로 수행하는 것이 훨씬 쉽습니다. 자세한 내용은 2페이지의 “인지 성능과 실제 성능 비교”를 참조하십시오.

그러나 작업을 비동기적으로 실행하는 경우 오버헤드가 발생합니다. 비동기 작업, 특히 짧은 시간 내에 완료되는 작업의 경우 실제 실행 시간이 더 길어질 수 있습니다.

런타임에서는 많은 작업들이 기본적으로 동기적이거나 비동기적이며 사용자가 이에 대한 실행 방법을 선택할 수 없습니다. 하지만 Adobe AIR에서는 동기 또는 비동기적으로 수행하도록 선택할 수 있는 세 가지 작업 종류가 있습니다.

- File 및 FileStream 클래스 작업

  File 클래스의 많은 작업들은 동기 또는 비동기적으로 수행될 수 있습니다. 예를 들어, 파일 또는 디렉토리를 복사하거나 삭제하고 디렉토리의 내용을 나열하기 위한 메서드는 모두 비동기 버전으로 수행됩니다. 이러한 메서드에는 비동기 버전의 이름에 접미사 "Async"가 추가됩니다. 예를 들어, 파일을 비동기적으로 삭제하려면 File.deleteFile() 메서드 대신 File.deleteFileAsync() 메서드를 호출합니다.

  FileStream 객체를 사용하여 파일에서 읽거나 파일에 쓰는 경우 FileStream 객체를 여는 방식에 따라 작업이 비동기적으로 실행되는지 여부가 결정됩니다. 비동기 작업에 대해서는 FileStream.openAsync() 메서드를 사용합니다. 데이터 쓰기는 비동기적으로 수행됩니다. 데이터 읽기는 청크 단위로 수행되므로 한 번에 한 부분씩 데이터를 사용할 수 있습니다. 반면에 비동기 모드에서는 FileStream 객체가 전체 파일을 읽은 다음에 코드 실행을 계속합니다.

- 로컬 SQL 데이터베이스 작업

  로컬 SQL 데이터베이스를 사용하여 작업할 경우 SQLConnection 객체를 통해 실행되는 모든 작업은 동기 또는 비동기 모드로 실행됩니다. 작업이 비동기적으로 실행되도록 지정하려면 SQLConnection.open() 메서드 대신 SQLConnection.openAsync() 메서드를 사용하여 데이터베이스에 연결하십시오. 데이터베이스 작업이 비동기적으로 실행될 때는 백그라운드에서 실행됩니다. 데이터베이스 엔진은 런타임 프레임 루프에서는 전혀 실행되지 않으므로 데이터베이스 작업으로 인해 렌더링 문제가 발생할 가능성이 매우 낮습니다.

  로컬 SQL 데이터베이스의 성능 향상을 위한 추가 전략은 78페이지의 “SQL 데이터베이스 성능”을 참조하십시오.

- Pixel Bender 독립 실행형 셰이더

  ShaderJob 클래스를 사용하면 Pixel Bender 셰이더를 통해 이미지 또는 데이터 집합을 실행하고 원시 결과 데이터에 액세스할 수 있습니다. 기본적으로 ShaderJob.start() 메서드를 호출하면 셰이더가 비동기적으로 실행됩니다. 실행 작업은 런타임 프레임 루프를 사용하지 않고 백그라운드에서 이루어집니다. ShaderJob 객체를 비동기적으로 강제로 실행하도록 하려면(권장하지 않음) true 값을 start() 메서드의 첫 번째 매개 변수에 전달합니다.

비동기적 코드 실행을 위한 이러한 내장 메커니즘 외에도 동기 대신 비동기적으로 실행되도록 자신의 코드를 직접 구성할 수도 있습니다. 실행 시간이 오래 걸릴 수 있는 작업을 실행하기 위한 코드를 작성할 경우에는 부분적으로 실행되도록 코드를 구성할 수 있습니다. 코드를 여러 부분으로 분할하면 런타임에서 코드 실행 블록 사이에 렌더링 작업을 수행할 수 있으므로 렌더링 문제가 발생할 가능성이 낮아집니다.

코드 분할을 위한 몇 가지 기술이 아래에 나열되어 있습니다. 이러한 기술의 기본적인 개념은 한 번에 작업 중 일부만 수행하도록 코드를 작성한다는 것입니다. 코드가 수행하는 작업과 작업이 중지되는 위치를 추적합니다. 작업이 남아 있는지 여부를 반복적으로 확인하고 작업이 완료될 때까지 청크에 있는 추가 작업을 수행하는 Timer 객체와 같은 메커니즘을 사용합니다.

이러한 방식으로 작업을 분할하도록 코드를 구성하기 위해 설정된 패턴은 얼마 되지 않습니다. 다음 문서 및 코드 라이브러리에서는 이러한 패턴에 대해 설명하고 자신의 응용 프로그램에서 구현해 볼 수 있는 코드를 제공합니다.

- Asynchronous ActionScript Execution(Trevor McCauley의 문서, 몇 가지 구현 예제와 함께 자세한 배경 지식 제공)
- Parsing & Rendering Lots of Data in Flash Player(Jesse Warden의 문서, 배경 세부 사항과 "builder pattern” 및 “green threads”의 두 가지 접근 방식 예제 제공)

- Green Threads(Drew Cummins의 문서, 예제 소스 코드와 함께 “green threads” 기술에 대해 설명)
- greenthreads(Charlie Hubbard의 오픈 소스 코드 라이브러리, ActionScript에서 “green threads” 구현을 위한 정보 제공. 자세한 내용은 greenthreads Quick Start 참조)
- http://www.adobe.com/go/learn_fp_as3_threads_kr("의사 스레딩" 기법 구현 예를 포함한 Alex Harui의 문서)의 ActionScript 3의 스레드

투명 윈도우

AIR 데스크톱 응용 프로그램에서 투명 윈도우 대신에 불투명한 사각형 응용 프로그램 윈도우를 사용해 보십시오.

AIR 데스크톱 응용 프로그램의 초기 윈도우에 대해 불투명한 윈도우를 사용하려면 응용 프로그램 설명자 XML 파일에서 다음 값을 설정합니다.

```
<initialWindow>
    <transparent>false</transparent>
</initialWindow>
```

응용 프로그램 코드로 만들어진 윈도우의 경우 transparent 속성을 false(기본값)로 설정하여 NativeWindowInitOptions 객체를 만듭니다. NativeWindow 객체를 만드는 동안 이를 NativeWindow 생성자에 전달합니다.

```
// NativeWindow: flash.display.NativeWindow class

var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
initOptions.transparent = false;
var win:NativeWindow = new NativeWindow(initOptions);
```

Flex Window 구성 요소의 경우 Window 객체의 open() 메서드를 호출하기 전에 구성 요소의 투명 속성이 false(기본값)로 설정되어 있는지 확인하십시오.

```
// Flex window component: spark.components.Window class

var win:Window = new Window();
win.transparent = false;
win.open();
```

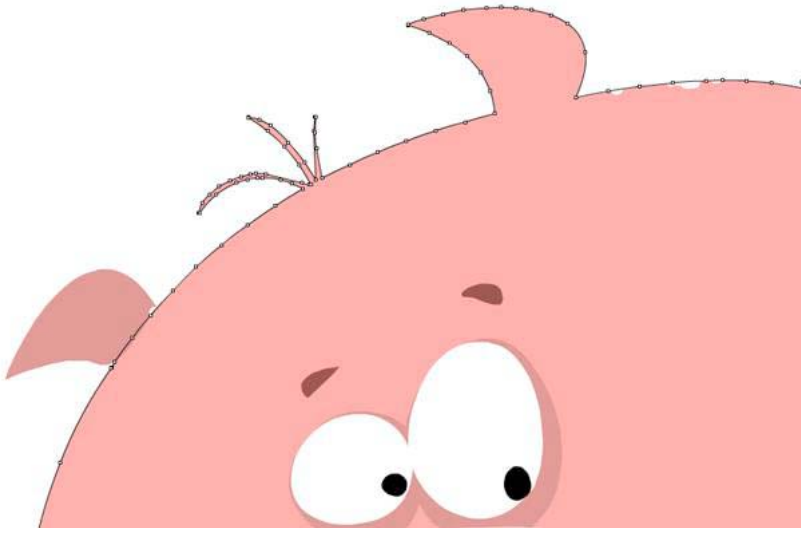
투명 윈도우에서는 응용 프로그램 윈도우 뒤에 사용자의 바탕화면이나 기타 응용 프로그램 윈도우가 표시될 수 있습니다. 따라서 투명 윈도우를 렌더링하려면 런타임에서 더 많은 리소스가 사용됩니다. 직사각형의 비투명 윈도우는 운영 체제의 크롬을 사용하든 사용자 정의 크롬을 사용하든 간에 렌더링 부담이 투명 윈도우와 같지 않습니다.

투명 윈도우는 직사각형이 아닌 디스플레이가 꼭 필요하거나 응용 프로그램 윈도우를 통해 백그라운드 내용을 표시해야 할 경우에만 사용하십시오.

벡터 모양 매끄럽게 하기

모양을 매끄럽게 하여 렌더링 성능을 향상시킬 수 있습니다.

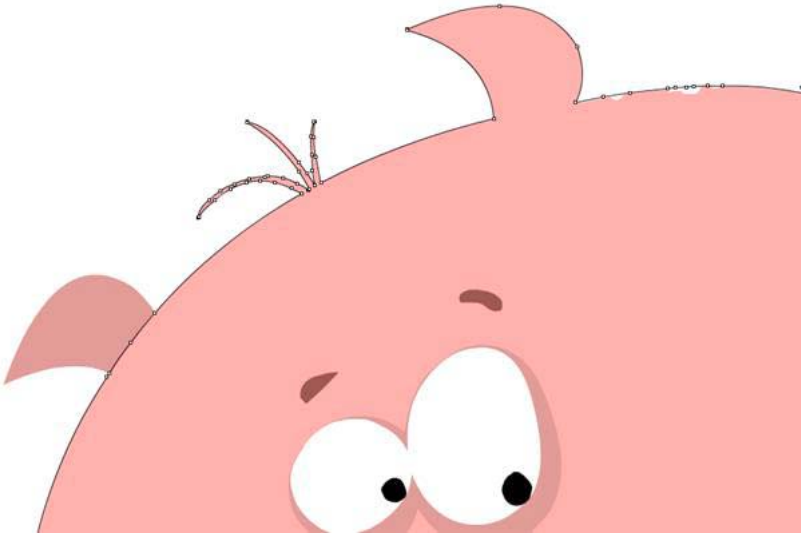
비트맵과 달리, 벡터 내용을 렌더링하려면 많은 계산이 필요합니다. 특히 많은 제어점이 포함된 그래디언트 및 복잡한 경로의 경우 더욱 그러합니다. 디자이너 또는 개발자로서 모양이 충분히 최적화되었는지 확인하십시오. 다음 그림은 많은 제어점이 포함된 단순화되지 않은 경로를 보여 줍니다.



최적화되지 않은 경로

Flash Professional의 매끄럽게 하기 도구를 사용하여 불필요한 제어점을 제거할 수 있습니다. 이에 상응하는 도구를 Adobe® Illustrator®에서 사용할 수 있으며 [문서 정보] 패널에서 전체 점 수와 경로를 볼 수 있습니다.

매끄럽게 하기는 SWF 파일의 최종 크기를 줄이고 렌더링 성능을 향상시켜 불필요한 제어점을 제거합니다. 다음 그림은 매끄럽게 한 후의 동일한 경로를 보여 줍니다.



최적화된 경로

경로를 과도하게 단순화하지 않는 한 이러한 최적화로 시각적으로는 아무 것도 변경되지 않습니다. 그러나 복잡한 경로를 단순화하면 최종 응용 프로그램의 평균 프레임 속도가 크게 향상될 수 있습니다.

6장: 네트워크 상호 작용 최적화

네트워크 상호 작용 향상 기능

Flash Player 10.1 및 AIR 2.5에는 순환 버퍼링 및 스마트 검색을 비롯하여 모든 플랫폼에서 네트워크를 최적화하는 데 사용할 수 있는 새로운 기능 집합이 추가되었습니다.

순환 버퍼링

휴대 장치에서 미디어 내용을 로드하는 경우 데스크톱 컴퓨터에서는 거의 발생하지 않을 문제가 나타날 수 있습니다. 예를 들어 디스크 공간 또는 메모리가 부족해질 가능성이 큽니다. 비디오를 로드할 때 데스크톱 버전의 Flash Player 10.1 및 AIR 2.5는 전체 FLV 파일(또는 MP4 파일)을 하드 드라이브에 다운로드하고 캐시합니다. 그런 다음 해당 캐시 파일에서 비디오를 재생합니다. 따라서 디스크 공간이 부족해지는 경우는 흔치 않습니다. 디스크 공간이 부족한 상황이 발생하면 데스크톱 런타임은 비디오 재생을 중지합니다.

휴대 장치에서는 디스크 공간 부족이 좀 더 쉽게 발생할 수 있습니다. 장치의 디스크 공간이 부족해져도 데스크톱 런타임에서처럼 재생이 중지되지는 않습니다. 대신, 런타임에서 파일의 처음부터 캐시 파일을 다시 작성하여 캐시 파일을 다시 사용하기 시작합니다. 따라서 사용자는 계속 비디오를 볼 수 있습니다. 하지만 파일의 시작 부분을 제외하고 다시 작성된 비디오의 영역은 검색할 수 없습니다. 순환 버퍼링은 기본적으로 시작되지 않습니다. 이러한 순환 버퍼링은 재생 중에 시작할 수 있으며, 동영상이 디스크 공간 또는 RAM보다 클 경우에는 재생이 시작될 때 시작할 수도 있습니다. 순환 버퍼링을 사용할 수 있으려면 런타임에 최소 4MB의 RAM 또는 20MB의 디스크 공간이 필요합니다.

참고: 장치에 충분한 공간이 있으면 모바일 버전의 런타임이 데스크톱 버전과 동일하게 동작합니다. 장치에 디스크가 없거나 디스크가 가득 찼을 경우에는 RAM의 버퍼가 대신 사용됩니다. 캐시 파일 및 RAM 버퍼의 크기 제한은 컴파일 타임에 설정할 수 있습니다. 일부 MP4 파일의 경우 재생을 시작하려면 먼저 전체 파일을 다운로드해야 하는 구조를 갖기도 합니다. 런타임에서는 이러한 파일을 감지하고 디스크 공간이 충분하지 않을 경우 다운로드를 금지하므로 MP4 파일을 재생할 수 없습니다. 이러한 파일의 다운로드는 요청하지 않는 것이 가장 좋습니다.

개발자는 캐시된 스트림의 경계 내에서만 검색이 작동한다는 점에 유의해야 합니다. NetStream.seek()는 오프셋이 범위를 벗어나는 경우 NetStream.Seek.InvalidTime 이벤트가 전달되기 때문에 실패하기도 합니다.

스마트 검색

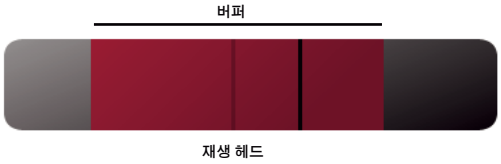
참고: 스마트 검색 기능을 사용하려면 Adobe® Flash® Media Server 3.5.3이 필요합니다.

Flash Player 10.1 및 AIR 2.5에는 스트리밍 비디오를 재생할 때 사용자 환경을 향상시키는 스마트 검색이라는 새 비헤이비어가 도입되었습니다. 사용자가 버퍼 경계 내에서 대상을 검색하는 경우 런타임에서 버퍼를 다시 사용하여 즉시 검색을 제공합니다. 이전 버전의 런타임에서는 버퍼가 다시 사용되지 않습니다. 예를 들어 사용자가 스트리밍 서버에서 비디오를 재생하고 있고 버퍼 시간(NetStream.bufferTime)이 20초로 설정된 경우 사용자가 10초 앞을 검색하면 런타임에서는 이미 로드한 10초를 다시 사용하지 않고 모든 버퍼 데이터를 버렸습니다. 이러한 비헤이비어는 런타임이 서버에서 새 데이터를 더욱 자주 요청하게 하며 이로 인해 저속 연결에서는 재생 속도가 느려집니다.

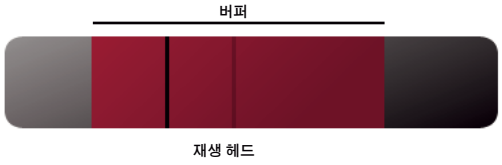
아래 그림에서는 버퍼가 런타임의 이전 릴리스에서 어떻게 작동했는지를 보여 줍니다. bufferTime 속성은 연결이 느려질 경우 비디오를 중지하지 않고 버퍼를 사용할 수 있도록 미리 로드할 초 수를 지정합니다.

스마트 검색 기능 이전의 버퍼 비헤이비어

스마트 검색 기능을 사용하면 사용자가 비디오를 탐색할 때 런타임에서 순간적으로 앞으로 검색하거나 뒤로 검색하는 버퍼를 사용합니다. 다음 그림에서는 새로운 비헤이비어를 보여 줍니다.



스마트 검색 기능을 사용하여 앞으로 검색



스마트 검색 기능을 사용하여 뒤로 검색

스마트 검색에서는 사용자가 앞 또는 뒤로 검색할 때 버퍼를 다시 사용하기 때문에 재생 환경이 더 빨라지고 원활해집니다. 이 새로운 비헤이비어의 이점 중 하나는 비디오 제작자의 대역폭이 절약된다는 것입니다. 하지만 버퍼 제한을 벗어나서 검색하는 경우 표준 비헤이비어가 발생하고 런타임은 서버에서 새 데이터를 요청합니다.

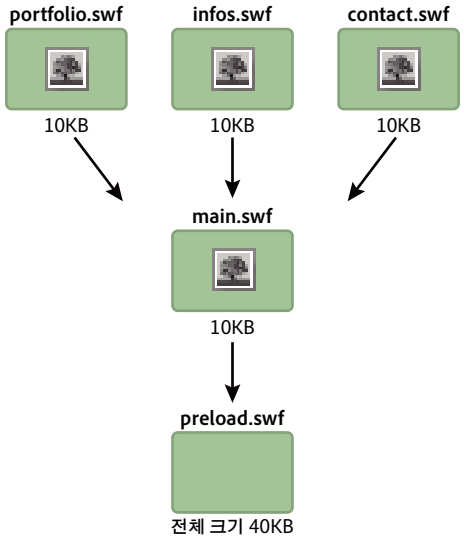
참고: 이 비헤이비어는 점진적 비디오 다운로드에는 적용되지 않습니다.

스마트 검색을 사용하려면 NetStream.inBufferSeek을 true로 설정합니다.

외부 내용

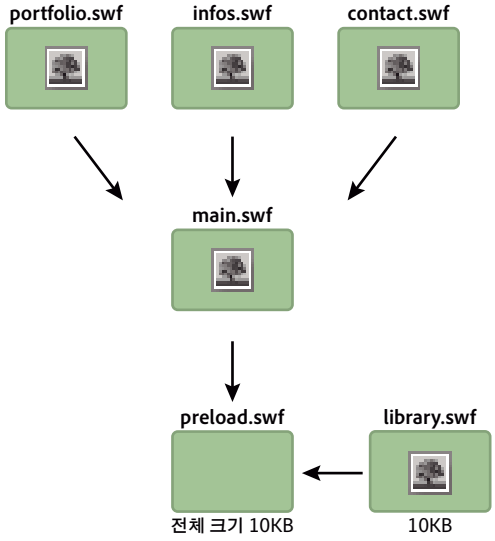
응용 프로그램을 여러 개의 SWF 파일로 분할합니다.

휴대 장치는 네트워크 액세스가 제한될 수 있습니다. 따라서 내용을 빠르게 로드하려면 응용 프로그램을 여러 SWF 파일로 분할하고, 전체 응용 프로그램에서 코드 논리와 에셋을 다시 사용하는 것이 좋습니다. 예를 들어 다음 다이어그램에서처럼 여러 SWF 파일로 분할된 응용 프로그램을 고려해 봅니다.



여러 SWF 파일로 분할된 응용 프로그램

이 예제에서 각 SWF 파일에는 동일한 비트맵의 고유한 복사본이 포함됩니다. 다음 다이어그램에 나타난 것처럼 런타임 공유 라이브러리를 사용하여 이러한 복제를 방지할 수 있습니다.



런타임 공유 라이브러리 사용

이 기술을 사용하면 런타임 공유 라이브러리가 로드되어 비트맵을 다른 SWF 파일에서 사용할 수 있게 됩니다. ApplicationDomain 클래스는 로드된 모든 클래스 정의를 저장하고 런타임에 getDefinition() 메서드를 통해 이러한 정의를 사용할 수 있도록 합니다.

런타임 공유 라이브러리는 또한 모든 코드 논리를 포함할 수 있습니다. 따라서 전체 응용 프로그램을 다시 컴파일하지 않고 런타임에 업데이트할 수 있습니다. 다음 코드에서는 런타임 공유 라이브러리를 로드하고 SWF 파일에 포함된 정의를 런타임에 추출합니다. 이 기술은 글꼴, 비트맵, 사운드 또는 ActionScript 클래스에 사용할 수 있습니다.

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load(new URLRequest("library.swf") );
var classDefinition:String = "Logo";

function loadingComplete(e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the loaded SWF file application domain
    var appDomain:ApplicationDomain = objectLoaderInfo.applicationDomain;

    // Check whether the definition is available
    if ( appDomain.hasDefinition(classDefinition) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate logo
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " +  classDefinition + " is not available.");
}
```

로드하는 SWF 파일의 응용 프로그램 도메인에서 클래스 정의를 로드하면 정의를 보다 쉽게 가져올 수 있습니다.

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load ( new URLRequest ("rsl.swf"), new LoaderContext ( false, ApplicationDomain.currentDomain) );
var classDefinition:String = "Logo";

function loadingComplete ( e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the current SWF file application domain
    var appDomain:ApplicationDomain = ApplicationDomain.currentDomain;

    // Check whether the definition is available
    if (appDomain.hasDefinition( classDefinition ) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate it
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " +  classDefinition + " is not available.");
}
```

이제 SWF 파일에서 사용 가능한 클래스를 현재 응용 프로그램 도메인에서 getDefinition() 메서드를 호출하여 사용할 수 있습니다. 또한 getDefinitionByName() 메서드를 호출하여 클래스에 액세스할 수도 있습니다. 이 기술은 글꼴 및 대용량 에셋을 한 번만 로드하고 다른 SWF 파일로 에셋을 내보내지 않는 방식으로 대역폭을 절약합니다. 유일한 제한은 loader.swf 파일을 통해 응용 프로그램을 테스트하고 실행해야 한다는 것입니다. 이 파일은 먼저 에셋을 로드한 다음 응용 프로그램을 구성하는 다른 SWF 파일을 로드합니다.

입출력 오류


IO 오류에 대한 이벤트 핸들러 및 오류 메시지를 제공합니다.

휴대 장치에서는 고속 인터넷에 연결된 데스크톱 컴퓨터에서보다 네트워크가 불안정할 수 있습니다. 휴대 장치에서 외부 내용에 액세스하는 데는 가용성과 속도라는 두 가지 제약이 따릅니다. 따라서 에셋의 리소스 소모를 적게 하고 사용자에게 피드백을 제공하도록 모든 IO_ERROR 이벤트에 대한 핸들러를 추가해야 합니다.

예를 들어 어떤 사용자가 휴대 장치에서 웹 사이트를 탐색하는 동안 두 광역 기지국 간에 네트워크 연결이 갑자기 끊겨 연결이 손실되었고 이때 동적 에셋이 로드 중이었다고 가정해 봅니다. 데스크톱의 경우 이러한 시나리오가 거의 일어나지 않기 때문에 빈 이벤트 리스너를 사용하여 런타임 오류가 발생하지 않도록 할 수 있습니다. 하지만 휴대 장치의 경우 단순히 빈 리스너 이상의 방법을 사용하여 상황을 처리해야 합니다.

다음 코드는 IO 오류에 응답하지 않습니다. 아래 코드를 실제로 사용하지는 마십시오.

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener( Event.COMPLETE, onComplete );
addChild( loader );
loader.load( new URLRequest ("asset.swf" ) );

function onComplete( e:Event ):void
{
    var loader:Loader = e.currentTarget.loader;
    loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
    loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}
```

더 나은 방법은 이러한 오류를 처리하고 사용자에게 오류 메시지를 제공하는 것입니다. 다음 코드는 오류를 적절히 처리합니다.

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );
loader.contentLoaderInfo.addEventListener ( IOErrorEvent.IO_ERROR, onIOError );
addChild ( loader );
loader.load ( new URLRequest ("asset.swf" ) );

function onComplete ( e:Event ):void
{
      var loader:Loader = e.currentTarget.loader;
      loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
      loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}

function onIOError ( e:IOErrorEvent ):void
{
      // Show a message explaining the situation and try to reload the asset.
      // If it fails again, ask the user to retry when the connection will be restored
}
```

최상의 방법은 사용자가 내용을 다시 로드하는 방법을 제공하는 것입니다. 이 비헤이비어는 onIOError() 핸들러에서 구현할 수 있습니다.

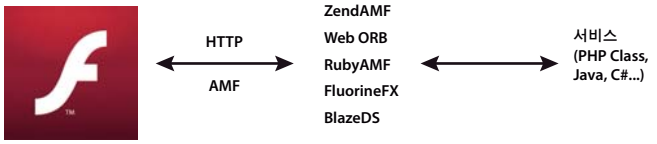
Flash Remoting

최적화된 클라이언트-서버 데이터 통신을 위해 Flash Remoting 및 AMF를 사용합니다.

XML을 사용하여 원격 내용을 SWF 파일로 로드할 수 있습니다. 하지만 XML은 런타임에서 로드하고 파싱하는 일반 텍스트이므로 제한된 크기의 내용을 로드하는 응용 프로그램에 가장 적합합니다. 대용량 내용을 로드하는 응용 프로그램을 개발하는 경우에는 Flash Remoting 기술과 AMF(Action Message Format)를 사용하는 것이 좋습니다.

AMF는 서버와 런타임 간에 데이터를 공유하는 데 사용되는 이진 형식입니다. AMF를 사용하면 데이터의 크기가 줄어들어 전송 속도가 향상됩니다. AMF는 런타임의 기본 형식이므로 런타임에 AMF 데이터를 전송하면 메모리를 많이 사용하는 클라이언트 측 직렬화 및 직렬화 해제를 방지할 수 있습니다. 원격 게이트웨이는 다음과 같은 작업을 처리합니다. ActionScript 데이터 유형을 서버에 보낼 때 원격 게이트웨이는 서버 쪽에서 직렬화를 처리합니다. 게이트웨이는 또한 해당 데이터 유형도 보냅니다. 이러한 데이터 유형은 런타임에서 호출할 수 있는 메서드 집합을 노출하는 서버에서 만들어진 클래스입니다. Flash Remoting 게이트웨이에는 ZendAMF, FluorineFX, WebORB 및 Adobe 공식 오픈 소스 Java Flash Remoting 게이트웨이인 BlazeDS 등이 포함됩니다.

다음 그림에서는 Flash Remoting의 개념을 보여 줍니다.



Flash Remoting

다음 예제에서는 NetConnection 클래스를 사용하여 Flash Remoting 게이트웨이에 연결합니다.

```
// Create the NetConnection object
var connection:NetConnection = new NetConnection ();

// Connect to a Flash Remoting gateway
connection.connect ("http://www.yourserver.com/remotingservice/gateway.php");

// Asynchronous handlers for incoming data and errors
function success ( incomingData:* ):void
{
    trace( incomingData );
}

function error ( error:* ):void
{
    trace( "Error occured" );
}

// Create an object that handles the mapping to success and error handlers
var serverResult:Responder = new Responder (success, error);

// Call the remote method
connection.call ("org.yourserver.HelloWorld.sayHello", serverResult, "Hello there ?");
```

원격 게이트웨이 연결은 간단합니다. 하지만 Adobe® Flex® SDK에 포함된 RemoteObject 클래스를 통해 Flash Remoting을 사용하는 것이 훨씬 간단할 수 있습니다.

참고: Flex 프레임워크의 파일 같은 외부 SWC 파일을 Adobe® Flash® Professional 프로젝트 내부에서 사용할 수 있습니다. SWC 파일을 사용하면 Flex SDK의 나머지 기능을 사용하지 않고도 RemoteObject 클래스와 이 클래스의 종속성을 사용할 수 있습니다. 고급 개발자는 필요한 경우 원시 Socket 클래스를 통해 직접 원격 게이트웨이와 통신할 수도 있습니다.

불필요한 네트워크 작업


에셋은 필요할 때마다 네트워크에서 로드하는 대신에 로드한 후 로컬로 캐시하십시오.

응용 프로그램에서 미디어 또는 데이터와 같은 에셋이 로드되는 경우 이를 로컬 장치에 저장하여 에셋을 캐시하십시오. 자주 변경되지 않는 에셋의 경우 캐시를 일정 간격에 따라 업데이트하십시오. 예를 들어, 응용 프로그램에서 이미지 파일의 새 버전을 하루에 한 번 확인하거나 두 시간 마다 데이터를 새로 고치도록 할 수 있습니다.

에셋 유형과 성질에 따라 여러 가지 방법으로 에셋을 캐시할 수 있습니다.

- 이미지 및 비디오와 같은 미디어 에셋: File 및 FileStream 클래스를 사용하여 파일을 파일 시스템에 저장하십시오.
- 개별 데이터 값 또는 소량의 데이터 집합: SharedObject 클래스를 사용하여 값을 로컬 공유 객체로 저장하십시오.
- 대량의 데이터 집합: 로컬 데이터베이스에 데이터를 저장하거나 데이터를 직렬화하고 파일에 저장하십시오.

데이터 값 캐싱을 위해 오픈 소스 AS3CoreLib 프로젝트에는 로드 및 캐싱 작업을 수행하는 ResourceCache 클래스가 포함됩니다.

7장: 미디어를 사용한 작업

비디오

휴대 장치의 비디오 성능을 최적화하는 방법에 대한 자세한 내용은 Adobe Developer Connection 웹 사이트의 휴대 장치용으로 웹 콘텐트 최적화를 참조하십시오.

특히 다음 섹션을 참조하십시오.

- 휴대 장치에서 비디오 재생
- 코드 샘플

다음 섹션에는 휴대 장치용 비디오 플레이어 개발과 관련된 정보가 들어 있습니다.

- 비디오 인코딩 지침
- 유용한 방법
- 비디오 플레이어의 성능을 프로파일링하는 방법
- 참조 비디오 플레이어 구현

StageVideo

StageVideo 클래스를 사용하면 하드웨어 가속을 사용하여 비디오를 나타낼 수 있습니다.

StageVideo 객체를 사용하는 방법에 대한 자세한 내용은 ActionScript 3.0 개발자 안내서에서 하드웨어 가속 프리젠테이션에 StageVideo 클래스 사용을 참조하십시오.

오디오

Flash Player 9.0.115.0 및 AIR 1.0부터 런타임에서 AAC 파일(AAC 기본, AAC LC 및 SBR)을 재생할 수 있습니다. mp3 파일 대신 AAC 파일을 사용하여 간단한 최적화를 실현할 수 있습니다. AAC 형식은 동등한 비트율에서 mp3 형식보다 품질은 높고 파일 크기는 작습니다. 파일 크기를 줄이면 대역폭이 절약되는데, 이는 고속 인터넷 연결을 제공하지 않는 휴대 장치에서 중요한 요소입니다.

하드웨어 오디오 디코딩

비디오 디코딩과 마찬가지로 오디오 디코딩의 경우 높은 CPU 주기가 필요하며 장치에서 사용 가능한 하드웨어를 활용하여 최적화할 수 있습니다. Flash Player 10.1 및 AIR 2.5에서는 AAC 파일(LC, HE/SBR 프로파일) 또는 mp3 파일(PCM은 지원되지 않음)을 디코딩할 때 하드웨어 오디오 드라이버를 감지 및 사용하여 성능을 향상시킬 수 있습니다. CPU 사용이 크게 감소하므로 배터리 사용이 줄어들고 다른 작업에 CPU를 사용할 수 있습니다.

참고: AAC 형식을 사용하는 경우 대부분의 장치에서 하드웨어 지원이 부족하므로 AAC Main 프로파일은 장치에서 지원되지 않습니다.

하드웨어 오디오 디코딩 과정은 사용자 및 개발자가 인지할 수 없는 상태로 수행됩니다. 런타임에서 오디오 스트림 재생을 시작하면 비디오의 경우와 마찬가지로 먼저 하드웨어를 확인합니다. 하드웨어 드라이버가 사용 가능하고 오디오 형식이 지원되는 경우 하드웨어 오디오 디코딩이 수행됩니다. 그러나 하드웨어를 통해 들어오는 AAC 또는 mp3 스트림 디코딩을 처리할 수 있더라도 경우에 따라 하드웨어에서 일부 효과를 처리하지 못할 수도 있습니다. 예를 들어 하드웨어에서 하드웨어 제한에 따라 오디오 믹싱 및 다시 샘플링을 처리하지 않는 경우가 있습니다.

8장: SQL 데이터베이스 성능

데이터베이스 성능을 위한 응용 프로그램 디자인

실행 후 SQLStatement 객체의 text 속성을 변경하지 마십시오. 대신, 각 SQL 문에 대해 하나의 SQLStatement 인스턴스를 사용하고 명령문 매개 변수를 사용하여 다른 값을 제공하십시오.

SQL 문이 실행되기 전에 런타임은 SQL 문을 준비(컴파일)하여 명령문을 수행하기 전에 내부적으로 수행되는 단계를 확인합니다. 이전에 실행되지 않은 SQLStatement 인스턴스에서 SQLStatement.execute()를 호출하면 명령문이 실행되기 전에 자동으로 준비됩니다. execute() 메서드를 이후에 호출하는 경우 SQLStatement.text 속성이 변경되지 않았으면 명령문이 여전히 준비된 상태입니다. 따라서 명령문이 더 빨리 실행됩니다.

문 실행마다 값이 변경되는 경우 명령문을 다시 사용하는 이점을 극대화하도록 명령문 매개 변수를 사용하여 명령문을 사용자 정의합니다. 문 매개 변수는 SQLStatement.parameters 연관 배열 속성을 사용하여 지정됩니다. SQLStatement 인스턴스의 text 속성을 변경하는 경우와 달리, 명령문 매개 변수의 값을 변경하면 런타임에서 명령문을 다시 준비할 필요가 없습니다.

SQLStatement 인스턴스를 다시 사용하는 경우 응용 프로그램은 SQLStatement 인스턴스가 준비되면 이 인스턴스에 대한 참조를 저장해야 합니다. 이 인스턴스에 대한 참조를 유지하려면 함수 범위 변수가 아니라 클래스 범위 변수로 변수를 선언합니다. SQLStatement를 클래스 범위 변수로 만들기 위한 한 가지 좋은 방법은 한 클래스에 SQL 문을 래핑하는 응용 프로그램을 구성하는 것입니다. 함께 실행되는 명령문 그룹도 한 클래스에 래핑할 수 있습니다. 이 기술은 명령 디자인 패턴을 사용하는 것으로 알려져 있습니다. 인스턴스를 클래스의 멤버 변수로 정의하면 래퍼 클래스의 인스턴스가 응용 프로그램에 있는 한 이러한 SQLStatement 인스턴스가 유지됩니다. 최소한 함수 밖에 SQLStatement 인스턴스가 포함된 변수를 정의하기만 해도 해당 인스턴스가 메모리에 유지됩니다. 예를 들어, SQLStatement 인스턴스를 ActionScript 클래스에서 멤버 변수로 선언하거나 JavaScript 파일에서 비함수 변수로 선언합니다. 그런 다음 쿼리를 정말로 실행하려고 할 때 명령문의 매개 변수 값을 설정하고 execute() 메서드를 호출할 수 있습니다.

데이터 비교 및 정렬을 위한 실행 속도를 향상시키려면 데이터베이스 인덱스를 사용하십시오..

한 열에 대한 인덱스를 만들면 데이터베이스에 해당 열 데이터에 대한 복사본이 저장됩니다. 이 복사본은 숫자 또는 알파벳순으로 정렬됩니다. 따라서 데이터베이스에서 항등 연산자를 사용할 때와 같이 신속하게 값을 일치시키고 ORDER BY 절을 사용하여 결과 데이터를 정렬할 수 있습니다.

데이터베이스 인덱스는 지속적으로 최신 상태로 유지되므로 해당 테이블에서 데이터 변경 작업(INSERT 또는 UPDATE)을 발생시켜 속도가 약간 느려집니다. 그러나 데이터 검색 속도가 크게 향상됩니다. 이러한 성능상의 장단점으로 인해 단순히 모든 테이블의 모든 열을 인덱싱하는 것은 피해야 합니다. 대신에 자신만의 인덱스 정의 전략을 사용하십시오. 인덱싱 전략을 계획하려면 다음 지침을 참조하십시오.

- 연결 테이블, WHERE 절 또는 ORDER BY 절에 사용되는 열은 인덱스를 설정하십시오.
- 열이 자주 함께 사용될 경우 단일 인덱스를 사용하여 하나로 지정하십시오.
- 검색하는 텍스트 데이터가 포함된 열이 알파벳순으로 정렬된 경우 인덱스에 대해 COLLATE NOCASE 상관 관계를 지정하십시오.

응용 프로그램 유휴 시간 동안 SQL 문을 미리 컴파일하는 것이 좋습니다..

SQL 문을 처음 실행하면 데이터베이스 엔진에서 SQL 텍스트를 준비(컴파일)하기 때문에 속도가 더 느립니다. 명령문의 준비 및 실행에는 많은 리소스가 필요할 수 있으므로 한 가지 전략은 초기 데이터를 미리 로드한 다음 다른 명령문을 백그라운드에서 실행하는 것입니다.

1. 응용 프로그램에서 처음 필요한 데이터를 로드합니다.
2. 응용 프로그램의 초기 시작 작업이 완료되었을 때나 응용 프로그램의 다른 "유휴" 시간에 다른 명령문을 실행합니다.

예를 들어, 응용 프로그램이 초기 화면을 표시하기 위해 데이터베이스에 전혀 액세스할 필요가 없다고 가정해 보십시오. 이 경우에는 데이터베이스 연결을 열기 전에 화면이 표시될 때까지 기다리십시오. 마지막으로 SQLStatement 인스턴스를 만들고 실행 가능한 인스턴스를 모두 실행합니다.

또는 응용 프로그램이 시작될 때 특정 쿼리의 결과와 같은 데이터를 즉시 표시한다고 가정해 보십시오. 이 경우에는 해당 쿼리에 대한 SQLStatement 인스턴스를 실행합니다. 초기 데이터가 로드되고 표시된 후 다른 데이터베이스 작업에 대한 SQLStatement 인스턴스를 만들고 가능하면 나중에 필요한 다른 명령문을 실행합니다.

실제로는 SQLStatement 인스턴스를 재사용할 경우 이 명령문을 준비하는데 걸리는 추가 시간은 단지 한 번만 필요합니다. 이러한 추가 시간은 전체 성능에 큰 영향을 주지 않을 것입니다.

여러 SQL 데이터 변경 작업은 하나의 트랜잭션으로 그룹화하십시오..

데이터 추가나 변경이 포함된 많은 SQL 문(INSERT 또는 UPDATE 문)을 실행하는 경우를 가정해 보십시오. 명시적 트랜잭션에서 모든 명령문을 실행하여 성능을 크게 향상시킬 수 있습니다. 트랜잭션을 명시적으로 시작하지 않으면 각 명령문이 자동으로 만들어진 자체 트랜잭션에서 실행됩니다. 각 트랜잭션(각 명령문)의 실행이 완료된 후 런타임은 결과 데이터를 디스크의 데이터베이스 파일에 씁니다.

반면에 트랜잭션을 명시적으로 만들고 해당 트랜잭션의 컨텍스트에서 명령문을 실행하는 경우 런타임은 메모리에서 모든 변경을 수행하고 트랜잭션이 커밋될 때 한 번에 모든 변경 사항을 데이터베이스 파일에 씁니다. 데이터를 디스크에 쓰는 것은 대개 작업에서 가장 시간이 오래 걸리는 부분입니다. 따라서 SQL 문마다 한 번이 아니라 전체적으로 한 번만 디스크에 쓰면 성능이 크게 향상될 수 있습니다.

큰 SELECT 쿼리 결과는 SQLStatement 클래스의 execute() 메서드(prefetch 매개 변수 사용) 및 next() 메서드를 사용하여 여러 부분으로 처리하십시오..

큰 결과 집합을 검색하는 SQL 문을 실행한다고 가정해 보십시오. 그런 다음 응용 프로그램이 각 데이터 행을 루프로 처리합니다. 예를 들어, 데이터의 형식을 지정하고 이로부터 객체를 만듭니다. 이러한 데이터 처리에는 시간이 많이 소요되어 화면이 멈추거나 응답하지 않는 등의 렌더링 문제가 발생할 수 있습니다. 65페이지의 "비동기 작업"에서 설명한 것처럼 한 가지 해결 방법은 작업을 여러 청크로 분할하는 것입니다. SQL 데이터베이스 API를 사용하면 데이터 처리를 쉽게 분할할 수 있습니다.

SQLStatement 클래스의 execute() 메서드에는 선택적인 prefetch 매개 변수(첫 번째 매개 변수)가 포함됩니다. 값을 제공하면 실행이 완료될 때 데이터베이스가 반환하는 최대 결과 행 수를 지정합니다.

```
dbStatement.addEventListener(SQLEvent.RESULT, resultHandler);
dbStatement.execute(100); // 100 rows maximum returned in the first set
```

결과 데이터의 첫 번째 집합이 반환되면 next() 메서드를 호출하여 명령문 실행을 계속하고 다른 결과 행 집합을 검색할 수 있습니다. execute() 메서드와 비슷하게 next() 메서드는 prefetch 매개 변수를 수락하여 반환할 최대 행 수를 지정합니다.

```
// This method is called when the execute() or next() method completes
function resultHandler(event:SQLEvent):void
{
    var result:SQLResult = dbStatement.getResult();
    if (result != null)
    {
        var numRows:int = result.data.length;
        for (var i:int = 0; i < numRows; i++)
        {
            // Process the result data
        }

        if (!result.complete)
        {
            dbStatement.next(100);
        }
    }
}
```

모든 데이터가 로드될 때까지 next() 메서드를 계속 호출할 수 있습니다. 이전 목록에서와 같이 데이터가 모두 로드되었는지를 확인할 수 있습니다. execute() 또는 next() 메서드가 완료될 때마다 생성되는 SQLResult 객체의 complete 속성을 확인합니다.

참고: prefetch 매개 변수와 next() 메서드를 사용하여 결과 데이터 처리를 분할합니다. 쿼리의 결과를 결과 집합의 부분으로 제한하려는 경우에는 이 매개 변수와 메서드를 사용하지 마십시오. 명령문의 결과 집합에서 일부 행만 검색하려는 경우에는 SELECT 문의 LIMIT 절을 사용합니다. 결과 집합이 큰 경우에도 prefetch 매개 변수 및 next() 메서드를 사용하여 결과 처리를 분할할 수 있습니다.


단일 데이터베이스에서 여러 비동기 SQLConnection 객체를 사용하여 여러 명령문을 동시에 실행한다고 가정해 보십시오..

openAsync() 메서드를 사용하여 SQLConnection 객체가 데이터베이스에 연결되면 기본 런타임 실행 스레드가 아닌 백그라운드에서 실행됩니다. 또한 각 SQLConnection은 고유한 백그라운드 스레드로 실행됩니다. 여러 SQLConnection 객체를 사용하면 여러 SQL 문을 동시에 효과적으로 실행할 수 있습니다.

하지만 이 방법도 잠재적인 단점을 갖고 있습니다. 가장 중요한 문제는 각각의 추가 SQLStatement 객체에 추가 메모리가 필요하다는 것입니다. 또한 동시 실행 작업으로 인해 프로세스의 부담이 증가하고 CPU 또는 CPU 코어가 하나 뿐인 시스템의 경우 특히 문제가 될 수 있습니다. 이러한 단점이 있기 때문에 이 방법은 휴대 장치의 경우 권장되는 방법이 아닙니다.

또 다른 문제는 SQLStatement 객체가 단일 SQLConnection 객체에 연결될 경우 SQLStatement 객체를 재사용하여 얻게 되는 잠재적인 이점이 사라질 수 있다는 것입니다. 따라서 연관된 SQLConnection 객체가 이미 사용 중이면 SQLStatement 객체를 재사용할 수 없습니다.

단일 데이터베이스에 연결된 여러 SQLConnection 객체를 사용하도록 선택한 경우에는 각 객체가 고유한 트랜잭션에서 자신의 명령문을 실행한다는 것을 염두에 둬야 합니다. 데이터 추가, 수정, 삭제 등 데이터를 변경하는 모든 코드에서 이러한 별개의 트랜잭션을 처리해야 합니다.

Paul Robertson이 만든 오픈 소스 코드 라이브러리를 이용하면 잠재적인 단점을 최소화하면서 여러 SQLConnection 객체 사용에 따른 이점을 효과적으로 활용할 수 있습니다. 이 라이브러리에서는 SQLConnection 객체 풀을 사용하고 연관된 SQLStatement 객체를 관리합니다. 이 방식으로 SQLStatement 객체가 재사용되도록 보장하고 여러 SQLConnection 객체를 사용하여 여러 명령문을 동시에 실행할 수 있도록 보장합니다. 자세한 내용을 확인하고 라이브러리를 다운로드하려면 http://probertson.com/projects/air-sqlite/를 방문하십시오.

데이터베이스 파일 최적화


데이터베이스 스키마 변경을 방지하십시오..

가능하면 데이터베이스의 테이블에 데이터를 추가한 후 데이터베이스의 스키마(테이블 구조)를 변경하지 마십시오. 일반적으로 데이터베이스 파일은 파일 시작 부분에서 테이블 정의로 구성되어 있습니다. 데이터베이스에 대한 연결을 열면 런타임은 해당 정의를 로드합니다. 데이터를 데이터베이스 테이블에 추가하면 해당 데이터가 파일에서 테이블 정의 데이터 뒤에 추가됩니다. 그러나 스키마를 변경하는 경우 새 테이블 정의 데이터가 데이터베이스 파일의 테이블 데이터와 혼합됩니다. 예를 들어 테이블에 열을 추가하거나 새 테이블을 추가하면 데이터 유형이 혼합됩니다. 테이블 정의 데이터가 모두 데이터베이스 파일 시작 부분에 있지 않으면 데이터베이스에 대한 연결을 여는 데 더 오래 걸립니다. 연결이 더 느리게 열리는 이유는 런타임에서 파일의 여러 부분에 있는 테이블 정의 데이터를 읽는 시간이 더 오래 걸리기 때문입니다.


스키마 변경 후 데이터베이스를 최적화하려면 SQLConnection.compact() 메서드를 사용합니다.

스키마를 변경해야 하는 경우에는 변경을 완료한 후 SQLConnection.compact() 메서드를 호출할 수 있습니다. 이 작업을 수행하면 테이블 정의 데이터가 파일의 시작 부분에 함께 있도록 데이터베이스 파일이 다시 구성됩니다. 그러나 compact() 작업은 특히 데이터베이스 파일이 커질수록 시간이 많이 걸릴 수 있습니다.

불필요한 데이터베이스 런타임 처리

SQL 문에서는 전체 테이블 이름(데이터베이스 이름 포함)을 사용하십시오..

항상 명령문에 각 테이블 이름과 함께 데이터베이스 이름을 명시적으로 지정하십시오. 기본 데이터베이스인 경우에는 “main”을 사용합니다. 예를 들어, 다음 코드에는 명시적인 데이터베이스 이름인 main이 포함됩니다.

```
SELECT employeeId
FROM main.employees
```

데이터베이스 이름을 명시적으로 지정하면 런타임에서 일치하는 테이블을 찾기 위해 연결된 각 데이터베이스를 확인할 필요가 없습니다. 또한 런타임에서 잘못된 데이터베이스를 선택할 가능성도 방지됩니다. SQLConnection이 단일 데이터베이스에만 연결되는 경우라도 이 규칙을 따르십시오. 내부적으로 SQLConnection은 SQL 문을 통해 액세스할 수 있는 임시 데이터베이스에도 연결됩니다.

SQL INSERT 및 SELECT 문에 열 이름을 명시적으로 사용하십시오..

다음 예에서는 열 이름을 명시적으로 사용하는 방법을 보여 줍니다.

```
INSERT INTO main.employees (firstName, lastName, salary)
VALUES ("Bob", "Jones", 2000)

SELECT employeeId, lastName, firstName, salary
FROM main.employees
```

이전 예와 다음 예를 비교해서 보십시오. 피해야 하는 코드 스타일:

```
-- bad because column names aren't specified
INSERT INTO main.employees
VALUES ("Bob", "Jones", 2000)

-- bad because it uses a wildcard
SELECT *
FROM main.employees
```

명시적인 열 이름이 없으면 런타임에서 열 이름을 확인하기 위해 추가적인 작업을 수행해야 합니다. SELECT 문에서 명시적 열이 아니라 와일드카드를 사용하는 경우 런타임에서 추가 데이터를 검색합니다. 이러한 추가 데이터로 인해 처리 작업이 추가로 발생하고 필요하지 않은 객체 인스턴스가 추가로 생성됩니다.

테이블을 자체 비교하지 않는 한 명령문 하나에 동일한 테이블을 여러 번 연결하지 마십시오..

SQL 문이 커지면 의도하지 않게 데이터베이스 테이블을 쿼리에 여러 번 연결할 수 있습니다. 테이블을 한 번만 사용하여 동일한 결과를 얻을 수 있는 경우가 많습니다. 쿼리에서 하나 이상의 보기를 사용하는 경우 동일한 테이블을 여러 번 연결할 가능성이 높습니다. 예를 들어 테이블을 쿼리에 연결하고 해당 테이블의 데이터를 포함하는 보기도 연결할 수 있습니다. 두 가지 작업으로 둘 이상의 연결이 발생합니다.

효율적인 SQL 구문

쿼리에 테이블을 포함하려면 WHERE 절에 하위 쿼리를 사용하는 대신 JOIN(FROM 절에서)을 사용하십시오. 이 팁은 결과 집합이 아닌 필터링을 위해서만 테이블 데이터가 필요한 경우에도 적용됩니다.

FROM 절에서 여러 테이블을 연결하면 WHERE 절에서 하위 쿼리를 사용할 때보다 성능이 향상됩니다.

인덱스를 활용할 수 없는 SQL 문은 사용하지 마십시오. 이러한 명령문에는 하위 쿼리에서 집계 함수를 사용하거나, 하위 쿼리에서 UNION 문을 사용하거나, UNION 문과 함께 ORDER BY 절을 사용하는 경우가 포함됩니다.

인덱스를 사용하면 SELECT 쿼리를 처리하는 속도를 크게 향상시킬 수 있습니다. 하지만 일부 SQL 구문에서는 데이터베이스에서 인덱스 사용을 방해하여 검색 또는 정렬 작업에 실제 데이터를 강제로 사용하도록 만듭니다.

LIKE('%XXXX%')와 같이 특히 선행 와일드카드 문자가 있는 경우에는 LIKE 같은 연산자를 사용하지 않는 것이 좋습니다..

LIKE 연산은 와일드카드 검색 사용을 지원하기 때문에 정확한 일치 비교를 사용하는 것보다 속도가 느립니다. 특히 와일드카드 문자로 검색 문자열을 시작하는 경우 데이터베이스에서 검색에 인덱스를 전혀 사용할 수 없습니다. 대신 데이터베이스에서는 테이블에 있는 각 행의 전체 텍스트를 검색해야 합니다.

IN 연산자는 사용하지 않는 것이 좋습니다. 가능한 값이 미리 알려진 경우에는 실행 속도를 높이기 위해 AND 및 OR을 사용하여 IN 연산을 작성할 수 있습니다.

다음 두 명령문 중에 두 번째 명령문이 더 빠르게 실행됩니다. 이 명령문은 IN() 또는 NOT IN() 문 대신 OR과 조합된 단순 동격 비교를 사용하기 때문에 속도가 더 빠릅니다.

```
-- Slower
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary IN (2000, 2500)

-- Faster
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary = 2000
    OR salary = 2500
```

성능 향상을 위해 SQL 문의 대체 형식을 사용하십시오..

이전 예에서 설명한 것처럼 SQL 문의 작성 방식에 따라 데이터베이스 성능에 영향을 줄 수 있습니다. SQL SELECT 문을 작성하여 특정 결과 집합을 검색하는 방법은 여러 가지가 있습니다. 일부 경우에는 다른 방법보다 속도가 크게 빠른 방법이 존재할 수 있습니다. 앞에서 설명한 권장 사항 외에도 SQL 언어에 대한 전문 리소스를 통해 여러 가지 SQL 문 및 성능 정보를 배울 수 있습니다.

SQL 문 성능

속도가 빠른 SQL 문을 확인하기 위해 SQL 문을 직접 비교하십시오..

여러 SQL 문의 성능을 비교하는 가장 좋은 방법은 데이터베이스 및 데이터로 직접 테스트해 보는 것입니다.

다음과 같은 개발 도구에서는 SQL 문을 실행할 때 걸리는 실행 시간을 제공합니다. 이를 사용하여 여러 명령문의 속도를 비교해 보십시오.

- Run!(Paul Robertson의 AIR SQL 쿼리 제작 및 테스트 도구)
- Lita(David Deraedt의 SQLite 관리 도구)

9장: 벤치마킹 및 배포

벤치마킹

벤치마킹 응용 프로그램에 사용할 수 있는 여러 가지 도구가 있습니다. Flash 커뮤니티 구성원이 개발한 Stats 클래스 및 PerformanceTest 클래스를 사용할 수 있습니다. 또한 Adobe® Flash® Builder™ 및 FlexPMD 도구에서 프로파일러를 사용할 수도 있습니다.

Stats 클래스

외부 도구 없이 런타임의 릴리스 버전을 사용하여 런타임에 코드를 프로파일링하려면 Flash 커뮤니티의 mr. doob가 개발한 Stats 클래스를 사용할 수 있습니다. Stats 클래스는 https://github.com/mrdoob/Hi-ReS-Stats에서 다운로드할 수 있습니다.

Stats 클래스를 사용하여 다음과 같은 사항을 추적할 수 있습니다.

- 초당 렌더링되는 프레임 수(숫자가 높을수록 성능이 높음)
- 프레임을 렌더링하는 데 사용되는 밀리초(숫자가 낮을수록 성능이 높음)
- 코드에서 사용하는 메모리 양. 각 프레임에서 사용하는 메모리가 증가하는 경우 응용 프로그램에서 메모리 누수가 발생할 수 있습니다. 가능한 메모리 누수를 조사하는 것이 중요합니다.
- 응용 프로그램에서 사용한 최대 메모리 양

다운로드한 후에는 Stats 클래스를 다음과 같은 압축 코드와 함께 사용할 수 있습니다.

```
import net.hires.debug.*;
addChild( new Stats() );
```

Adobe® Flash® Professional 또는 Flash Builder에서 조건부 컴파일을 사용하여 Stats 객체를 활성화할 수 있습니다.

```
CONFIG::DEBUG
{
    import net.hires.debug.*;
    addChild( new Stats() );
}
```

DEBUG 상수의 값을 전환하여 Stats 객체의 컴파일을 활성화 또는 비활성화할 수 있습니다. 동일한 방식을 사용하여 응용 프로그램에서 컴파일하지 않을 모든 코드 논리를 바꿀 수 있습니다.

PerformanceTest 클래스

ActionScript 코드 실행을 프로파일링하기 위해 Grant Skinner는 단위 테스팅 작업 과정에 통합될 수 있는 도구를 개발했습니다. 코드에 대한 일련의 테스트를 수행하는 PerformanceTest 클래스에 사용자 정의 클래스를 전달합니다. 이 PerformanceTest 클래스를 사용하면 여러 가지 방식을 쉽게 벤치마킹할 수 있습니다. PerformanceTest 클래스는 http://www.gskinner.com/blog/archives/2009/04/as3_performance.html에서 다운로드할 수 있습니다.

Flash Builder 프로파일러

Flash Builder에는 매우 자세히 코드를 벤치마킹할 수 있는 프로파일러가 함께 제공됩니다.

참고: 디버거 버전의 Flash Player를 사용하여 프로파일러에 액세스하십시오. 그러지 않으면 오류 메시지가 표시됩니다.

프로파일러를 Adobe Flash Professional에서 만든 내용에 사용할 수도 있습니다. 이렇게 하려면 ActionScript 또는 Flex 프로젝트에서 Flash Builder로 컴파일된 SWF 파일을 로드하십시오. 그러면 해당 파일에 대해 프로파일러를 실행할 수 있습니다. 프로파일러에 대한 자세한 내용은 Using Flash Builder 4의 "Profiling Flex applications"를 참조하십시오.

FlexPMD

Adobe Technical Services에서는 ActionScript 3.0 코드의 품질을 감사할 수 있는 FlexPMD라는 도구를 출시했습니다. FlexPMD는 JavaPMD와 비슷한 ActionScript 도구입니다. FlexPMD는 ActionScript 3.0 또는 Flex 소스 디렉토리를 감사하여 코드 품질을 향상시킵니다. 이 도구는 사용되지 않는 코드, 지나치게 복잡한 코드, 지나치게 긴 코드, Flex 구성 요소 수명 주기의 잘못된 사용 등 문제가 될 수 있는 코딩 방식을 감지합니다.

FlexPMD는 Adobe 오픈 소스 프로젝트이며, http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD에서 사용할 수 있습니다. Eclipse 플러그인도 http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD+Eclipse+plugin에서 사용할 수 있습니다.

FlexPMD를 사용하면 코드를 보다 쉽게 감사할 수 있으며, 문제 없이 최적화된 상태로 코드를 유지할 수 있습니다. FlexPMD의 가장 강력한 장점은 확장성입니다. 개발자는 고유한 규칙 집합을 만들어 모든 코드를 감사할 수 있습니다. 예를 들어 과도한 필터 사용이나 포착하려는 잘못된 코딩 방식을 감지하는 규칙 집합을 만들 수 있습니다.

배포

Flash Builder에서 최종 버전의 응용 프로그램을 내보낼 때는 응용 프로그램의 릴리스 버전을 내보내야 합니다. 릴리스 버전을 내보내면 SWF 파일에 포함된 디버깅 정보가 제거됩니다. 디버깅 정보를 제거하면 SWF 파일 크기가 보다 작아지므로 응용 프로그램 실행 속도를 높일 수 있습니다.

릴리스 버전의 프로젝트를 내보내려면 Flash Builder의 프로젝트 패널 및 릴리스 빌드 내보내기 옵션을 사용하십시오.

참고: Flash Professional에서 프로젝트를 컴파일할 때 릴리스 버전과 디버깅 버전 중에서 선택할 수 없습니다. 컴파일된 SWF 파일은 기본적으로 릴리스 버전입니다.