Optimizing Performance for the
ADOBE® FLASH® PLATFORM

Legal notices

For legal notices, see http://help.adobe.com/en_US/legalnotices/index.html.

Contents

Chapter 1: Introduction

Adobe® AIR® and Adobe® Flash® Player applications run on many platforms, including desktops, mobile devices, tablets, and television devices. Through code examples and use cases, this document outlines best practices for developers deploying these applications. Topics include:

- Conserving memory
- Minimizing CPU usage
- Improving ActionScript 3.0 performance
- Increasing rendering speed
- Optimizing network interaction
- Working with audio and video
- Optimizing SQL database performance
- Benchmarking and deploying applications

Most of these optimizations apply to applications on all devices, on both the AIR runtime and Flash Player runtime. Additions and exceptions for specific devices are also discussed.

Some of these optimizations focus on capabilities introduced in Flash Player 10.1 and AIR 2.5. However, many of these optimizations apply to earlier AIR and Flash Player releases, too.

Runtime code execution fundamentals

One key to understanding how to improve application performance is to understand how the Flash Platform runtime executes code. The runtime operates in a loop with certain actions occurring each "frame." A frame in this case is simply a block of time determined by the frame rate specified for the application. The amount of time allotted to each frame directly corresponds to the frame rate. For example, if you specify a frame rate of 30 frames per second, the runtime attempts to make each frame last one-thirtieth of a second.

You specify the initial frame rate for your application at authoring time. You can set the frame rate using settings in Adobe® Flash® Builder™ or Flash Professional. You can also specify the initial frame rate in code. Set the frame rate in an ActionScript-only application by applying the `[SWF(frameRate="24")]` metadata tag to your root document class. In MXML, set the `frameRate` attribute in the Application or WindowedApplication tag.

Each frame loop consists of two phases, divided into three parts: events, the `enterFrame` event, and rendering.

The first phase includes two parts (events and the `enterFrame` event), both of which potentially result in your code being called. In the first part of the first phase, runtime events arrive and are dispatched. These events can represent completion or progress of asynchronous operations, such as a response from loading data over a network. They also include events from user input. As events are dispatched, the runtime executes your code in listeners you've registered. If no events occur, the runtime waits to complete this execution phase without performing any action. The runtime never speeds up the frame rate due to lack of activity. If events occur during other parts of the execution cycle, the runtime queues up those events and dispatches them in the next frame.

The second part of the first phase is the `enterFrame` event. This event is distinct from the others because it is always dispatched once per frame.

Once all the events are dispatched, the rendering phase of the frame loop begins. At that point the runtime calculates the state of all visible elements on the screen and draws them to the screen. Then the process repeats itself, like a runner going around a racetrack.

Note: For events that include an `updateAfterEvent` property, rendering can be forced to occur immediately instead of waiting for the rendering phase. However, avoid using `updateAfterEvent` if it frequently leads to performance problems.

It's easiest to imagine that the two phases in the frame loop take equal amounts of time. In that case, during half of each frame loop event handlers and application code are running, and during the other half, rendering occurs. However, the reality is often different. Sometimes application code takes more than half the available time in the frame, stretching its time allotment, and reducing the allotment available for rendering. In other cases, especially with complex visual content such as filters and blend modes, the rendering requires more than half the frame time. Because the actual time taken by the phases is flexible, the frame loop is commonly known as the "elastic racetrack."

If the combined operations of the frame loop (code execution and rendering) take too long, the runtime isn't able to maintain the frame rate. The frame expands, taking longer than its allotted time, so there is a delay before the next frame is triggered. For example, if a frame loop takes longer than one-thirtieth of a second, the runtime is not able to update the screen at 30 frames per second. When the frame rate slows, the experience degrades. At best animation becomes choppy. In worse cases, the application freezes and the window goes blank.

For more details about the Flash Platform runtime code execution and rendering model, see the following resources:

- Flash Player Mental Model - The Elastic Racetrack (article by Ted Patrick)

- Asynchronous ActionScript Execution (article by Trevor McCauley)

- Optimizing Adobe AIR for code execution, memory & rendering at
  http://www.adobe.com/go/learn_fp_air_perf_tv_en (Video of MAX conference presentation by Sean Christmann)

Perceived performance versus actual performance

The ultimate judges of whether your application performs well are the application's users. Developers can measure application performance in terms of how much time certain operations take to run, or how many instances of objects are created. However, those metrics aren't important to end users. Sometimes users measure performance by different criteria. For example, does the application operate quickly and smoothly, and respond quickly to input? Does it have a negative affect on the performance of the system? Ask yourself the following questions, which are tests of perceived performance:

- Are animations smooth or choppy?

- Does video content look smooth or choppy?

- Do audio clips play continuously, or do they pause and resume?

- Does the window flicker or turn blank during long operations?

- When you type, does the text input keep up or lag behind?

- If you click, does something happen immediately, or is there a delay?

- Does the CPU fan get louder when the application runs?

- On a laptop computer or mobile device, does the battery deplete quickly while running the application?

- Do other applications respond poorly when the application is running?

The distinction between perceived performance and actual performance is important. The way to achieve the best perceived performance isn't always the same as the way to get the absolute fastest performance. Make sure that your application never executes so much code that the runtime isn't able to frequently update the screen and gather user input. In some cases, achieving this balance involves dividing up a program task into parts so that, between parts, the runtime updates the screen. (See "Rendering performance" on page 45 for specific guidance.)

The tips and techniques described here target improvements in both actual code execution performance and in how users perceive performance.

Target your optimizations

Some performance improvements do not create a noticeable improvement for users. It's important to concentrate your performance optimizations on areas that are problems for your specific application. Some performance optimizations are general good practices and can always be followed. For other optimizations, whether they are useful depends on your application's needs and its anticipated user base. For example, applications always perform better if you don't use any animation, video, or graphic filters and effects. However, one of the reasons for using the Flash Platform to build applications is because of the media and graphics capabilities that allow rich expressive applications. Consider whether your desired level of richness is a good match for the performance characteristics of the machines and devices on which your application runs.

One common piece of advice is to "avoid optimizing too early." Some performance optimizations require writing code in a way that is harder to read or less flexible. Such code, once optimized, is more difficult to maintain. For these optimizations, it is often better to wait and determine whether a particular section of code performs poorly before choosing to optimize the code.

Improving performance sometimes involves making trade-offs. Ideally, reducing the amount of memory consumed by an application also increases the speed at which the application performs a task. However, that type of ideal improvement isn't always possible. For example, if an application freezes during an operation, the solution often involves dividing up work to run over multiple frames. Because the work is being divided up, it is likely to take longer overall to accomplish the process. However, it is possible for the user to not notice the additional time, if the application continues to respond to input and doesn't freeze.

One key to knowing what to optimize, and whether optimizations are helpful, is to conduct performance tests. Several techniques and tips for testing performance are described in "Benchmarking and deploying" on page 91.

For more information about determining parts of an application that are good candidates for optimization, see the following resources:

* Performance-tuning apps for AIR at http://www.adobe.com/go/learn_fp_goldman_tv_en (Video of MAX conference presentation by Oliver Goldman)

* Performance-tuning Adobe AIR applications at http://www.adobe.com/go/learn_fp_air_perf_devnet_en (Adobe Developer Connection article by Oliver Goldman, based on the presentation)

Chapter 2: Conserving memory

Conserving memory is always important in application development, even for desktop applications. Mobile devices, however, place a premium on memory consumption, and it is worthwhile to limit the amount of memory your application consumes.

Display objects

Choose an appropriate display object.

ActionScript 3.0 includes a large set of display objects. One of the most simple optimization tips to limit memory usage is to use the appropriate type of display object. For simple shapes that are not interactive, use Shape objects. For interactive objects that don't need a timeline, use Sprite objects. For animation that uses a timeline, use MovieClip objects. Always choose the most efficient type of object for your application.

The following code shows memory usage for different display objects:

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

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

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

The `getSize()` method shows how many bytes an object consumes in memory. You can see that using multiple MovieClip objects instead of simple Shape objects can waste memory if the capabilities of a MovieClip object are not needed.

Primitive types

Use the `getSize()` method to benchmark code and determine the most efficient object for the task.

All primitive types except String use 4 – 8 bytes in memory. There is no way to optimize memory by using a specific type for a primitive:

```
// 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
```

A Number, which represents a 64-bit value, is allocated 8 bytes by the ActionScript Virtual Machine (AVM), if it is not assigned a value. All other primitive types are stored in 4 bytes.

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

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

The behavior differs for the String type. The amount of storage allocated is based on the length of the String:

```
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
```

Use the `getSize()` method to benchmark code and determine the most efficient object for the task.

Reusing objects

Reuse objects, when possible, instead of recreating them.

Another simple way to optimize memory is to reuse objects and avoid recreating them whenever possible. For example, in a loop, do not use the following code:

```
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);
}
```

Recreating the Rectangle object in each loop iteration uses more memory and is slower because a new object is created in each iteration. Use the following approach:

```
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);
}
```

The previous example used an object with a relatively small memory impact. The next example demonstrates larger memory savings by reusing a BitmapData object. The following code to create a tiling effect wastes memory:

```
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);
}
```

Note: *When using positive values, casting the rounded value to int is much faster than using the* `Math.floor()` *method.*

The following image shows the result of the bitmap tiling:
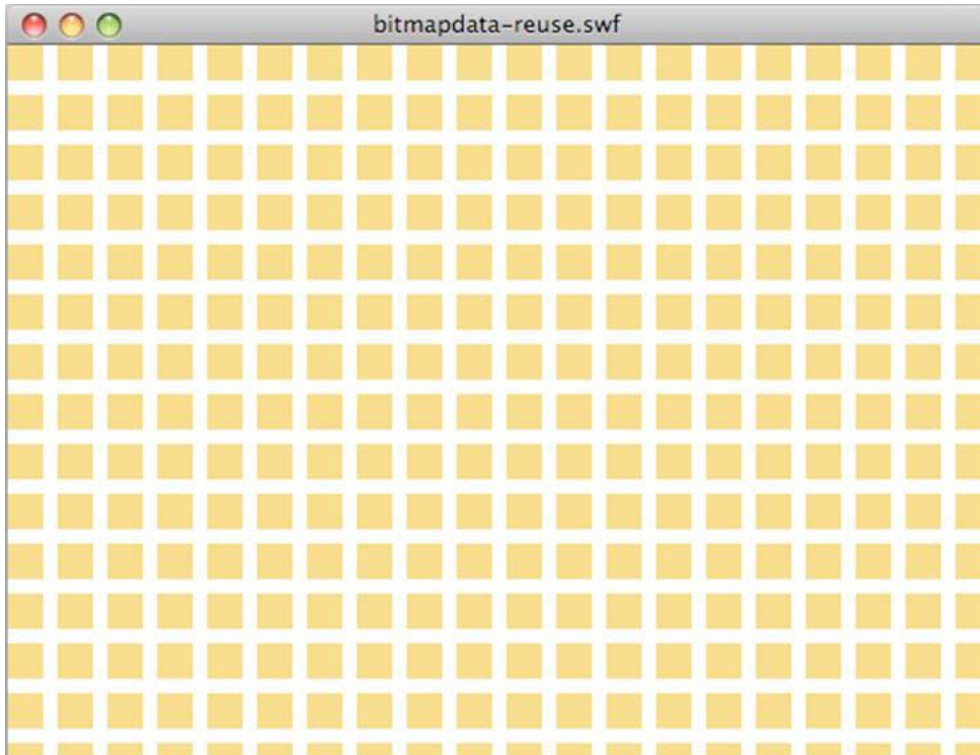
Result of bitmap tiling

An optimized version creates a single BitmapData instance referenced by multiple Bitmap instances and produces the same result:

```
// 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);
}
```

This approach saves about 700 KB in memory, which is a significant savings on a traditional mobile device. Each bitmap container can be manipulated without altering the original BitmapData instance by using the Bitmap properties:

```
// 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();
}
```

The following image shows the result of the bitmap transformations:

Result of bitmap transformations

More Help topics
"Bitmap caching" on page 51

Object pooling

Use object pooling, when possible.

Another important optimization is called object pooling, which involves reusing objects over time. You create a defined number of objects during the initialization of your application and store them inside a pool, such as an Array or Vector object. Once you are done with an object, you deactivate it so that it does not consume CPU resources, and you remove all mutual references. However, you do not set the references to `null`, which would make it eligible for garbage collection. You just put the object back into the pool, and retrieve it when you need a new object.

Reusing objects reduces the need to instantiate objects, which can be expensive. It also reduces the chances of the garbage collector running, which can slow down your application. The following code illustrates the object pooling technique:

```
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;
        }
    }
}
```

The SpritePool class creates a pool of new objects at the initialization of the application. The `getSprite()` method returns instances of these objects, and the `disposeSprite()` method releases them. The code allows the pool to grow when it has been consumed completely. It's also possible to create a fixed-size pool where new objects would not be allocated when the pool is exhausted. Try to avoid creating new objects in loops, if possible. For more information, see "Freeing memory" on page 11. The following code uses the SpritePool class to retrieve new instances:

```
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 );
    }
}
```

The following code removes all the display objects from the display list when the mouse is clicked, and reuses them later for another task:

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

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

Note: The pool vector always references the Sprite objects. If you want to remove the object from memory completely, you would need a `dispose()` method on the SpritePool class, which would remove all remaining references.

Freeing memory

Delete all references to objects to make sure that garbage collection is triggered.

You cannot launch the garbage collector directly in the release version of Flash Player. To make sure that an object is garbage collected, delete all references to the object. Keep in mind that the old `delete` operator used in ActionScript 1.0 and 2.0 behaves differently in ActionScript 3.0. It can only be used to delete dynamic properties on a dynamic object.

Note: You can call the garbage collector directly in Adobe® AIR® and in the debug version of Flash Player.

For example, the following code sets a Sprite reference to `null`:

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

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

Remember that when an object is set to `null`, it is not necessarily removed from memory. Sometimes the garbage collector does not run, if available memory is not considered low enough. Garbage collection is not predictable. Memory allocation, rather than object deletion, triggers garbage collection. When the garbage collector runs, it finds graphs of objects that haven't been collected yet. It detects inactive objects in the graphs by finding objects that reference each other, but that the application no longer uses. Inactive objects detected this way are deleted.

In large applications, this process can be CPU-intensive and can affect performance and generate a noticeable slowdown in the application. Try to limit garbage collection passes by reusing objects as much as possible. Also, set references to null, when possible, so that the garbage collector spends less processing time finding the objects. Think of garbage collection as insurance, and always manage object lifetimes explicitly, when possible.

Note: Setting a reference to a display object to null does not ensure that the object is frozen. The object continues consume CPU cycles until it is garbage collected. Make sure that you properly deactivate your object before setting its reference to `null`.

The garbage collector can be launched using the `System.gc()` method, available in Adobe AIR and in the debug version of Flash Player. The profiler bundled with Adobe® Flash® Builder™ allows you to start the garbage collector manually. Running the garbage collector allows you to see how your application responds and whether objects are correctly deleted from memory.

Note: If an object was used as an event listener, another object can reference it. If so, remove event listeners using the `removeEventListener()` method before setting the references to `null`.

Fortunately, the amount of memory used by bitmaps can be instantly reduced. For example, the BitmapData class includes a `dispose()` method. The next example creates a BitmapData instance of 1.8 MB. The current memory in use grows to 1.8 MB, and the `System.totalMemory` property returns a smaller value:

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

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

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

Next, the BitmapData is manually removed (disposed) from memory and the memory use is once again checked:

```
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
```

Although the `dispose()` method removes the pixels from memory, the reference must still be set to `null` to release it completely. Always call the `dispose()` method and set the reference to `null` when you no longer need a BitmapData object, so that memory is freed immediately.

Note: Flash Player 10.1 and AIR 1.5.2 introduce a new method called `disposeXML()` on the System class. This method allows you to make an XML object immediately available for garbage collection, by passing the XML tree as a parameter.

More Help topics

"Freezing and unfreezing objects" on page 26

Using bitmaps

Using vectors instead of bitmaps is good way to save memory. However, using vectors, especially in large numbers, dramatically increases the need for CPU or GPU resources. Using bitmaps is a good way to optimize rendering, because the runtime needs fewer processing resources to draw pixels on the screen than to render vector content.

More Help topics

"Manual bitmap caching" on page 58

Bitmap downsampling

To make better use of memory, 32-bit opaque images are reduced to 16-bit images when Flash Player detects a 16-bit screen. This downsampling consumes half the memory resources, and images render more quickly. This feature is available only in Flash Player 10.1 for Windows Mobile.

*Note: Before Flash Player 10.1, all pixels created in memory were stored in 32 bits (4 bytes). A simple logo of 300 x 300 pixels consumed 350 KB of memory (300*300*4/1024). With this new behavior, the same opaque logo consumes only 175 KB. If the logo is transparent, it is not downsampled to 16 bits, and it keeps the same size in memory. This feature applies only to embedded bitmaps or runtime-loaded images (PNG, GIF, JPG).*

On mobile devices, it can be hard to tell the difference between an image rendered in 16 bits and the same image rendered in 32 bits. For a simple image containing just a few colors, there is no detectable difference. Even for a more complex image, it is difficult to detect the differences. However, there can be some color degradation when zooming in on the image, and a 16-bit gradient can look less smooth than the 32-bit version.

BitmapData single reference

It is important to optimize the use of the BitmapData class by reusing instances as much as possible. Flash Player 10.1 and AIR 2.5 introduce a new feature for all platforms called BitmapData single reference. When creating BitmapData instances from an embedded image, a single version of the bitmap is used for all BitmapData instances. If a bitmap is modified later, it is given its own unique bitmap in memory. The embedded image can be from the library or an [Embed] tag.

Note: Existing content also benefits from this new feature, because Flash Player 10.1 and AIR 2.5 automatically reuse bitmaps.

When instantiating an embedded image, an associated bitmap is created in memory. Before Flash Player 10.1 and AIR 2.5, each instance was given a separate bitmap in memory, as shown in the following diagram:

Bitmaps in memory before Flash Player 10.1 and AIR 2.5
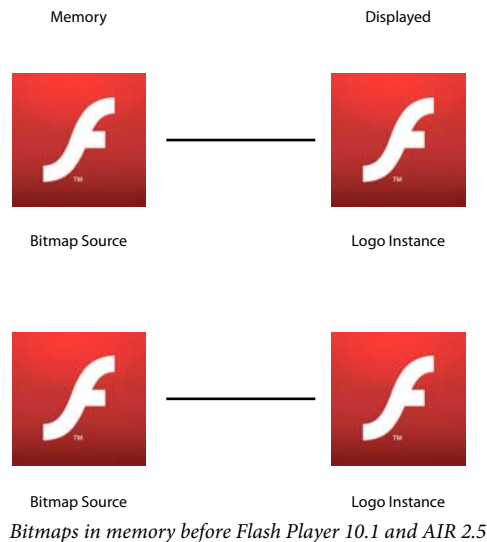
In Flash Player 10.1 and AIR 2.5, when multiple instances of the same image are created, a single version of the bitmap is used for all BitmapData instances. The following diagram illustrates this concept:
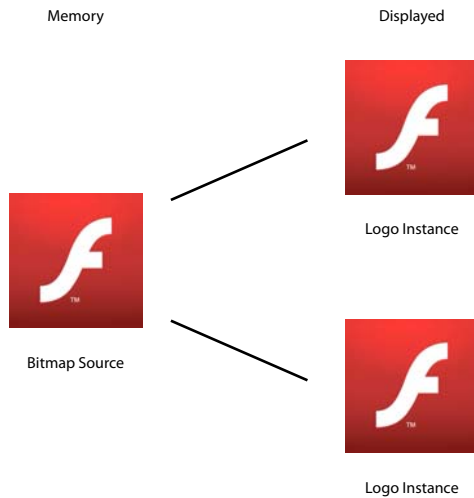
Bitmaps in memory in Flash Player 10.1 and AIR 2.5

This approach dramatically reduces the amount of memory used by an application with many bitmaps. The following code creates multiple instances of a `Star` symbol:

```
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)
    }
}
```

The following image shows the result of the code:
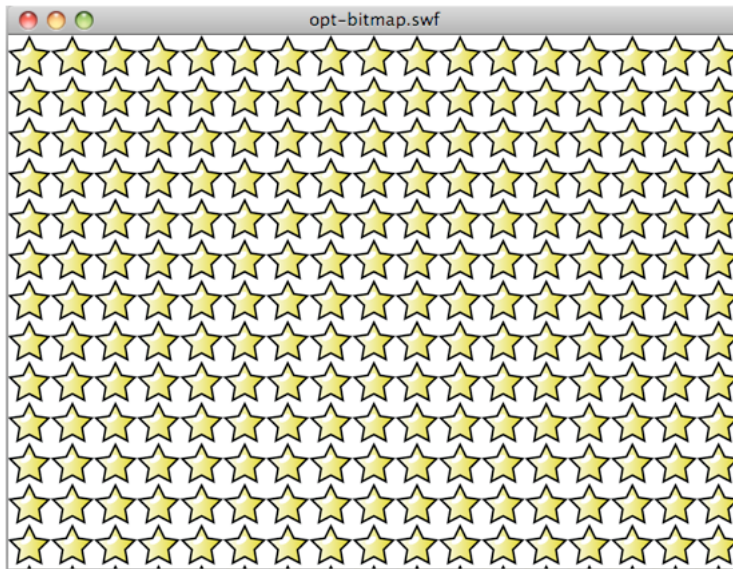
Result of code to create multiple instances of symbol

With Flash Player 10, for example, the animation above uses about 1008 KB of memory. With Flash Player 10.1, on the desktop and on a mobile device, the animation uses only 4 KB.

The following code modifies one BitmapData instance:

```
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);
```

The following image shows the result of modifying one `Star` instance:
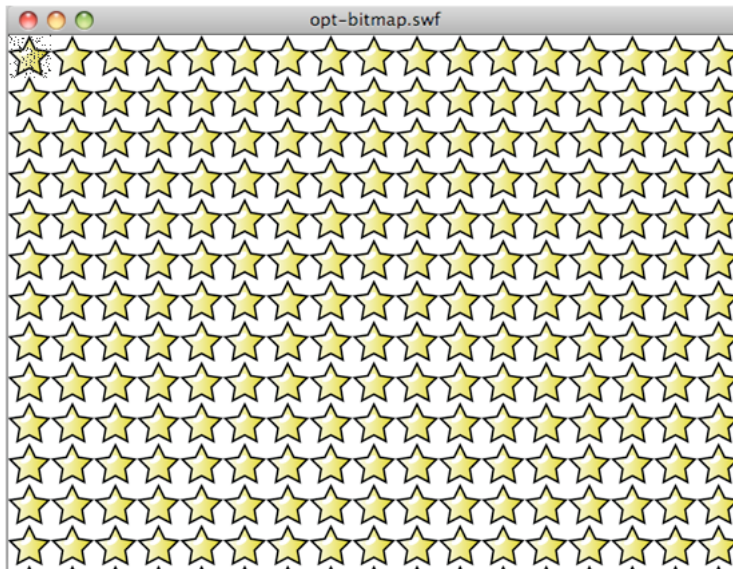
Result of modifying one instance

Internally, the runtime automatically assigns and creates a bitmap in memory to handle the pixel modifications. When a method of the BitmapData class is called, leading to pixel modifications, a new instance is created in memory, and no other instances are updated. The following figure illustrates the concept:
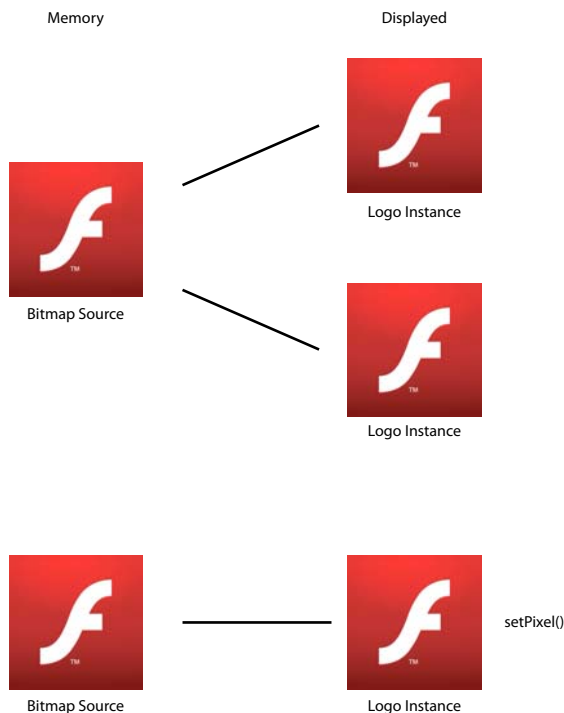
Result in memory of modifying one bitmap

If one star is modified, a new copy is created in memory. The resulting animation uses about 8 KB in memory on Flash Player 10.1 and AIR 2.5.

In the previous example, each bitmap is available individually for transformation. To create only the tiling effect, the `beginBitmapFill()` method is the most appropriate method:

```
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);
```

This approach produces the same result with only a single BitmapData instance created. To rotate the stars continuously, instead of accessing each Star instance, use a Matrix object that is rotated on each frame. Pass this Matrix object to the `beginBitmapFill()` method:

```
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);
}
```

Using this technique, no ActionScript loop is required to create the effect. The runtime does everything internally. The following image shows the result of transforming the stars:
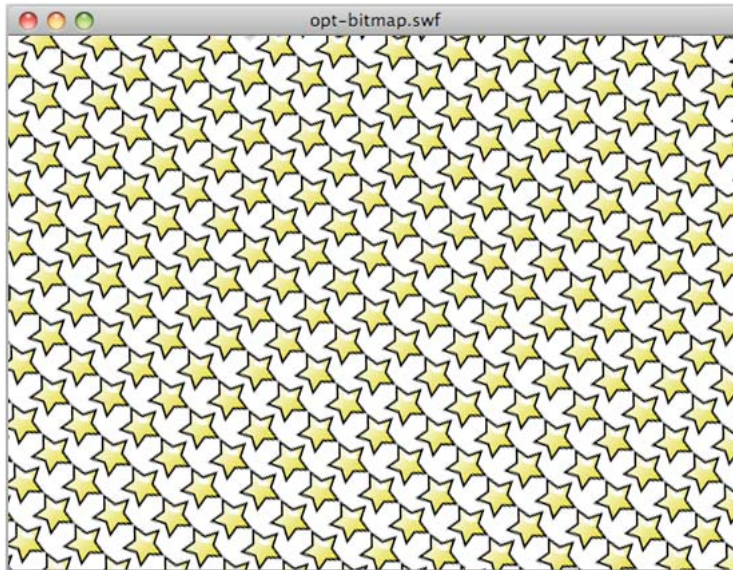
Result of rotating stars

With this approach, updating the original source BitmapData object automatically updates its use elsewhere on the stage, which can be a powerful technique. This approach would not, however, allow each star to be scaled individually, as in the previous example.

Note: When using multiple instances of the same image, drawing depends on whether a class is associated with the original bitmap in memory. If no class is associated with the bitmap, images are drawn as Shape objects with bitmap fills.

Filters and dynamic bitmap unloading

Avoid filters, including filters processed through Pixel Bender.

Try to minimize the use of effects like filters, including filters processed in mobile devices through Pixel Bender. When a filter is applied to a display object, the runtime creates two bitmaps in memory. These bitmaps are each the size of the display object. The first is created as a rasterized version of the display object, which in turn is used to produce a second bitmap with the filter applied:
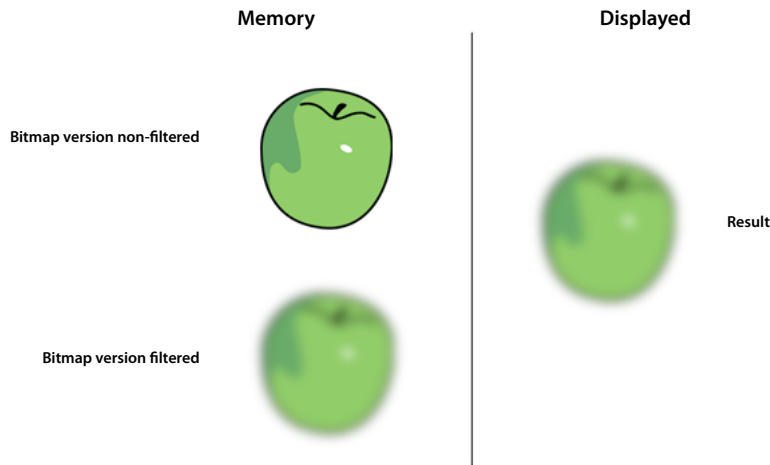
Two bitmaps in memory when filter is applied

When modifying one of the properties of a filter, both bitmaps are updated in memory to create the resulting bitmap. This process involves some CPU processing and the two bitmaps can use a significant amount of memory.

Flash Player 10.1 and AIR 2.5 introduce a new filtering behavior on all platforms. If the filter is not modified within 30 seconds, or if it is hidden or offscreen, the memory used by the non-filtered bitmap is freed.

This feature saves half the memory used by a filter on all platforms. For example, consider a text object with a blur filter applied. The text in this case is used for simple decoration and is not modified. After 30 seconds, the non-filtered bitmap in memory is freed. The same result occurs if the text is hidden for 30 seconds or is offscreen. When one of the filter properties is modified, the non-filtered bitmap in memory is recreated. This feature is called dynamic bitmap unloading. Even with these optimizations, be cautious with filters; they still require extensive CPU or GPU processing when being modified.

As a best practice, use bitmaps created through an authoring tool, such as Adobe® Photoshop®, to emulate filters when possible. Avoid using dynamic bitmaps created at runtime in ActionScript. Using externally authored bitmaps helps the runtime to reduce the CPU or GPU load, especially when the filter properties do not change over time. If possible, create any effects that you need on a bitmap in an authoring tool. You can then display the bitmap in the runtime without performing any processing on it, which can be much faster.

Direct mipmapping

💡 *Use mipmapping to scale large images, if needed.*

Another new feature available in Flash Player 10.1 and AIR 2.5 on all platforms is related to mipmapping. Flash Player 9 and AIR 1.0 introduced a mipmapping feature that improved the quality and performance of downscaled bitmaps.

Note: The mipmapping feature applies only to dynamically loaded images or embedded bitmaps. Mipmapping does not apply to display objects that have been filtered or cached. Mipmapping can be processed only if the bitmap has a width and height that are even numbers. When a width or height that is an odd number is encountered, mipmapping stops. For example, a 250 x 250 image can be mipmapped down to 125 x 125, but it cannot be mipmapped further. In this case, at least one of the dimensions is an odd number. Bitmaps with dimensions that are powers of two achieve the best results, for example: 256 x 256, 512 x 512, 1024 x 1024, and so on.

As an example, imagine that a 1024 x 1024 image is loaded, and a developer wants to scale the image to create a thumbnail in a gallery. The mipmapping feature renders the image properly when scaled by using the intermediate downsampled versions of the bitmap as textures. Previous versions of the runtime created intermediate downscaled versions of the bitmap in memory. If a 1024 x 1024 image was loaded and displayed at 64 x 64, older versions of the runtime would create every half-sized bitmap. For example, in this case 512 x 512, 256 x 256, 128 x 128, and 64 x 64 bitmaps would be created.

Flash Player 10.1 and AIR 2.5 now support mipmapping directly from the original source to the destination size required. In the previous example, only the 4 MB (1024 x 1024) original bitmap and the 16 KB (64 x 64) mipmapped bitmap would be created.

The mipmapping logic also works with the dynamic bitmap unloading feature. If only the 64 x 64 bitmap is used, the 4-MB original bitmap is freed from memory. If the mipmap must be recreated, then the original is reloaded. Also, if other mipmapped bitmaps of various sizes are required, the mipmap chain of bitmaps is used to create the bitmap. For example, if a 1:8 bitmap must be created, the 1:4 and 1:2 and 1:1 bitmaps are examined to determine which is loaded into memory first. If no other versions are found, the 1:1 original bitmap is loaded from the resource and used.

The JPEG decompressor can perform mipmapping within its own format. This direct mipmapping allows a large bitmap to be decompressed directly to a mipmap format without loading the entire uncompressed image. Generating the mipmap is substantially faster, and memory used by large bitmaps is not allocated and then freed. The JPEG image quality is comparable to the general mipmapping technique.

Note: Use mipmapping sparingly. Although it improves the quality of downscaled bitmaps, it has an impact on bandwidth, memory, and speed. In some cases, a better option can be to use a pre-scaled version of the bitmap from an external tool and import it into your application. Don't start with large bitmaps if you only intend to scale them down.

Using 3D effects

Consider creating 3D effects manually.

Flash Player 10 and AIR 1.5 introduced a 3D engine, which allows you to apply perspective transformation on display objects. You can apply these transformations using the `rotationX` and `rotationY` properties or with the `drawTriangles()` method of the Graphics class. You can also apply depth with the `z` property. Keep in mind that each perspective-transformed display object is rasterized as a bitmap and therefore requires more memory.

The following figure illustrates the anti-aliasing produced by the rasterization when using perspective transformation:
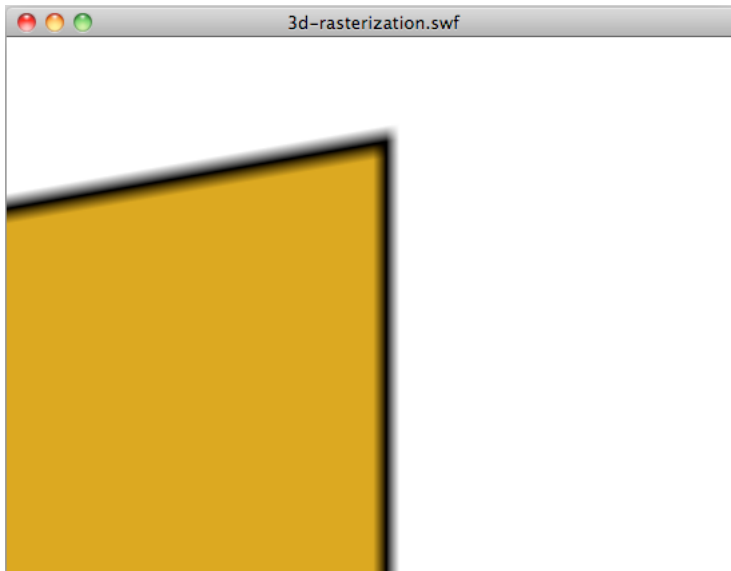
Anti-aliasing resulting from perspective transformation

The anti-aliasing is a result of vector content being dynamically rasterized as a bitmap. This anti-aliasing happens when you use 3D effects in the desktop version of AIR and Flash Player, and in AIR 2.0.1 and AIR 2.5 for mobile. However, anti-aliasing is not applied on Flash Player for mobile devices.

If you can create your 3D effect manually without relying on the native API, that can reduce memory usage. However, the new 3D features introduced in Flash Player 10 and AIR 1.5 make texture mapping easier, because of methods like `drawTriangles()`, which handles texture mapping natively.

As a developer, decide whether the 3D effect you want to create provides better performance if it is processed through the native API or manually. Consider ActionScript execution and rendering performance, as well as memory usage.

In AIR 2.0.1 and AIR 2.5 mobile applications in which you set the `renderMode` application property to `GPU`, the GPU does the 3D transformations. However, if the `renderMode` is `CPU`, the CPU, not the GPU, performs the 3D transformations. In Flash Player 10.1 applications, the CPU performs the 3D transformations.

When the CPU does the 3D transformations, consider that applying any 3D transformation to a display object requires two bitmaps in memory. One bitmap is for the source bitmap, and a second one is for the perspective-transformed version. In this way, 3D transformations work in a similar way to filters. As a result, use the 3D properties sparingly when the CPU does the 3D transformations.

Text objects and memory

💡 *Use the Adobe® Flash® Text Engine for read-only text; use TextField objects for input text.*

Flash Player 10 and AIR 1.5 introduced a powerful new text engine, the Adobe Flash Text Engine (FTE), that conserves system memory. However, the FTE is a low-level API that requires an additional ActionScript 3.0 layer on top of it, provided in the flash.text.engine package.

For read-only text, it's best to use the Flash Text Engine, which offers low memory usage and better rendering. For input text, TextField objects are a better choice, because less ActionScript code is required to create typical behaviors, such as input handling and word-wrap.

More Help topics

"Rendering text objects" on page 64

Event model versus callbacks

Consider using simple callbacks, instead of the event model.

The ActionScript 3.0 event model is based on the concept of object dispatching. The event model is object-oriented and optimized for code reuse. The `dispatchEvent()` method loops through the list of listeners and calls the event handler method on each registered object. However, one of the drawbacks of the event model is that you are likely to create many objects over the lifetime of your application.

Imagine that you must dispatch an event from the timeline, indicating the end of an animation sequence. To accomplish the notification, you can dispatch an event from a specific frame in the timeline, as the following code illustrates:

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

The Document class can listen for this event with the following line of code:

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

Although this approach is correct, using the native event model can be slower and consume more memory than using a traditional callback function. Event objects must be created and allocated in memory, which creates a performance slowdown. For example, when listening to the Event.ENTER_FRAME event, a new event object is created on each frame for the event handler. Performance can be especially slow for display objects, due to the capture and bubbling phases, which can be expensive if the display list is complex.

Chapter 3: Minimizing CPU usage

Another important area of focus for optimization is CPU usage. Optimizing CPU processing improves performance, and as a result, battery life on mobile devices.

Flash Player 10.1 enhancements for CPU usage

Flash Player 10.1 introduces two new features that help save CPU processing. The features involve pausing and resuming SWF content when it goes offscreen, and limiting the number of Flash Player instances on a page.

Pause and resume

Note: The pause and resume feature does not apply to Adobe® AIR® applications.

To optimize CPU and battery usage, Flash Player 10.1 introduces a new feature on mobile platforms (and netbooks) related to inactive instances. This feature allows you to limit CPU usage by pausing and resuming the SWF file when content goes off and on the screen. With this feature, Flash Player releases as much memory as possible by removing any objects that can be recreated when the playing of content is resumed. Content is considered offscreen when the entire content is offscreen.

Two scenarios cause the SWF content to be offscreen:

* The user scrolls the page and causes the SWF content to move offscreen.

  In this case, if there is any audio or video playback, content continues to play, but rendering is stopped. If there is no audio or video playing, to ensure that the playback or ActionScript execution is not paused, set the `hasPriority` HTML parameter to true. However, keep in mind that SWF content rendering is paused when content is offscreen or hidden, regardless of the value of the `hasPriority` HTML parameter.

* A tab is opened in the browser, which causes the SWF content to move to the background.

  In this case, regardless of the value of the `hasPriority` HTML tag, the SWF content is slowed down to 2 fps. Audio and video playback is stopped and no content rendering is processed unless the SWF content becomes visible again.

Instance management

Note: The instance management feature does not apply to Adobe® AIR® applications.

Use the `hasPriority` HTML parameter to delay loading of offscreen SWF files.
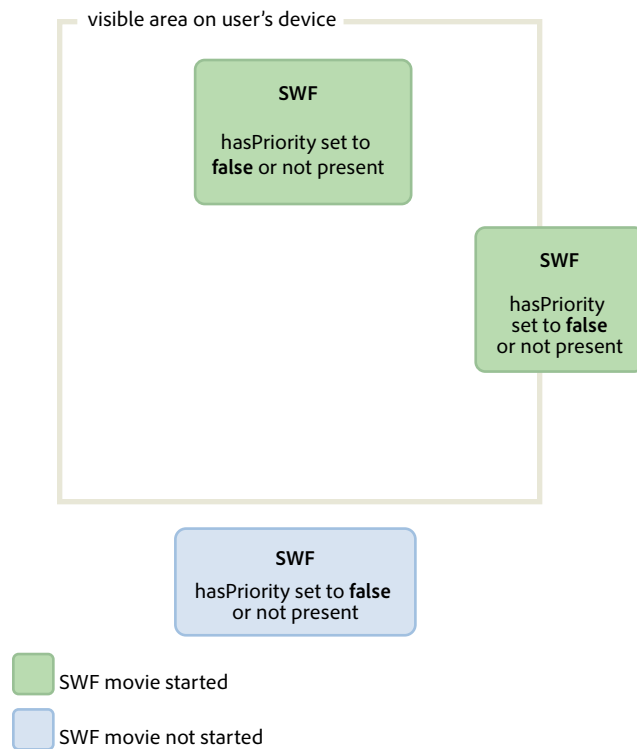
Flash Player 10.1 introduces a new HTML parameter called `hasPriority`:

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

This feature limits the number of Flash Player instances that are started on a page. Limiting the number of instances helps conserve CPU and battery resources. The idea is to assign a specific priority to SWF content, giving some content priority over other content on a page. Consider a simple example: a user is browsing a website and the index page hosts three different SWF files. One of them is visible, another one is partially visible onscreen, and the last one is offscreen, requiring scrolling. The first two animations are started normally, but the last one is deferred until it becomes visible. This scenario is the default behavior when the `hasPriority` parameter is not present or set to `false`. To ensure that a SWF file is started, even if it is offscreen, set the `hasPriority` parameter to `true`. However, regardless of the value of the `hasPriority` parameter, a SWF file that is not visible to the user always has its rendering paused.

Note: If available CPU resources become low, Flash Player instances are no longer started automatically, even if the `hasPriority` parameter is set to `true`. If new instances are created through JavaScript after the page has been loaded, those instances will ignore the `hasPriority` flag. Any 1x1 pixel or 0x0 pixel content is started, preventing helper SWF files from being deferred if the webmaster fails to include the `hasPriority` flag. SWF files can still be started when clicked, however. This behavior is called "click to play."

The following diagrams show the effects of setting the `hasPriority` parameter to different values:

Effects of different values for the hasPriority parameter
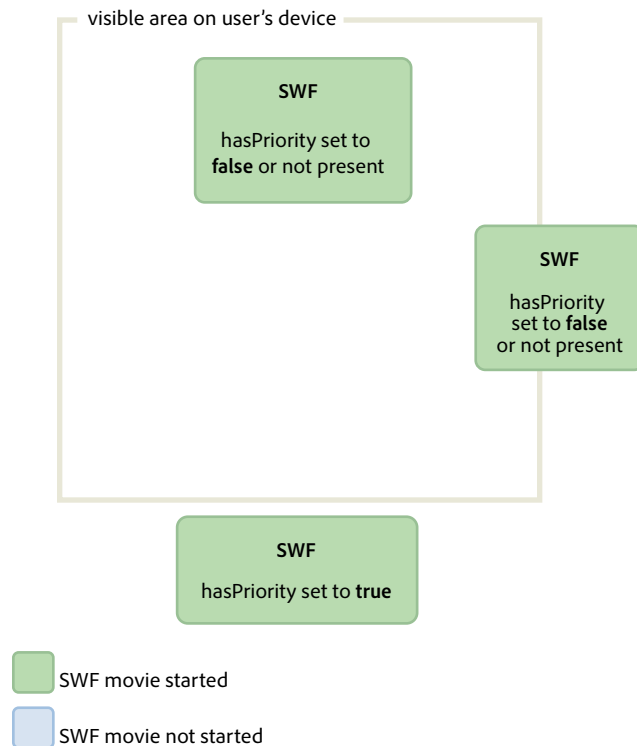
Effects of different values for the hasPriority parameter

Sleep mode

Flash Player 10.1 and AIR 2.5 introduce a new feature on mobile devices that helps save CPU processing, and as a result, battery life. This feature involves the backlight found on many mobile devices. For example, if a user running a mobile application is interrupted and stops using the device, the runtime detects when the backlight goes into sleep mode. It then drops the frame rate to 4 frames per second (fps), and pauses rendering. For AIR applications, sleep mode also begins when the application moves to the background.

ActionScript code continues to execute in sleep mode, similar to setting the `Stage.frameRate` property to 4 fps. But the rendering step is skipped, so the user cannot see that the player is running at 4 fps. A frame rate of 4 fps was chosen, rather than zero, because it allows all the connections to remain open (NetStream, Socket, and NetConnection). Switching to zero would break open connections. A 250 ms refresh rate was chosen (4 fps) because many device manufacturers use this frame rate as their refresh rate. Using this value keeps the frame rate of the runtime in the same ballpark as the device itself.

Note: When the runtime is in sleep mode, the `Stage.frameRate` property returns the frame rate of the original SWF file, rather than 4 fps.

When the backlight goes back into on mode, rendering is resumed. The frame rate returns to its original value. Consider a media player application in which a user is playing music. If the screen goes into sleep mode, the runtime responds based on the type of content being played. Here is a list of situations with the corresponding runtime behavior:

• The backlight goes into sleep mode and non-A/V content is playing: The rendering is paused and the frame rate is set to 4 fps.

- The backlight goes into sleep mode and A/V content is playing: the runtime forces the backlight to be always on, continuing the user experience.

- The backlight goes from sleep mode to on mode: the runtime sets the frame rate to the original SWF file frame rate setting and resumes rendering.

- Flash Player is paused while A/V content is played: Flash Player resets the backlight state to the default system behavior because A/V is no longer playing.

- Mobile device receives a phone call while A/V content is played: The rendering is paused and the frame rate is set to 4 fps.

- The backlight sleep mode is disabled on a mobile device: the runtime behaves normally.

When the backlight goes into sleep mode, rendering pauses and the frame rate slows down. This feature saves CPU processing, but it cannot be relied upon on to create a real pause, as in a game application.

Note: No ActionScript event is dispatched when the runtime enters or leaves sleep mode.

Freezing and unfreezing objects

💡 *Freeze and unfreeze objects properly by using the* REMOVED_FROM_STAGE *and* ADDED_TO_STAGE *events.*

To optimize your code, always freeze and unfreeze your objects. Freezing and unfreezing are important for all objects, but are especially important for display objects. Even if display objects are no longer in the display list and are waiting to be garbage collected, they could still be using CPU-intensive code. For example, they can still be using Event.ENTER_FRAME. As a result, it is critical to freeze and unfreeze objects properly with the Event.REMOVED_FROM_STAGE and Event.ADDED_TO_STAGE events. The following example shows a movie clip playing on stage that interacts with the keyboard:

```
// 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;
    }
}
```
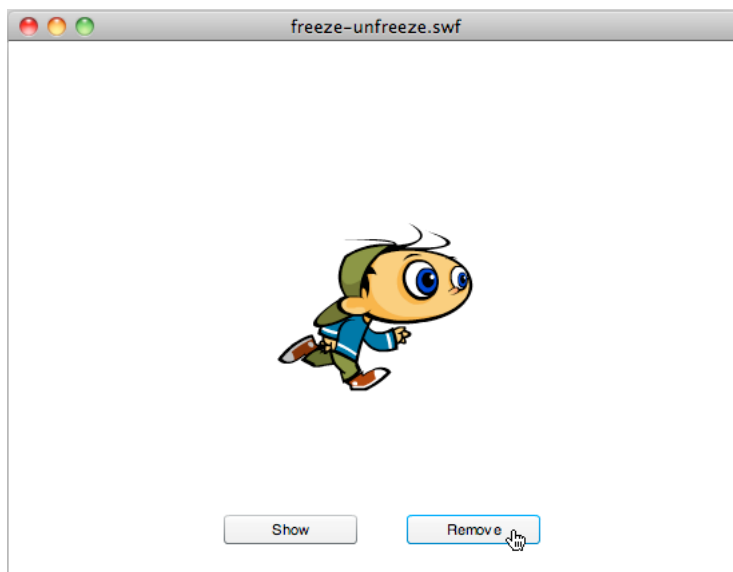

Movie clip that interacts with keyboard

When the Remove button is clicked, the movie clip is removed from the display list:

```
// 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);
}
```

Even when removed from the display list, the movie clip still dispatches the `Event.ENTER_FRAME` event. The movie clip still runs, but it is not rendered. To handle this situation correctly, listen to the proper events and remove event listeners, to prevent CPU-intensive code from being executed:

```
// 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();
}
```

When the Show button is pressed, the movie clip is restarted, it listens to `Event.ENTER_FRAME` events again, and the keyboard correctly controls the movie clip.

Note: If a display object is removed from the display list, setting its reference to `null` after removing it does not ensure that the object is frozen. If the garbage collector doesn't run, the object continues to consume memory and CPU processing, even though the object is no longer displayed. To make sure that the object consumes the least CPU processing possible, make sure that you completely freeze it when removing it from the display list.

Starting with Flash Player 10 and AIR 1.5, the following behavior also occurs. If the playhead encounters an empty frame, the display object is automatically frozen even if you did not implement any freezing behavior.

The concept of freezing is also important when loading remote content with the Loader class. When using the Loader class with Flash Player 9 and AIR 1.0, it was necessary to manually freeze content by listening to the `Event.UNLOAD` event dispatched by the LoaderInfo object. Every object had to be manually frozen, which was a non-trivial task. Flash Player 10 and AIR 1.5 introduced an important new method on the Loader class called `unloadAndStop()`. This method allows you to unload a SWF file, automatically freeze every object in the loaded SWF file, and force the garbage collector to run.

In the following code, the SWF file is loaded and then unloaded using the `unload()` method, which requires more processing and manual freezing:

```
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();
}
```

A best practice is to use the `unloadAndStop()` method, which handles the freezing natively and forces the garbage collecting process to run:

```
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();
}
```

The following actions occur when the `unloadAndStop()` method is called:

• Sounds are stopped.

• Listeners registered to the SWF file's main timeline are removed.

• Timer objects are stopped.

• Hardware peripheral devices (such as camera and microphone) are released.

• Every movie clip is stopped.

• Dispatching of `Event.ENTER_FRAME`, `Event.FRAME_CONSTRUCTED`, `Event.EXIT_FRAME`, `Event.ACTIVATE` and `Event.DEACTIVATE` is stopped.

Activate and deactivate events

💡 *Use `Event.ACTIVATE` and `Event.DEACTIVATE` events to detect background inactivity and optimize your application appropriately.*

Two events (`Event.ACTIVATE` and `Event.DEACTIVATE`) can assist you in fine-tuning your application so that it uses the fewest CPU cycles possible. These events allow you to detect when the runtime gains or loses focus. As a result, code can be optimized to react to context changes. The following code listens to both events and dynamically changes the frame rate to zero when the application loses its focus. For example, the animation can lose focus when the user switches to another tab or puts the application into the background:

```
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;
}
```

When the application gains focus again, the frame rate is reset to its original value. Instead of changing the frame rate dynamically, you could also consider making other optimizations, such as freezing and unfreezing objects.

The activate and deactivate events allow you to implement a similar mechanism to the "Pause and Resume" feature sometimes found on mobile devices and Netbooks.

More Help topics

"Application frame rate" on page 50

"Freezing and unfreezing objects" on page 26

Mouse interactions

Consider disabling mouse interactions, when possible.

When using an interactive object, such as a MovieClip or Sprite object, the runtime executes native code to detect and handle mouse interactions. Detecting mouse interaction can be CPU-intensive when many interactive objects are shown onscreen, especially if they overlap. An easy way to avoid this processing is to disable mouse interactions on objects that do not require any mouse interaction. The following code illustrates the use of the `mouseEnabled` and `mouseChildren` properties:

```
// 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;
```

When possible, consider disabling mouse interaction, which helps your application to use less CPU processing, and as a result, reduce battery usage on mobile devices.

Timers versus ENTER_FRAME events

💡 *Choose either timers or ENTER_FRAME events, depending on whether content is animated.*

Timers are preferred over Event.ENTER_FRAME events for non-animated content that executes for a long time.

In ActionScript 3.0, there are two ways of calling a function at specific intervals. The first approach is to use the Event.ENTER_FRAME event dispatched by display objects (DisplayObject). The second approach is to use a timer. ActionScript developers frequently use the ENTER_FRAME event approach. The ENTER_FRAME event is dispatched on every frame. As a result, the interval at which the function is called is related to the current frame rate. The frame rate is accessible through the Stage.frameRate property. However, in some cases, using a timer can be a better choice than using the ENTER_FRAME event. For example, if you don't use animation, but would like your code called at specific intervals, using a timer can be a better choice.

A timer can behave in a similar way to an ENTER_FRAME event, but an event can be dispatched without being tied to the frame rate. This behavior can offer some significant optimization. Consider a video player application as an example. In this case, you do not need to use a high frame rate, because only the application controls are moving.

Note: The frame rate does not affect the video, because the video is not embedded in the timeline. Instead, the video is loaded dynamically through progressive downloading or streaming.

In this example, the frame rate is set to a low value of 10 fps. The timer updates the controls at a rate of one update per second. The higher update rate is made possible by the updateAfterEvent() method, which is available on the TimerEvent object. This method forces the screen to be updated each time the timer dispatches an event, if needed. The following code illustrates the idea:

```
// 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();
}
```

Calling the `updateAfterEvent()` method does not modify the frame rate. It just forces the runtime to update the content onscreen that has changed. The timeline still runs at 10 fps. Remember that timers and `ENTER_FRAME` events are not perfectly accurate on low performance devices, or if event handler functions contain code that requires heavy processing. Just like the SWF file frame rate, the update frame rate of the timer can vary in some situations.

💡 *Minimize the number of Timer objects and registered `enterFrame` handlers in your application.*

Each frame, the runtime dispatches an `enterFrame` event to each display object in its display list. Although you can register listeners for the `enterFrame` event with multiple display objects, doing so means that more code is executed each frame. Instead, consider using a single centralized `enterFrame` handler that executes all the code that is to run each frame. By centralizing this code, it is easier to manage all the code that is running frequently.

Likewise, if you're using Timer objects, there is overhead associated with creating and dispatching events from multiple Timer objects. If you must trigger different operations at different intervals, here are some suggested alternatives:

- Use a minimal number of Timer objects and group operations according to how frequently they happen.

  For example, use one Timer for frequent operations, set to trigger every 100 milliseconds. Use another Timer for less-frequent or background operations, set to trigger every 2000 milliseconds.

- Use a single Timer object, and have operations triggered at multiples of the Timer object's `delay` property interval.

  For example, suppose you have some operations that are expected to happen every 100 milliseconds, and others that you want to happen every 200 milliseconds. In that case, use a single Timer object with a `delay` value of 100 milliseconds. In the `timer` event handler, add a conditional statement that only runs the 200-millisecond operations every other time. The following example demonstrates this technique:

```
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;
}
```

Stop Timer objects when not in use.

If a Timer object's `timer` event handler only performs operations under certain conditions, call the Timer object's `stop()` method when none of the conditions are true.

In `enterFrame` event or Timer handlers, minimize the number of changes to the appearance of display objects that cause the screen to be redrawn.

Each frame, the rendering phase redraws the portion of the stage that has changed during that frame. If the redraw region is large, or if it's small but contain a large quantity or complex display objects, the runtime needs more time for rendering. To test the amount of redrawing required, use the "show redraw regions" feature in the debug Flash Player or AIR.

For more information about improving performance for repeated actions, see the following article:

• Writing well-behaved, efficient, AIR applications (article and sample application by Arno Gourdol)

More Help topics
"Isolating behaviors" on page 61

Tweening syndrome

To save CPU power, limit the use of tweening, which saves CPU processing, memory, and battery life.

Designers and developers producing content for Flash on the desktop tend to use many motion tweens in their applications. When producing content for mobile devices with low performance, try to minimize the use of motion tweens. Limiting their use helps content run faster on low-tier devices.

Chapter 4: ActionScript 3.0 performance

Vector class versus Array class

Use the Vector class instead of the Array class, when possible.

The Vector class allows faster read and write access than the Array class.

A simple benchmark shows the benefits of the Vector class over the Array class. The following code shows a benchmark for the Array class:

```
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
```

The following code shows a benchmark for the Vector class:

```
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
```

The example can be further optimized by assigning a specific length to the vector and setting its length to fixed:

```
// 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
```

If the size of the vector is not specified ahead of time, the size increases when the vector runs out of space. Each time the size of the vector increases, a new block of memory is allocated. The current content of the vector is copied into the new block of memory. This extra allocation and copying of data hurts performance. The above code is optimized for performance by specifying the initial size of the vector. However, the code is not optimized for maintainability. To also improve maintainability, store the reused value in a constant:

```
// 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
```

Try to use the Vector object APIs, when possible, as they are likely to run faster.

Drawing API

Use the drawing API for faster code execution.

Flash Player 10 and AIR 1.5 provided a new drawing API, which allows you to get better code execution performance. This new API does not provide rendering performance improvement, but it can dramatically reduce the number of lines of code you have to write. Fewer lines of code can provide better ActionScript execution performance.

The new drawing API includes the following methods:

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

Note: This discussion does not focus on the `drawTriangles()` method, which is related to 3D. However, this method can improve ActionScript performance, because it handles native texture mapping.

The following code explicitly calls the appropriate method for each line being drawn:

```
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 );
```

The following code runs faster than the previous example, because it executes fewer lines of code. The more complex the path, the greater the performance gain from using the `drawPath()` method:

```
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 );
```

The `drawGraphicsData()` method provides similar performance improvements.

Event capture and bubbling

💡 *Use event capture and bubbling to minimize event handlers.*

The event model in ActionScript 3.0 introduced the concepts of event capture and event bubbling. Taking advantage of the bubbling of an event can help you to optimize ActionScript code execution time. You can register an event handler on one object, instead of multiple objects, to improve performance.

As an example, imagine creating a game in which the user must click apples as fast as possible to destroy them. The game removes each apple from the screen when it is clicked and adds points to the user's score. To listen to the `MouseEvent.CLICK` event dispatched by each apple, you could be tempted to write the following code:

```
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 );
}
```

The code calls the `addEventListener()` method on each Apple instance. It also removes each listener when an apple is clicked, using the `removeEventListener()` method. However, the event model in ActionScript 3.0 provides a capture and bubbling phase for some events, allowing you to listen to them from a parent InteractiveObject. As a result, it is possible to optimize the code above and minimize the number of calls to the `addEventListener()` and `removeEventListener()` methods. The following code uses the capture phase to listen to the events from the parent object:

```
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 );
}
```

The code is simplified and much more optimized, with only one call to the `addEventListener()` method on the parent container. Listeners are no longer registered to the Apple instances, so there is no need to remove them when an apple is clicked. The `onAppleClick()` handler can be optimized further, by stopping the propagation of the event, which prevents it from going further:

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

The bubbling phase can also be used to catch the event, by passing `false` as the third parameter to the `addEventListener()` method:

```
// 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 );
```

The default value for the capture phase parameter is `false`, so you can omit it:

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

Working with pixels

Paint pixels using the `setVector()` method.

When painting pixels, some simple optimizations can be made just by using the appropriate methods of the BitmapData class. A fast way to paint pixels is to use the `setVector()` method:

```
// 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 );
```

When using slow methods, such as `setPixel()` or `setPixel32()`, use the `lock()` and `unlock()` methods to make things run faster. In the following code, the `lock()` and `unlock()` methods are used to improve performance:

```
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
```

The `lock()` method of the BitmapData class locks an image and prevents objects that reference it from being updated when the BitmapData object changes. For example, if a Bitmap object references a BitmapData object, you can lock the BitmapData object, change it, and then unlock it. The Bitmap object is not changed until the BitmapData object is unlocked. To improve performance, use this method along with the `unlock()` method before and after numerous calls to the `setPixel()` or `setPixel32()` method. Calling `lock()` and `unlock()` prevents the screen from being updated unnecessarily.

Note: When processing pixels on a bitmap not on the display list (double-buffering), sometimes this technique does not improve performance. If a bitmap object does not reference the bitmap buffer, using `lock()` and `unlock()` does not improve performance. Flash Player detects that the buffer is not referenced, and the bitmap is not rendered onscreen.

Methods that iterate over pixels, such as `getPixel()`, `getPixel32()`, `setPixel()`, and `setPixel32()`, are likely to be slow, especially on mobile devices. If possible, use methods that retrieve all the pixels in one call. For reading pixels, use the `getVector()` method, which is faster than the `getPixels()` method. Also, remember to use APIs that rely on Vector objects, when possible, as they are likely to run faster.

Regular expressions

💡 *Use String class methods such as `indexOf()`, `substr()`, or `substring()` instead of regular expressions for basic string finding and extraction.*

Certain operations that can be performed using a regular expression can also be accomplished using methods of the String class. For example, to find whether a string contains another string, you can either use the `String.indexOf()` method or use a regular expression. However, when a String class method is available, it runs faster than the equivalent regular expression and does not require the creation of another object.

💡 *Use a non-capturing group ("`(?:xxxx)`") instead of a group ("`(xxxx)`") in a regular expression to group elements without isolating the contents of the group in the result.*

Frequently in regular expressions of moderate complexity, you group parts of the expression. For example, in the following regular expression pattern the parentheses create a group around the text "ab." Consequently, the "+" quantifier applies to the group rather than to a single character:

`/(ab)+/`

By default, the contents of each group are "captured." You can get the contents of each group in your pattern as part of the result of executing the regular expression. Capturing these group results takes longer and requires more memory because objects are created to contain the group results. As an alternative, you can use the non-capturing group syntax by including a question mark and colon after the opening parenthesis. This syntax specifies that the characters behave as a group but are not captured for the result:

`/(?:ab)+/`

Using the non-capturing group syntax is faster and uses less memory than using the standard group syntax.

💡 *Consider using an alternative regular expression pattern if a regular expression performs poorly.*

Sometimes more than one regular expression pattern can be used to test for or identify the same text pattern. For various reasons, certain patterns execute faster than other alternatives. If you determine that a regular expression is causing your code to run slower than necessary, consider alternative regular expression patterns that achieve the same result. Test these alternative patterns to determine which is the fastest.

Miscellaneous optimizations

For a TextField object, use the `appendText()` *method instead of the* `+=` *operator.*

When working with the `text` property of the TextField class, use the `appendText()` method instead of the `+=` operator. Using the `appendText()` method provides performance improvements.

As an example, the following code uses the `+=` operator, and the loop takes 1120 ms to complete:

```
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
```

In the following example, the `+=` operator is replaced with the `appendText()` method:

```
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
```

The code now takes 847 ms to complete.

Update text fields outside loops, when possible.

This code can be optimized even further by using a simple technique. Updating the text field in each loop uses much internal processing. By simply concatenating a string and assigning it to the text field outside the loop, the time to run the code drops substantially. The code now takes 2 ms to complete:

```
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
```

When working with HTML text, the former approach is so slow that it can throw a `Timeout` exception in Flash Player, in some cases. For example, an exception can be thrown if the underlying hardware is too slow.

Note: Adobe® AIR® does not throw this exception.

```
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 );
```

By assigning the value to a string outside the loop, the code requires only 29 ms to complete:

```
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
```

Note: In Flash Player 10.1 and AIR 2.5, the String class has been improved so that strings use less memory.

Avoid using the square bracket operator, when possible.

Using the square bracket operator can slow down performance. You can avoid using it by storing your reference in a local variable. The following code example demonstrates inefficient use of the square bracket operator:

```
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
```

The following optimized version reduces the use of the square bracket operator:

```
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
```

Inline code, when possible, to reduce the number of function calls in your code.

Calling functions can be expensive. Try to reduce the number of function calls by moving code inline. Moving code inline is a good way to optimize for pure performance. However, keep in mind that inline code can make your code harder to reuse and can increase the size of your SWF file. Some function calls, such the Math class methods, are easily to move inline. The following code uses the `Math.abs()` method to calculate absolute values:

```
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
```

The calculation performed by `Math.abs()` can be done manually and moved inline:

```
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
```

Moving the function call inline results in code that is more than four times faster. This approach is useful in many situations, but be aware of the effect it can have on reusability and maintainability.

Note: Code size has a large impact on overall player execution. If the application includes large amounts of ActionScript code, then the virtual machine spends significant amounts of time verifying code and JIT compiling. Property lookups can be slower, because of deeper inheritance hierarchies and because internal caches tend to thrash more. To reduce code size, avoid using the Adobe® Flex® framework, the TLF framework library, or any large third-party ActionScript libraries.

Avoid evaluating statements in loops.

Another optimization can be achieved by not evaluating a statement inside a loop. The following code iterates over an array, but is not optimized because the array length is evaluated for each iteration:

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

It is better to store the value and reuse it:

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

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

Use reverse order for while loops.

A while loop in reverse order is faster than a forward loop:

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

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

These tips provide a few ways to optimize ActionScript, showing how a single line of code can affect performance and memory. Many other ActionScript optimizations are possible. For more information, see the following link: http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/.

Chapter 5: Rendering performance

Redraw regions

💡 *Always use the redraw regions option when building your project.*

To improve rendering, it's important to use the redraw regions option when building your project. Using this option allows you to see the regions that Flash Player is rendering and processing. You can enable this option by selecting Show Redraw Regions in the context menu of the debug version of Flash Player.

Note: *The Show Redraw Regions option is not available in Adobe AIR or in the release version of Flash Player. (In Adobe AIR, the context menu is available only in desktop applications, but it has no built-in or standard items such as Show Redraw Regions.)*

The image below illustrates the option enabled with a simple animated MovieClip on the timeline:
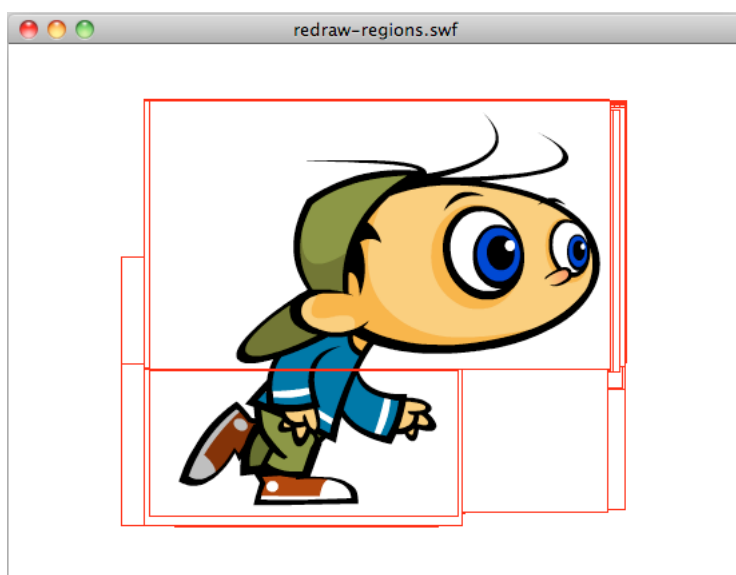
Redraw regions option enabled

You can also enable this option programmatically, by using the `flash.profiler.showRedrawRegions()` method:

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

In Adobe AIR applications, this method is the only way to enable the redraw regions option.

Use redraw regions to identify opportunities for optimization. Remember that although some display objects are not shown, they still consume CPU cycles because they are still being rendered. The following image illustrates this idea. A black vector shape covers the animated running character. The image shows that the display object has not been removed from the display list and is still being rendered. This wastes CPU cycles:
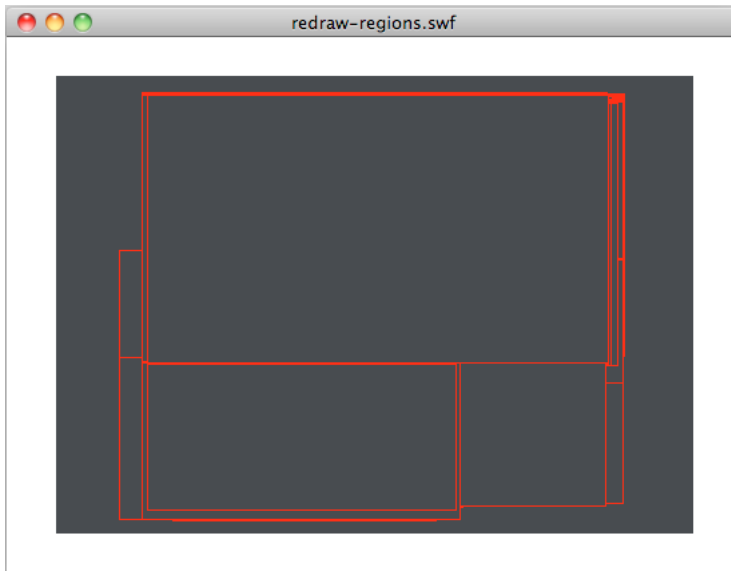
Redrawn regions

To improve performance, set the `visible` property of the hidden running character to `false` or remove it from the display list altogether. You should also stop its timeline. These steps ensure that the display object is frozen and uses minimal CPU power.

Remember to use the redraw regions option during the whole development cycle. Using this option prevents you from being surprised at the end of the project by unnecessary redraw regions and optimization areas that have been missed.

More Help topics

"Freezing and unfreezing objects" on page 26

Off-stage content

💡 *Avoid placing content off-stage. Instead, just place objects on the display list when needed.*

If possible, try not to place graphical content off-stage. Designers and developers commonly place elements off-stage to reuse assets during the lifetime of the application. The following figure illustrates this common technique:
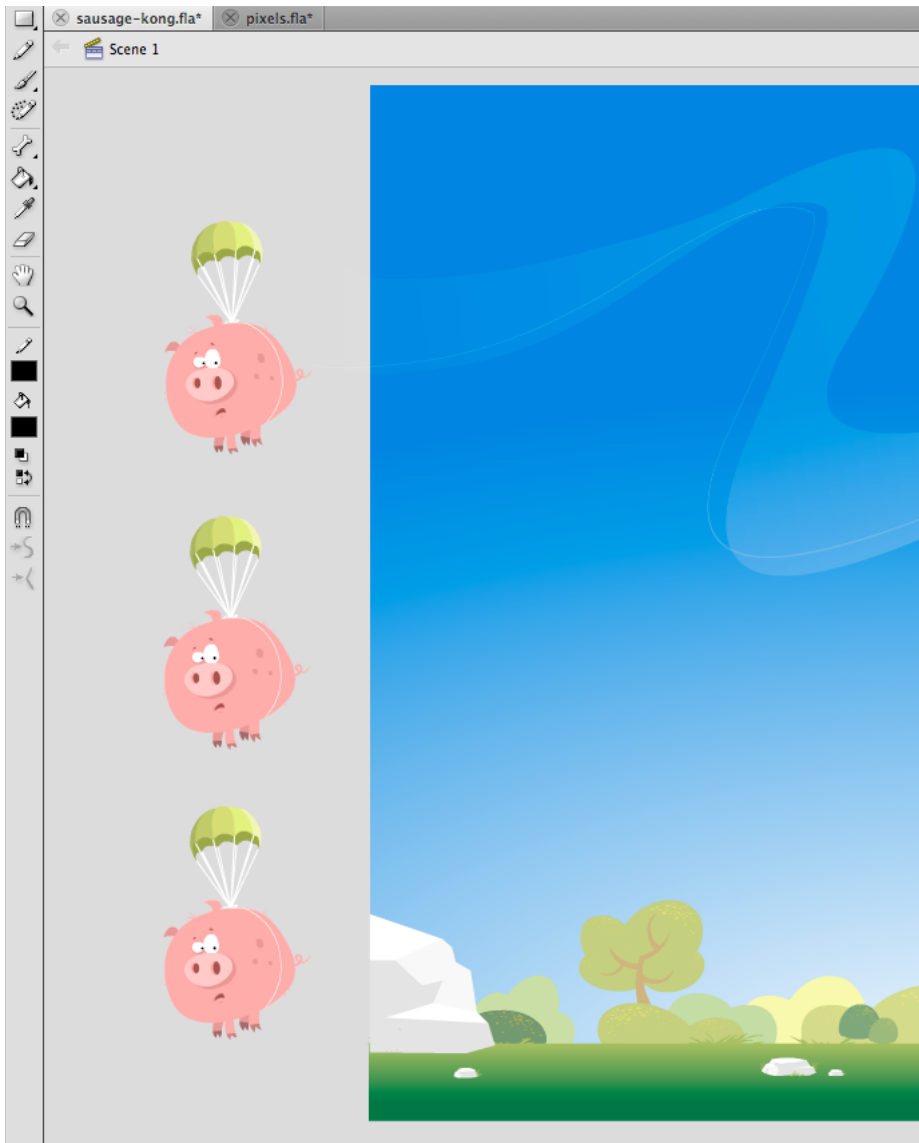
Off-stage content

Even if the off-stage elements are not shown onscreen and are not rendered, they still exist on the display list. The runtime continues to run internal tests on these elements to make sure that they are still off-stage and the user is not interacting with them. As a result, as much as possible, avoid placing objects off-stage and remove them from the display list instead.

Movie quality

Use the appropriate Stage quality setting to improve rendering.

When developing content for mobile devices with small screens, such as phones, image quality is less important than when developing desktop applications. Setting the Stage quality to the appropriate setting can improve rendering performance.

The following settings are available for Stage quality:

- `StageQuality.LOW`: Favors playback speed over appearance and does not use anti-aliasing. This setting is not supported in Adobe AIR for desktop or TV.

- `StageQuality.MEDIUM`: Applies some anti-aliasing but does not smooth scaled bitmaps. This setting is the default value for AIR on mobile devices, but is not supported in AIR for desktop or TV.

- `StageQuality.HIGH`: (Default on the desktop) Favors appearance over playback speed and always uses anti-aliasing. If the SWF file does not contain animation, scaled bitmaps are smoothed; if the SWF file contains animation, bitmaps are not smoothed.

- `StageQuality.BEST`: Provides the best display quality and does not consider playback speed. All output is anti-aliased and scaled bitmaps are always smoothed.

Using `StageQuality.MEDIUM` often provides enough quality for applications on mobile devices, and in some cases using `StageQuality.LOW` can provide enough quality. Since Flash Player 8, anti-aliased text can be rendered accurately, even with Stage quality set to `LOW`.

Note: On some mobile devices, even though the quality is set to `HIGH`, `MEDIUM` is used for better performance in Flash Player applications. However, setting the quality to HIGH often does not make a noticeable difference, because mobile screens usually have a higher dpi. (The dpi can vary depending on the device.)

In the following figure, the movie quality is set to `MEDIUM` and the text rendering is set to Anti-Alias for Animation:

Medium Stage quality and text rendering set to Anti-Alias for Animation

The Stage quality setting affects the text quality, because the appropriate text rendering setting is not being used.

The runtime allows you to set text rendering to Anti-Alias for Readability. This setting maintains perfect quality of your (anti-aliased) text regardless of which Stage quality setting you use:

Low Stage quality and text rendering set to Anti-Alias for Readability

The same rendering quality can be obtained by setting text rendering to Bitmap Text (No anti-alias):

Low Stage quality and text rendering set to Bitmap Text (No anti-alias)

The last two examples show that you can get high-quality text, no matter which Stage quality setting you use. This feature has been available since Flash Player 8 and can be used on mobile devices. Keep in mind that Flash Player 10.1 automatically switches to `StageQuality.MEDIUM` on some devices to increase performance.

Alpha blending

Avoid using the alpha property, when possible.

Avoid using effects that require alpha blending when using the `alpha` property, such as fading effects. When a display object uses alpha blending, the runtime must combine the color values of every stacked display object and the background color to determine the final color. Thus, alpha blending can be more processor-intensive than drawing an opaque color. This extra computation can hurt performance on slow devices. Avoid using the `alpha` property, when possible.

More Help topics
"Bitmap caching" on page 51

"Rendering text objects" on page 64

Application frame rate

💡 *In general, use the lowest possible frame rate for better performance.*

An application's frame rate determines how much time is available for each "application code and rendering" cycle, as described in "Runtime code execution fundamentals" on page 1. A higher frame rate creates smoother animation. However, when animation or other visual changes aren't happening, there is often no reason to have a high frame rate. A higher frame rate expends more CPU cycles and energy from the battery than a lower rate.

The following are some general guidelines for an appropriate default frame rate for your application:

• If you are using the Flex framework, leave the starting frame rate at the default value.

• If your application includes animation, an adequate frame rate is at least 20 frames per second. Anything more than 30 frames per second is often unnecessary.

• If your application doesn't include animation, a frame rate of 12 frames per second is probably sufficient.

The "lowest possible frame rate" can vary depending on the current activity of the application. See the next tip "Dynamically change the frame rate of your application" for more information.

💡 *Use a low frame rate when video is the only dynamic content in your application.*

The runtime plays loaded video content at its native frame rate, regardless of the frame rate of the application. If your application has no animation or other rapidly changing visual content, using a low frame rate does not degrade the experience of the user interface.

💡 *Dynamically change the frame rate of your application.*

You define the application's initial frame rate in project or compiler settings, but the frame rate is not fixed at that value. You can change the frame rate by setting the `Stage.frameRate` property (or the `WindowedApplication.frameRate` property in Flex).

Change the frame rate according to the current needs of your application. For example, during a time when your application isn't performing any animation, lower the frame rate. When an animated transition is about to start, raise the frame rate. Similarly, if your application is running in the background (after having lost focus), you can usually lower the frame rate even further. The user is likely to be focused on another application or task.

The following are some general guidelines to use as a starting point in determining the appropriate frame rate for different types of activities:

• If you are using the Flex framework, leave the starting frame rate at the default value.

• When animation is playing, set the frame rate to at least 20 frames per second. Anything more than 30 frames per second is often unnecessary.

• When no animation is playing, a frame rate of 12 frames per second is probably sufficient.

• Loaded video plays at its native frame rate regardless of the application frame rate. If video is the only moving content in your application, a frame rate of 12 frames per second is probably sufficient.

• When the application doesn't have input focus, a frame rate of 5 frames per second is probably sufficient.

• When an AIR application is not visible, a frame rate of 2 frames per second or less is probably appropriate. For example, this guideline applies if an application is minimized. It also applies on desktop devices if the native window's `visible` property is `false`.

For applications built in Flex, the spark.components. WindowedApplication class has built-in support for dynamically changing the application's frame rate. The `backgroundFrameRate` property defines the application's frame rate when the application isn't active. The default value is 1, which changes the frame rate of an application built with the Spark framework to 1 frame per second. You can change the background frame rate by setting the `backgroundFrameRate` property. You can set the property to another value, or set it to -1 to turn off automatic frame rate throttling.

For more information on dynamically changing an application's frame rate, see the following articles:

- Reducing CPU usage in Adobe AIR (Adobe Developer Center article and sample code by Jonnie Hallman)

- Writing well-behaved, efficient, AIR applications (article and sample application by Arno Gourdol)

Grant Skinner has created a frame rate throttler class. You can use this class in your applications to automatically decrease the frame rate when your application is in the background. For more information and to download the source code for the FramerateThrottler class, see Grant's article Idle CPU Usage in Adobe AIR and Flash Player at http://gskinner.com/blog/archives/2009/05/idle_cpu_usage.html.

Adaptive frame rate

When compiling a SWF file, you set a specific frame rate for the movie. In a restricted environment with a low CPU rate, the frame rate sometimes drops down during playback. To preserve an acceptable frame rate for the user, the runtime skips the rendering of some frames. Skipping the rendering of some frames keeps the frame rate from falling below an acceptable value.

Note: In this case, the runtime does not skip frames, it only skips the rendering of content on frames. Code is still executed and the display list is updated, but the updates don't show onscreen. There is no way to specify a threshold fps value indicating how many frames to skip if the runtime can't keep the frame rate stable.

Bitmap caching

Use the bitmap caching feature for complex vector content, when appropriate.

A good optimization can be made by using the bitmap caching feature. This feature caches a vector object, renders it as a bitmap internally, and uses that bitmap for rendering. The result can be a huge performance boost for rendering, but it can require a significant amount of memory. Use the bitmap caching feature for complex vector content, like complex gradients or text.

Turning on bitmap caching for an animated object that contains complex vector graphics (such as text or gradients) improves performance. However, if bitmap caching is enabled on a display object such as a movie clip that has its timeline playing, you get the opposite result. On each frame, the runtime must update the cached bitmap and then redraw it onscreen, which requires many CPU cycles. The bitmap caching feature is an advantage only when the cached bitmap can be generated once and then used without the need to update it.

If you turn on bitmap caching for a Sprite object, the object can be moved without causing the runtime to regenerate the cached bitmap. Changing the x and y properties of the object does not cause regeneration. However, any attempt to rotate it, scale it, or change its alpha value causes the runtime to regenerate the cached bitmap, and as a result, hurts performance.

Note: The `DisplayObject.cacheAsBitmapMatrix` property available in AIR and the Packager for iPhone does not have this limitation. By using the `cacheAsBitmapMatrix` property, you can rotate, scale, skew, and change the alpha value of a display object without triggering bitmap regeneration.

A cached bitmap can use more memory than a regular movie clip instance. For example, if the movie clip on the Stage is 250 x 250 pixels, when cached it uses about 250 KB, instead of 1 KB uncached.

The following example involves a Sprite object that contains an image of an apple. The following class is attached to the apple symbol:

```
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();
        }
    }
}
```

The code uses the Sprite class instead of the MovieClip class, because a timeline is not needed for each apple. For best performance, use the most lightweight object possible. Next, the class is instantiated with the following code:

```
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);
        }
    }
}
```

When the user clicks the mouse, the apples are created without caching. When the user presses the C key (keycode 67), the apple vectors are cached as bitmaps and shown onscreen. This technique greatly improves rendering performance, both on the desktop and on mobile devices, when the CPU is slow.

However, although using the bitmap caching feature improves rendering performance, it can quickly consume large amounts of memory. As soon as an object is cached, its surface is captured as a transparent bitmap and stored in memory, as shown in the following diagram:
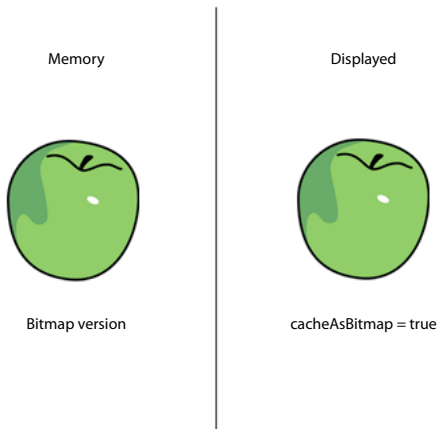
Object and its surface bitmap stored in memory

Flash Player 10.1 and AIR 2.5 optimize the use of memory by taking the same approach as described in the "Filters and dynamic bitmap unloading" on page 18. If a cached display object is hidden or offscreen, its bitmap in memory is freed when unused for a while.

Note: If the display object's opaqueBackground *property is set to a specific color, the runtime considers the display object to be opaque. When used with the* cacheAsBitmap *property, the runtime creates a non-transparent 32-bit bitmap in memory. The alpha channel is set to 0xFF, which improves performance, because no transparency is required to draw the bitmap onscreen. Avoiding alpha blending makes rendering even faster. If the current screen depth is limited to 16 bits, then the bitmap in memory is stored as a 16-bit image. Using the* opaqueBackground *property does not implicitly activate bitmap caching.*

To save memory, use the cacheAsBitmap property and activate it on each display object rather than on the container. Activating bitmap caching on the container makes the final bitmap much larger in memory, creating a transparent bitmap with dimensions of 211 x 279 pixels. The image uses around 229 KB of memory:
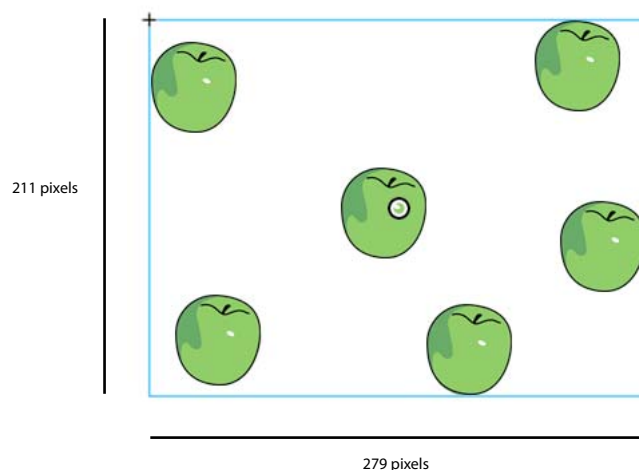
Activating bitmap caching on container

In addition, by caching the container, you risk having the whole bitmap updated in memory, if any apple starts to move on a frame. Activating the bitmap caching on the individual instances results in caching six 7-KB surfaces in memory, which uses only 42 KB of memory:
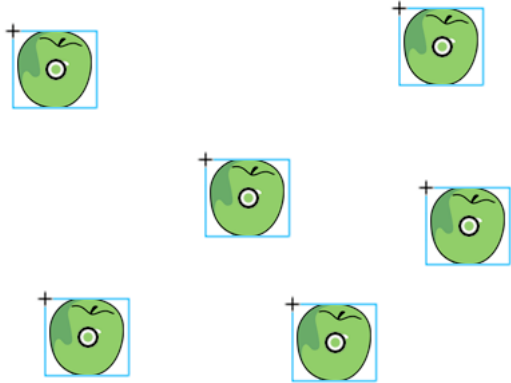
Activating bitmap caching on instances

Accessing each apple instance through the display list and calling the `getChildAt()` method stores references in a Vector object for easier access:

```
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);
        }
    }
}
```

Keep in mind that bitmap caching improves rendering if the cached content is not rotated, scaled, or changed on each frame. However, for any transformation other than translation on the x- and y-axes, rendering is not improved. In these cases, Flash Player updates the cached bitmap copy for every transformation occurring on the display object. Updating the cached copy can result in high CPU usage, slow performance, and high battery usage. Again, the cacheAsBitmapMatrix property in AIR or the Packager for iPhone does not have this limitation.

The following code changes the alpha value in the movement method, which changes the opacity of the apple on every frame:

```
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();
}
```

Using bitmap caching causes a performance slowdown. Changing the alpha value forces the runtime to update the cached bitmap in memory whenever the alpha value is modified.

Filters rely on bitmaps that are updated whenever the playhead of a cached movie clip moves. So, using a filter automatically sets the `cacheAsBitmap` property to `true`. The following figure illustrates an animated movie clip:

Animated movie clip

Avoid using filters on animated content, because it can cause performance problems. In the following figure, the designer adds a drop shadow filter:

Animated movie clip with drop shadow filter

As a result, if the timeline inside the movie clip is playing, the bitmap must be regenerated. The bitmap must also be regenerated if the content is modified in any way other than a simple x or y transformation. Each frame forces the runtime to redraw the bitmap, which requires more CPU resources, causes poor performance, and consumes more battery life.

Paul Trani provides examples of using Flash Professional and ActionScript to optimize graphics using bitmaps in the following training videos:

• Optimizing Graphics

• Optimizing Graphics with ActionScript

Cached bitmap transform matrixes in AIR

Set the `cacheAsBitmapMatrix` property when using cached bitmaps in mobile AIR apps.

In the AIR mobile profile, you can assign a Matrix object to the `cacheAsBitmapMatrix` property of a display object. When you set this property, you can apply any two-dimensional transformation to the object without regenerating the cached bitmap. You can also change the alpha property without regenerating the cached bitmap. The `cacheAsBitmap` property must also be set to `true` and the object must not have any 3D properties set.

Setting the `cacheAsBitmapMatrix` property generates the cached bitmap even if the display object is off screen, hidden from view, or has its `visible` property set to `false`. Resetting the `cacheAsBitmapMatrix` property using a matrix object that contains a different transform also regenerates the cached bitmap.

The matrix transformation you apply to the `cacheAsBitmapMatrix` property is applied to the display object as it is rendered into the bitmap cache. Thus, if the transform contains a 2x scale, the bitmap rendering is twice the size of the vector rendering. The renderer applies the inverse transformation to the cached bitmap so that the final display looks the same. You can scale the cached bitmap to a smaller size to reduce memory usage, possibly at the expense of rendering fidelity. You can also scale the bitmap to a larger size to increase rendering quality in some cases, at the expense of increased memory usage. In general, though, use an identity matrix, which is a matrix that applies no transformation, to avoid changes in appearance, as shown in the following example:

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

Once the `cacheAsBitmapMatrix` property is set, you can scale, skew, rotate, and translate the object without triggering bitmap regeneration.

You can also change the alpha value within the range of 0 and 1. If you change the alpha value through the `transform.colorTransform` property with a color transform, the alpha used in the transform object must be within 0 and 255. Changing the color transformation in any other way regenerates the cached bitmap.

Always set the `cacheAsBitmapMatrix` property whenever you set `cacheAsBitmap` to `true` in content created for mobile devices. However, consider the following potential drawback. After an object is rotated, scaled or skewed, the final rendering can exhibit bitmap scaling or aliasing artifacts compared to a normal vector rendering.

Manual bitmap caching

Use the BitmapData class to create custom bitmap caching behavior.

The following example reuses a single rasterized bitmap version of a display object and references the same BitmapData object. When scaling each display object, the original BitmapData object in memory is not updated and is not redrawn. This approach saves CPU resources and makes applications run faster. When scaling the display object, the contained bitmap is stretched.

Here is the updated `BitmapApple` class:

```
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();
        }
    }
}
```

The alpha value is still modified on each frame. The following code passes the original source buffer to each BitmapApple instance:

```
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);
}
```

This technique uses only a small amount of memory, because only a single cached bitmap is used in memory and shared by all the `BitmapApple` instances. In addition, despite any modifications that are made to the `BitmapApple` instances, such as alpha, rotation, or scaling, the original source bitmap is never updated. Using this technique prevents a performance slowdown.

For a smooth final bitmap, set the `smoothing` property to `true`:

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

    smoothing = true;

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

Adjusting the Stage quality can also improve performance. Set the Stage quality to `HIGH` before rasterization, then switch to `LOW` afterwards:

```
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 );
}
```

Toggling Stage quality before and after drawing the vector to a bitmap can be a powerful technique to get anti-aliased content onscreen. This technique can be effective regardless of the final stage quality. For example, you can get an anti-aliased bitmap with anti-aliased text, even with the Stage quality set to LOW. This technique is not possible with the cacheAsBitmap property. In that case, setting the Stage quality to LOW updates the vector quality, which updates the bitmap surfaces in memory, and updates the final quality.

Isolating behaviors

Isolate events such as the Event.ENTER_FRAME *event in a single handler, when possible.*

The code can be further optimized by isolating the Event.ENTER_FRAME event in the Apple class in a single handler. This technique saves CPU resources. The following example shows this different approach, where the BitmapApple class no longer handles the movement behavior:

```
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;
        }
    }
}
```

The following code instantiates the apples and handles their movement in a single handler:

```
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;
            }
        }
}
```

The result is a single Event.ENTER_FRAME event handling the movement, instead of 200 handlers moving each apple. The whole animation can be paused easily, which can be useful in a game.

For example, a simple game can use the following handler:

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

The next step is to make the apples interact with the mouse or keyboard, which requires modifications to the BitmapApple class.

```
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);
        }
    }
}
```

The result is `BitmapApple` instances that are interactive, like traditional Sprite objects. However, the instances are linked to a single bitmap, which is not resampled when the display objects are transformed.

Rendering text objects

💡 *Use the bitmap caching feature and the `opaqueBackground` property to improve text rendering performance.*

The Flash Text Engine provides some great optimizations. However, numerous classes are required to show a single line of text. For this reason, creating an editable text field with the TextLine class requires a great deal of memory and many lines of ActionScript code. The TextLine class is best used for static and non-editable text, for which it renders faster and requires less memory.

The bitmap caching feature allows you to cache vector content as bitmaps to improve rendering performance. This feature is helpful for complex vector content and also when used with text content that requires processing to be rendered.

The following example shows how the bitmap caching feature and the `opaqueBackground` property can be used to improve rendering performance. The following figure illustrates a typical Welcome screen that can be displayed when a user is waiting for something to load:
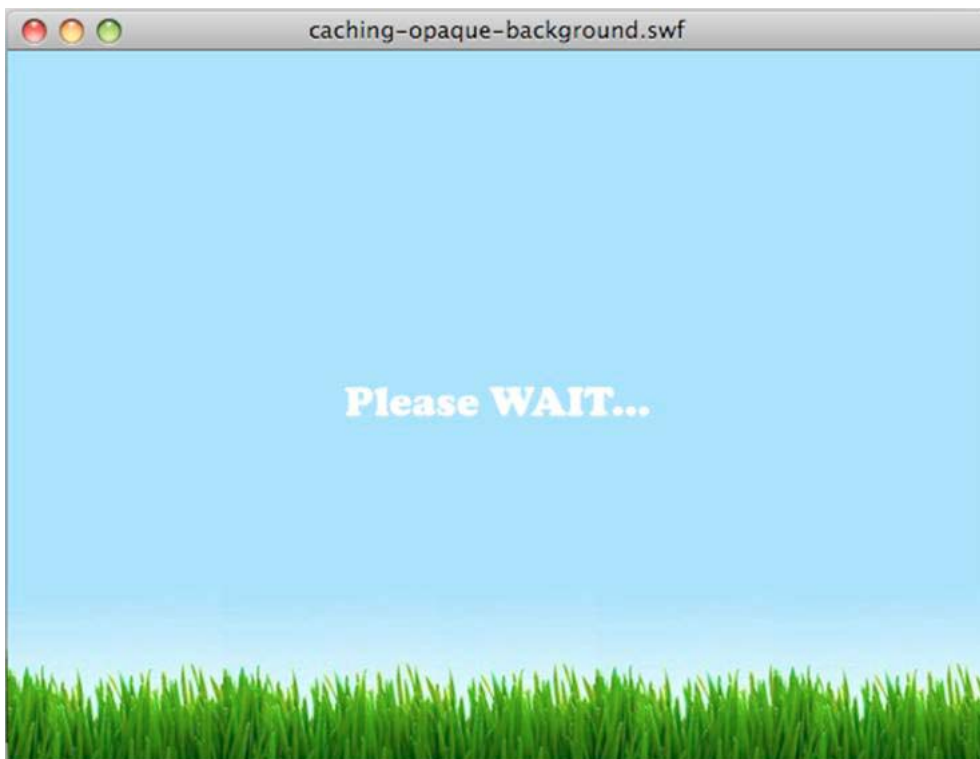
Welcome screen

The following figure illustrates the easing that is applied to the TextField object programmatically. The text eases slowly from the top of the scene to the center of the scene:
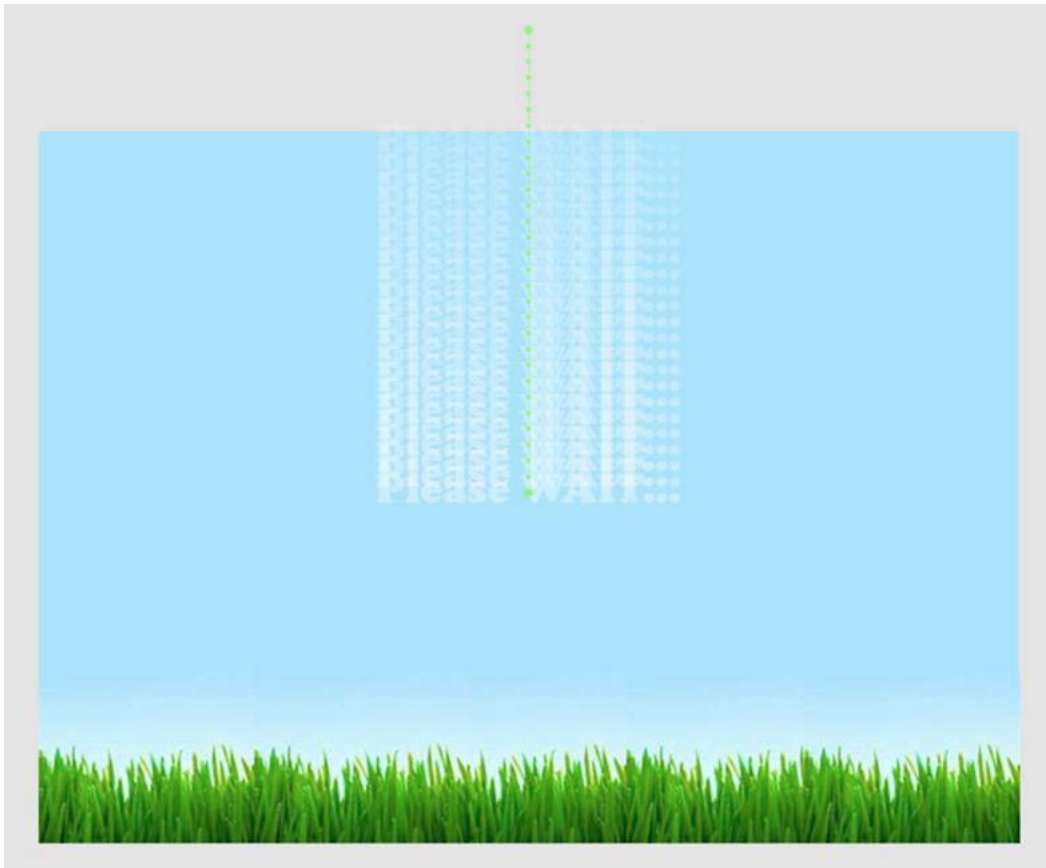
Easing of text

The following code creates the easing. The `preloader` variable stores the current target object to minimize property lookups, which can hurt performance:

```
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 );
}
```

The `Math.abs()` function could be moved inline here to reduce the number of function calls and get more performance improvements. A best practice is to use the int type for the `destX` and `destY` properties, so that you have fixed-point values. Using the int type allows you to get perfect pixel snapping without having to round values manually through slow methods like `Math.ceil()` or `Math.round()`. This code doesn't round the coordinates to int, because by rounding the values constantly, the object does not move smoothly. The object can have jittery movements, because the coordinates are snapped to the nearest rounded integers on every frame. However, this technique can be useful when setting a final position for a display object. Don't use the following code:

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

The following code is much faster:

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

The previous code can be optimized even further, by using bitwise-shifting operators to divide the values:

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

The bitmap caching feature makes it easier for the runtime to render objects by using dynamic bitmaps. In the current example, the movie clip containing the TextField object is cached:

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

One additional way to improve performance is by removing alpha transparency. Alpha transparency places an additional burden on the runtime when drawing transparent bitmap images, as in the previous code. You can use the `opaqueBackground` property to bypass that, by specifying a color as a background.

When using the `opaqueBackground` property, the bitmap surface created in memory still uses 32 bits. However, the alpha offset is set to 255 and no transparency is used. As a result, the `opaqueBackground` property does not reduce memory usage, but it improves rendering performance when using the bitmap caching feature. The following code contains all of the optimizations:

```
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 );
}
```

The animation is now optimized, and the bitmap caching has been optimized by removing transparency. On mobile devices, consider switching the Stage quality to `LOW` and `HIGH` during the different states of the animation while using the bitmap caching feature:

```
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 );
    }
}
```

However, in this case, changing the Stage quality forces the runtime to regenerate the bitmap surface of the TextField object to match the current Stage quality. For this reason, it is best to not change the Stage quality when using the bitmap caching feature.

A manual bitmap caching approach could have been used here. To simulate the `opaqueBackground` property, the movie clip can be drawn to a non-transparent BitmapData object, which does not force the runtime to regenerate the bitmap surface.

This technique works well for content that doesn't change over time. However, if the content of the text field can change, consider using a different strategy. For example, imagine a text field that is continuously updated with a percentage representing how much of the application has loaded. If the text field, or its containing display object, has been cached as a bitmap, its surface must be generated each time the content changes. You cannot use manual bitmap caching here, because the display object content is constantly changing. This constant change would force you to manually call the `BitmapData.draw()` method to update the cached bitmap.

Remember that since Flash Player 8 (and AIR 1.0), regardless of the Stage quality value, a text field with rendering set to Anti-Alias for Readability remains perfectly anti-aliased. This approach consumes less memory, but it requires more CPU processing and it renders a little more slowly than the bitmap caching feature.

The following code uses this approach:

```
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 );
    }
}
```

Using this option (Anti-Alias for Readability) for text in motion is not recommended. When scaling the text, this option causes the text tries to stay aligned, which produces a shifting effect. However, if the display object's content is constantly changing and you need scaled text, you can improve performance in mobile applications by setting the quality to LOW. When the motion finishes, switch the quality back to HIGH.

GPU

GPU rendering in Flash Player applications

An important new feature of Flash Player 10.1 is that it can use the GPU to render graphical content on mobile devices. In the past, graphics were rendered through the CPU only. Using the GPU optimizes the rendering of filters, bitmaps, video, and text. Keep in mind that GPU rendering is not always as accurate as software rendering. Content can look slightly chunky when using the hardware renderer. In addition, Flash Player 10.1 has a limitation that can prevent onscreen Pixel Bender effects from rendering. These effects can render as a black square when using hardware acceleration.

Although Flash Player 10 had a GPU acceleration feature, the GPU was not used to calculate the graphics. It was only used to send all the graphics to the screen. In Flash Player 10.1, the GPU is also used to calculate the graphics, which can improve rendering speed significantly. It also reduces the CPU workload, which is helpful on devices with limited resources, such as mobile devices.

GPU mode is set automatically when running content on mobile devices, for the best possible performance. Although wmode no longer must be set to gpu to get GPU rendering, setting wmode to opaque or transparent disables GPU acceleration.

Note: Flash Player on the desktop still uses the CPU to do software rendering. Software rendering is used because drivers vary widely on the desktop, and drivers can accentuate rendering differences. There can also be rendering differences between the desktop and some mobile devices.

GPU rendering in mobile AIR applications

Enable hardware graphics acceleration in an AIR application by including `<renderMode>gpu</renderMode>` in the application descriptor. You cannot change render modes at runtime. On desktop computers, the `renderMode` setting is ignored; GPU graphics acceleration is not currently supported.

GPU rendering mode limitations

The following limitations exist when using GPU rendering mode in AIR 2.5:

- If the GPU cannot render an object, it is not displayed at all. There is no fallback to CPU rendering.

- The following blend modes are not supported: layer, alpha, erase, overlay, hardlight, lighten, and darken.

- Filters are not supported.

- PixelBender is not supported.

- Many GPU units have a maximum texture size of 1024x1024. In ActionScript, this translates to the maximum final rendered size of a display object after all transformations.

- Adobe does not recommend the use of GPU rendering mode in AIR applications that play video.

- In GPU rendering mode, text fields are not always moved to a visible location when the virtual keyboard opens. To ensure that your text field is visible while the user enters text, do one of the following. Place the text field in the top half of the screen or move it to the top half of the screen when it receives focus.

- GPU rendering mode is disabled for some devices on which the mode does not work reliably. See the AIR developer release notes for the latest information.

GPU rendering mode best practices

The following guidelines can make GPU rendering faster:

- Limit the numbers of items visible on stage. Each item takes some time to render and composite with the other items around it. When you no longer want to display a display object, set its `visible` property to `false`. Do not simply move it off stage, hide it behind another object, or set its `alpha` property to 0. If the display object is no longer needed at all, remove it from the stage with `removeChild()`.

- Reuse objects rather than creating and destroying them.

- Make bitmaps in sizes that are close to, but less than, 2^n by 2^m bits. The dimensions do not have to be exact powers of 2, but they should be close to a power of 2, without being larger. For example, a 31-by-15-pixel image renders faster than a 33-by-17-pixel image. (31 and 15 are just less than powers of 2: 32 and 16.)

- If possible, set the repeat parameter to false when calling the `Graphic.beginBitmapFill()` method.

- Don't overdraw. Use the background color as a background. Don't layer large shapes on top of each other. There is a cost for every pixel that must be drawn.

- Avoid shapes with long thin spikes, self -intersecting edges, or lots of fine detail along the edges. These shapes take longer to render than display objects with smooth edges.

- Limit the size of display objects.

- Enable cacheAsBitMap and cacheAsBitmapMatrix for display objects whose graphics aren't updated frequently.

- Avoid using the ActionScript drawing API (the Graphics class) to create graphics. When possible, create those objects statically at authoring time instead.

- Scale bitmap assets to the final size before importing them.

GPU rendering mode in mobile AIR 2.0.3

GPU rendering is more restrictive in mobile AIR apps created with the Packager for iPhone. The GPU is only effective for bitmaps, solid shapes, and display objects that have the `cacheAsBitmap` property set. Also, for objects that have `cacheAsBitmap` and `cacheAsBitmapMatrix` set, the GPU can effectively render objects that rotate or scale. The GPU is used in tandem for other display objects and this generally results in poor rendering performance.

Tips for optimizing GPU rendering performance

Although GPU rendering can greatly improve performance of SWF content, the content's design plays an important role. Remember that settings that have historically worked well in software rendering sometimes do not work well with GPU rendering. The following tips can help you achieve good performance with GPU rendering without incurring any performance penalty in software rendering.

Note: Mobile devices that support hardware rendering often access SWF content from the web. Therefore, a best practice is to consider these tips when creating any SWF content to ensure the best experience across all screens.

- Avoid using `wmode=transparent` or `wmode=opaque` in HTML embed parameters. These modes can result in decreased performance. They can also result in a small loss in audio-video synchronization in both software and hardware rendering. Furthermore, many platforms do not support GPU rendering when these modes are in effect, significantly impairing performance.

- Use only the normal and alpha blend modes. Avoid using other blend modes, especially the layer blend mode. Not all blend modes can be reproduced faithfully when rendered with the GPU.

- When a GPU renders vector graphics, it breaks them up into meshes made of small triangles before drawing them. This process is called tessellating. Tessellating incurs a small performance cost, which increases as the complexity of the shape increases. To minimize performance impact, avoid morph shapes, which GPU rendering tessellates on every frame.

- Avoid self-intersecting curves, very thin curved regions (such as a thin crescent moon), and intricate details along the edges of a shape. These shapes are complex for the GPU to tessellate into triangle meshes. To understand why, consider two vectors: a 500 × 500 square and a 100 × 10 crescent moon. A GPU can easily render the large square because it's just two triangles. However, it takes many triangles to describe the curve of the crescent moon. Therefore, rendering the shape is more complicated even though it involves fewer pixels.

- Avoid large changes in scale, because such changes can also cause the GPU to again tessellate the graphics.

- Avoid overdrawing whenever possible. Overdrawing is layering multiple graphical elements so that they obscure each other. Using the software renderer, each pixel is drawn only once. Therefore, for software rendering, the application incurs no performance penalty regardless how many graphical elements are covering each other at that pixel location. By contrast, the hardware renderer draws each pixel for each element whether other elements obscure that region or not. If two rectangles overlap each other, the hardware renderer draws the overlapped region twice while the software renderer draws the region only once.

  Therefore, on the desktop, which use the software renderer, you typically do not notice a performance impact of overdraw. However, many overlapping shapes can adversely affect performance on devices using GPU rendering. A best practice is to remove objects from the display list rather than hiding them.

- Avoid using a large filled rectangle as a background. Set the background color of the Stage instead.

- Avoid the default bitmap fill mode of bitmap repeat whenever possible. Use bitmap clamp mode instead to achieve better performance.

Asynchronous operations

Favor using asynchronous versions of operations rather than synchronous ones, where available.

Synchronous operations run as soon as your code tells them to, and your code waits for them to complete before moving on. Consequently, they run in the application code phase of the frame loop. If a synchronous operation takes too long, it stretches the size of the frame loop, potentially causing the display to appear to freeze or stutter.

When your code executes an asynchronous operation, it doesn't necessarily run immediately. Your code and other application code in the current execution thread continues executing. The runtime then performs the operation as soon as possible, while attempting to prevent rendering issues. In some cases, the execution happens in the background and doesn't run as part of the runtime frame loop at all. Finally, when the operation completes the runtime dispatches an event, and your code can listen for that event to perform further work.

Asynchronous operations are scheduled and divided to avoid rendering issues. Consequently, it is much easier to have a responsive application using asynchronous versions of operations. See "Perceived performance versus actual performance" on page 2 for more information.

However, there is some overhead involved in running operations asynchronously. The actual execution time can be longer for asynchronous operations, especially for operations that take a short time to complete.

In the runtime, many operations are inherently synchronous or asynchronous and you can't choose how to execute them. However, in Adobe AIR, there are three types of operations that you can choose to perform synchronously or asynchronously:

- File and FileStream class operations

  Many operations of the File class can be performed synchronously or asynchronously. For example, the methods for copying or deleting a file or directory and listing the contents of a directory all have asynchronous versions. These methods have the suffix "Async" added to the name of the asynchronous version. For example, to delete a file asynchronously, call the `File.deleteFileAsync()` method instead of the `File.deleteFile()` method.

  When using a FileStream object to read from or write to a file, the way you open the FileStream object determines whether the operations execute asynchronously. Use the `FileStream.openAsync()` method for asynchronous operations. Data writing is performed asynchronously. Data reading is done in chunks, so the data is available one portion at a time. In contrast, in synchronous mode the FileStream object reads the entire file before continuing with code execution.

- Local SQL database operations

  When working with a local SQL database, all the operations executed through a SQLConnection object execute in either synchronous or asynchronous mode. To specify that operations execute asynchronously, open the connection to the database using the `SQLConnection.openAsync()` method instead of the `SQLConnection.open()` method. When database operations run asynchronously, they execute in the background. The database engine does not run in the runtime frame loop at all, so the database operations are much less likely to cause rendering issues.

  For additional strategies for improving performance with the local SQL database, see "SQL database performance" on page 85.

- Pixel Bender standalone shaders

The ShaderJob class allows you to run an image or set of data through a Pixel Bender shader and access the raw result data. By default, when you call the `ShaderJob.start()` method the shader executes asynchronously. The execution happens in the background, not using the runtime frame loop. To force the ShaderJob object to execute synchronously (which is not recommended), pass the value `true` to the first parameter of the `start()` method.

In addition to these built-in mechanisms for running code asynchronously, you can also structure your own code to run asynchronously instead of synchronously. If you are writing code to perform a potentially long-running task, you can structure your code so that it executes in parts. Breaking your code into parts allows the runtime to perform its rendering operations in between your code execution blocks, making rendering problems less likely.

Several techniques for dividing up your code are listed next. The main idea behind all these techniques is that your code is written to only perform part of its work at any one time. You track what the code does and where it stops working. You use a mechanism such as a Timer object to repeatedly check whether work remains and perform additional work in chunks until the work is complete.

There are a few established patterns for structuring code to divide up work in this way. The following articles and code libraries describe these patterns and provide code to help you implement them in your applications:

*   Asynchronous ActionScript Execution (article by Trevor McCauley with more background details as well as several implementation examples)

*   Parsing & Rendering Lots of Data in Flash Player (article by Jesse Warden with background details and examples of two approaches, the "builder pattern" and "green threads")

*   Green Threads (article by Drew Cummins describing the "green threads" technique with example source code)

*   greenthreads (open-source code library by Charlie Hubbard, for implementing "green threads" in ActionScript. See the greenthreads Quick Start for more information.)

*   Threads in ActionScript 3 at http://www.adobe.com/go/learn_fp_as3_threads_en (article by Alex Harui, including an example implementation of the "pseudo threading" technique)

Transparent windows

In AIR desktop applications, consider using an opaque rectangular application window instead of a transparent window.

To use an opaque window for the initial window of an AIR desktop application, set the following value in the application descriptor XML file:

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

For windows created by application code, create a NativeWindowInitOptions object with the `transparent` property set to `false` (the default). Pass it to the NativeWindow constructor while creating the NativeWindow object:

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

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

For a Flex Window component, make sure the component's transparent property is set to false, the default, before calling the Window object's `open()` method.

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

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

A transparent window potentially shows part of the user's desktop or other application windows behind the application window. Consequently, the runtime uses more resources to render a transparent window. A rectangular non-transparent window, whether it uses operating system chrome or custom chrome, does not have the same rendering burden.

Only use a transparent window when it is important to have a non-rectangular display or to have background content display through your application window.

Vector shape smoothing

Smooth shapes to improve rendering performance.

Unlike bitmaps, rendering vector content requires many calculations, especially for gradients and complex paths that contain many control points. As a designer or developer, make sure that shapes are optimized enough. The following figure illustrates non-simplified paths with many control points:

Non-optimized paths

By using the Smooth tool in Flash Professional, you can remove extra control points. An equivalent tool is available in Adobe® Illustrator®, and the total number of points and paths can be seen in the Document Info panel.

Smoothing removes extra control points, reducing the final size of the SWF file and improving rendering performance. The next figure illustrates the same paths after smoothing:

Optimized paths

As long as you do not oversimplify paths, this optimization does not change anything visually. However, the average frame rate of your final application can be greatly improved by simplifying complex paths.

Chapter 6: Optimizing network interaction

Enhancements for network interaction

Flash Player 10.1 and AIR 2.5 introduce a set of new features for network optimization on all platforms, including circular buffering and smart seeking.

Circular buffering

When loading media content on mobile devices, you can encounter problems that you would almost never expect on a desktop computer. For example, you are more likely to run out of disk space or memory. When loading video, the desktop versions of Flash Player 10.1 and AIR 2.5 download and cache the entire FLV file (or MP4 file) onto the hard drive. The runtime then plays back the video from that cache file. It is unusual for the disk space to run out. If such a situation occurs, the desktop runtime stops the playback of the video.

A mobile device can run out of disk space more easily. If the device runs out of disk space, the runtime does not stop the playback, as on the desktop runtime. Instead, the runtime starts to reuse the cache file by writing to it again from the beginning of the file. The user can continue watching the video. The user cannot seek in the area of the video that has been rewritten to, except to the beginning of the file. Circular buffering is not started by default. It can be started during playback, and also at the beginning of playback if the movie is bigger than the disk space or RAM. The runtime requires at least 4 MB of RAM or 20 MB of disk space to be able to use circular buffering.

Note: If the device has enough disk space, the mobile version of the runtime behaves the same as on the desktop. Keep in mind that a buffer in RAM is used as a fallback if the device does not have a disk or the disk is full. A limit for the size of the cache file and the RAM buffer can be set at compile time. Some MP4 files have a structure that requires the whole file is downloaded before the playback can start. The runtime detects those files and prevents the download, if there is not enough disk space, and the MP4 file cannot be played back. It can be best not to request the download of those files at all.

As a developer, remember that seeking only works within the boundary of the cached stream. `NetStream.seek()` sometimes fails if the offset is out of range, and in this case, a `NetStream.Seek.InvalidTime` event is dispatched.

Smart seeking

Note: The smart seeking feature requires Adobe® Flash® Media Server 3.5.3.

Flash Player 10.1and AIR 2.5 introduce a new behavior, called smart seeking, which improves the user's experience when playing streaming video. If the user seeks a destination inside the buffer bounds, the runtime reuses the buffer to offer instant seeking. In previous versions of the runtime, the buffer was not reused. For example, if a user was playing a video from a streaming server and the buffer time was set to 20 seconds (`NetStream.bufferTime`), and the user tried to seek 10 seconds ahead, the runtime would throw away all the buffer data instead of reusing the 10 seconds already loaded. This behavior forced the runtime to request new data from the server much more frequently and cause poor playback performance on slow connections.

The figure below illustrates how the buffer behaved in the previous releases of the runtime. The `bufferTime` property specifies the number of seconds to preload ahead so that if connection drops the buffer can be used without stopping the video:
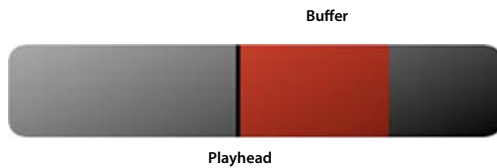
Buffer behavior before the smart seeking feature

With the smart seeking feature, the runtime now uses the buffer to provide instant backward or forward seeking when the user scrubs the video. The following figure illustrates the new behavior:

Forward seeking with the smart seeking feature

Backward seeking with the smart seeking feature

Smart seeking reuses the buffer when the user seeks forward or backward, so that playback experience is faster and smoother. One of the benefits of this new behavior is bandwidth savings for video publishers. However, if the seeking is outside the buffer limits, standard behavior occurs, and the runtime requests new data from the server.

*Note: This behavior does not apply to progressive video download.*

To use smart seeking, set `NetStream.inBufferSeek` to `true`.

External content

*Divide your application into multiple SWF files.*

Mobile devices can have limited access to the network. To load your content quickly, divide your application into multiple SWF files. Try to reuse code logic and assets through the entire application. For example, consider an application that has been divided into multiple SWF files, as shown in the following diagram:
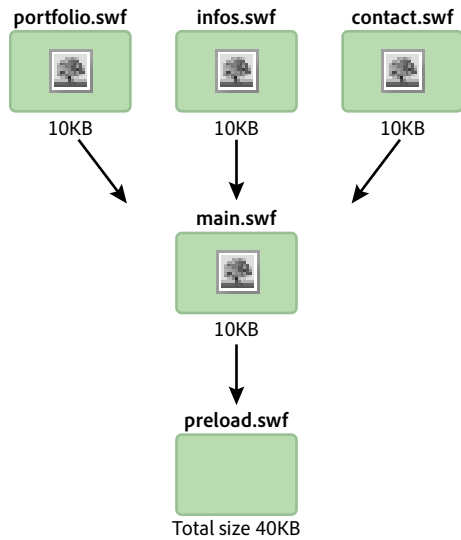
Application divided into multiple SWF files

In this example, each SWF file contains its own copy of the same bitmap. This duplication can be avoided by using a runtime shared library, as the following diagram illustrates:
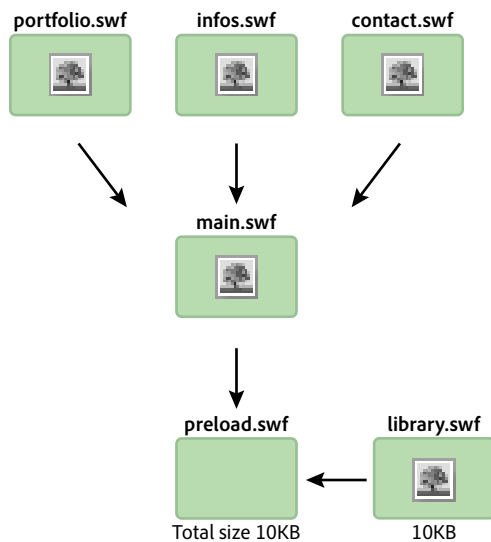
Using a runtime shared library

Using this technique, a runtime shared library is loaded to make the bitmap available to the other SWF files. The ApplicationDomain class stores all class definitions that have been loaded, and makes them available at runtime through the `getDefinition()` method.

A runtime shared library can also contain all the code logic. The entire application can be updated at runtime without recompiling. The following code loads a runtime shared library and extracts the definition contained in the SWF file at runtime. This technique can be used with fonts, bitmaps, sounds, or any ActionScript class:

```
// 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.");
}
```

Getting the definition can be made easier by loading the class definitions in the loading SWF file's application domain:

```
// 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.");
}
```

Now the classes available in the loaded SWF file can be used by calling the `getDefinition()` method on the current application domain. You can also access the classes by calling the `getDefinitionByName()` method. This technique saves bandwidth by loading fonts and large assets only once. Assets are never exported in any other SWF files. The only limitation is that the application has to be tested and run through the loader.swf file. This file loads the assets first, and then loads the different SWF files that compose the application.

Input output errors

Provide event handlers and error messages for IO errors.

On a mobile device, the network can be less reliable as on a desktop computer connected to high-speed Internet. Accessing external content on mobile devices has two constraints: availability and speed. Therefore, make sure that assets are lightweight and add handlers for every IO_ERROR event to provide feedback to the user.

For example, imagine a user is browsing your website on a mobile device and suddenly loses the network connection between two metro stations. A dynamic asset was being loaded when the connection is lost. On the desktop, you can use an empty event listener to prevent a runtime error from showing, because this scenario would almost never happen. However, on a mobile device you must handle the situation with more than just a simple empty listener.

The following code does not respond to an IO error. Do not use it as it is shown:

```
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;
}
```

A better practice is to handle such a failure and provide an error message for the user. The following code handles it properly:

```
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
}
```

As a best practice, remember to offer a way for the user to load the content again. This behavior can be implemented in the `onIOError()` handler.
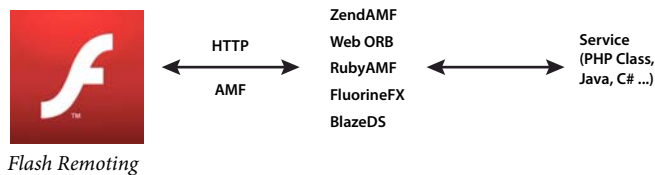
Flash Remoting

Use Flash Remoting and AMF for optimized client-server data communication.

You can use XML to load remote content into SWF files. However, XML is plain text that the runtime loads and parses. XML works best for applications that load a limited amount of content. If you are developing an application that loads a large amount of content, consider using the Flash Remoting technology and Action Message Format (AMF).

AMF is a binary format used to share data between a server and the runtime. Using AMF reduces the size of the data and improves the speed of transmission. Because AMF is a native format for the runtime, sending AMF data to the runtime avoids memory-intensive serialization and deserialization on the client side. The remoting gateway handles these tasks. When sending an ActionScript data type to a server, the remoting gateway handles the serialization for you on the server side. The gateway also sends you the corresponding data type. This data type is a class created on the server that exposes a set of methods that can be called from the runtime. Flash Remoting gateways include ZendAMF, FluorineFX, WebORB, and BlazeDS, an official open-source Java Flash Remoting gateway from Adobe.

The following figure illustrates the concept of Flash Remoting:

Flash Remoting

The following example uses the NetConnection class to connect to a Flash Remoting gateway:

```
// 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 ?");
```

Connecting to a remoting gateway is straightforward. However, using Flash Remoting can be made even easier by using the RemoteObject class included in the Adobe® Flex® SDK.

Note: External SWC files, such as the ones from the Flex framework, can be used inside an Adobe® Flash® Professional project. Using SWC files allows you to use the RemoteObject class and its dependencies without using the rest of the Flex SDK. Advanced developers can even communicate with a remoting gateway directly through the raw Socket class, if necessary.

Unnecessary network operations

Cache assets locally after loading them, instead of loading them from the network each time they're needed.

If your application loads assets such as media or data, cache the assets by saving them to the local device. For assets that change infrequently, consider updating the cache at intervals. For example, your application could check for a new version of an image file once per day, or check for fresh data once every two hours.

You can cache assets in several ways, depending on the type and nature of the asset:

* Media assets such as images and video: save the files to the file system using the File and FileStream classes

* Individual data values or small sets of data: save the values as local shared objects using the SharedObject class

* Larger sets of data: save the data in a local database, or serialize the data and save it to a file

For caching data values, the open-source AS3CoreLib project includes a ResourceCache class that does the loading and caching work for you.

Chapter 7: Working with media

Video

For information about optimizing the performance of video on mobile devices, see Optimize web content for mobile delivery on the Adobe Developer Connection web site.

In particular, see the sections called:

- *Playing video on mobile devices*
- *Code samples*

These sections contain information for developing video players for mobile devices, such as:

- Video encoding guidelines
- Best practices
- How to profile the video player's performance
- A reference video player implementation

StageVideo

Use the StageVideo class to take advantage of hardware acceleration to present video.

For information about using the StageVideo object, see Using the StageVideo class for hardware accelerated presentation in the ActionScript 3.0 Developer's Guide.

Audio

Starting with Flash Player 9.0.115.0 and AIR 1.0, the runtime can play AAC files (AAC Main, AAC LC, and SBR). A simple optimization can be made by using AAC files instead of mp3 files. The AAC format offers better quality and smaller file size than the mp3 format at an equivalent bitrate. Reducing file size saves bandwidth, which is an important factor on mobile devices that don't offer high-speed Internet connections.

Hardware Audio Decoding

Similar to video decoding, audio decoding requires high CPU cycles and can be optimized by leveraging available hardware on the device. Flash Player 10.1 and AIR 2.5 can detect and use hardware audio drivers to improve performance when decoding AAC files (LC, HE/SBR profiles) or mp3 files (PCM is not supported). CPU usage is reduced dramatically, which results in less battery usage and makes the CPU available for other operations.

Note: When using the AAC format, the AAC Main profile is not supported on devices due to the lack of hardware support on most devices.

Hardware audio decoding is transparent to the user and developer. When the runtime starts playing audio streams, it checks the hardware first, as it does with video. If a hardware driver is available and the audio format is supported, hardware audio decoding takes place. However, even if the incoming AAC or mp3 stream decoding can be handled through the hardware, sometimes the hardware cannot process all effects. For example, sometimes the hardware does not process audio mixing and resampling, depending on hardware limitations.

Chapter 8: SQL database performance

Application design for database performance

Don't change a SQLStatement object's `text` property after executing it. Instead, use one SQLStatement instance for each SQL statement and use statement parameters to provide different values.

Before any SQL statement is executed, the runtime prepares (compiles) it to determine the steps that are performed internally to carry out the statement. When you call `SQLStatement.execute()` on a SQLStatement instance that hasn't executed previously, the statement is automatically prepared before it is executed. On subsequent calls to the `execute()` method, as long as the `SQLStatement.text` property hasn't changed the statement is still prepared. Consequently, it executes faster.

To gain the maximum benefit from reusing statements, if values change between statement executions, use statement parameters to customize your statement. (Statement parameters are specified using the `SQLStatement.parameters` associative array property.) Unlike changing the SQLStatement instance's `text` property, if you change the values of statement parameters the runtime isn't required to prepare the statement again.

When you're reusing a SQLStatement instance, your application must store a reference to the SQLStatement instance once it has been prepared. To keep a reference to the instance, declare the variable as a class-scope variable rather than a function-scope variable. One good way to make the SQLStatement a class-scope variable is to structure your application so that a SQL statement is wrapped in a single class. A group of statements that are executed in combination can also be wrapped in a single class. (This technique is known as using the Command design pattern.) By defining the instances as member variables of the class, they persist as long as the instance of the wrapper class exists in the application. At a minimum, you can simply define a variable containing the SQLStatement instance outside a function so that the instance persists in memory. For example, declare the SQLStatement instance as a member variable in an ActionScript class or as a non-function variable in a JavaScript file. You can then set the statement's parameter values and call its `execute()` method when you want to actually run the query.

Use database indexes to improve execution speed for data comparing and sorting.

When you create an index for a column, the database stores a copy of that column's data. The copy is kept sorted in numeric or alphabetical order. The sorting allows the database to quickly match values (such as when using the equality operator) and sort result data using the `ORDER BY` clause.

Database indexes are kept continuously up-to-date, which causes data change operations (INSERT or UPDATE) on that table to be slightly slower. However, the increase in data retrieval speed can be significant. Because of this performance tradeoff, don't simply index every column of every table. Instead, use a strategy for defining your indexes. Use the following guidelines to plan your indexing strategy:

• Index columns that are used in joining tables, in WHERE clauses, or ORDER BY clauses

• If columns are frequently used together, index them together in a single index

• For a column that contains text data that you retrieve sorted alphabetically, specify COLLATE NOCASE collation for the index

Consider pre-compiling SQL statements during application idle times.

The first time a SQL statement executes, it is slower because the SQL text is prepared (compiled) by the database engine. Because preparing and executing a statement can be demanding, one strategy is to preload initial data and then execute other statements in the background:

1 Load the data that the application needs first.

2 When the initial startup operations of your application have completed, or at another "idle" time in the application, execute other statements.

For example, suppose your application doesn't access the database at all to display its initial screen. In that case, wait until that screen displays before opening the database connection. Finally, create the SQLStatement instances and execute any that you can.

Alternatively, suppose that when your application starts up it immediately displays some data, such as the result of a particular query. In that case, go ahead and execute the SQLStatement instance for that query. After the initial data is loaded and displayed, create SQLStatement instances for other database operations and if possible execute other statements that are needed later.

In practice, if you are reusing SQLStatement instances, the additional time required to prepare the statement is only a one-time cost. It probably doesn't have a large impact on overall performance.

🔆 *Group multiple SQL data change operations in a transaction.*

Suppose you're executing many SQL statements that involve adding or changing data (`INSERT` or `UPDATE` statements). You can get a significant increase in performance by executing all the statements within an explicit transaction. If you don't explicitly begin a transaction, each of the statements runs in its own automatically created transaction. After each transaction (each statement) finishes executing, the runtime writes the resulting data to the database file on the disk.

On the other hand, consider what happens if you explicitly create a transaction and execute the statements in the context of that transaction. The runtime makes all the changes in memory, then writes all the changes to the database file at one time when the transaction is committed. Writing the data to disk is usually the most time-intensive part of the operation. Consequently, writing to the disk one time rather than once per SQL statement can improve performance significantly.

🔆 *Process large `SELECT` query results in parts using the SQLStatement class's `execute()` method (with `prefetch` parameter) and `next()` method.*

Suppose you execute a SQL statement that retrieves a large result set. The application then processes each row of data in a loop. For example, it formats the data or creates objects from it. Processing that data can take a large amount of time, which could cause rendering problems such as a frozen or non-responsive screen. As described in "Asynchronous operations" on page 71, one solution is to divide up the work into chunks. The SQL database API makes dividing up data processing easy to do.

The SQLStatement class's `execute()` method has an optional `prefetch` parameter (the first parameter). If you provide a value, it specifies the maximum number of result rows that the database returns when the execution completes:

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

Once the first set of result data returns, you can call the `next()` method to continue executing the statement and retrieve another set of result rows. Like the `execute()` method, the `next()` method accepts a `prefetch` parameter to specify a maximum number of rows to return:

```
// 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);
        }
    }
}
```

You can continue calling the `next()` method until all the data loads. As shown in the previous listing, you can determine when the data has all loaded. Check the `complete` property of the SQLResult object that's created each time the `execute()` or `next()` method finishes.

Note: Use the `prefetch` parameter and the `next()` method to divide up the processing of result data. Don't use this parameter and method to limit a query's results to a portion of its result set. If you only want to retrieve a subset of rows in a statement's result set, use the `LIMIT` clause of the `SELECT` statement. If the result set is large, you can still use the `prefetch` parameter and `next()` method to divide up the processing of the results.

Consider using multiple asynchronous SQLConnection objects with a single database to execute multiple statements simultaneously.

When a SQLConnection object is connected to a database using the `openAsync()` method, it runs in the background rather than the main runtime execution thread. In addition, each SQLConnection runs in its own background thread. By using multiple SQLConnection objects, you can effectively run multiple SQL statements simultaneously.

There are also potential downsides to this approach. Most importantly, each additional SQLStatement object requires additional memory. In addition, simultaneous executions also cause more work for the processor, especially on machines that only have one CPU or CPU core. Because of these concerns, this approach is not recommended for use on mobile devices.

An additional concern is that the potential benefit from reusing SQLStatement objects can be lost because a SQLStatement object is linked to a single SQLConnection object. Consequently, the SQLStatement object can't be reused if its associated SQLConnection object is already in use.

If you choose to use multiple SQLConnection objects connected to a single database, keep in mind that each one executes its statements in its own transaction. Be sure to account for these separate transactions in any code that changes data, such as adding, modifying, or deleting data.

Paul Robertson has created an open-source code library that helps you incorporate the benefits of using multiple SQLConnection objects while minimizing the potential downsides. The library uses a pool of SQLConnection objects and manages the associated SQLStatement objects. In this way it ensures that SQLStatement objects are reused, and multiple SQLConnection objects are available to execute multiple statements simultaneously. For more information and to download the library, visit http://probertson.com/projects/air-sqlite/.

Database file optimization

💡 *Avoid database schema changes.*

If possible, avoid changing the schema (table structure) of a database once you've added data into the database's tables. Normally a database file is structured with the table definitions at the start of the file. When you open a connection to a database, the runtime loads those definitions. When you add data to database tables, that data is added to the file after the table definition data. However, if you make schema changes, the new table definition data is mixed in with the table data in the database file. For example, adding a column to a table or adding a new table can result in the mixing of types of data. If the table definition data is not all located at the beginning of the database file, it takes longer to open a connection to the database. The connection is slower to open because it takes the runtime longer to read the table definition data from different parts of the file.

💡 *Use the `SQLConnection.compact()` method to optimize a database after schema changes.*

If you must make schema changes, you can call the `SQLConnection.compact()` method after completing the changes. This operation restructures the database file so that the table definition data is located together at the start of the file. However, the `compact()` operation can be time-intensive, especially as a database file grows larger.

Unnecessary database run-time processing

💡 *Use a fully qualified table name (including database name) in your SQL statement.*

Always explicitly specify the database name along with each table name in a statement. (Use "main" if it's the main database). For example, the following code includes an explicit database name `main`:

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

Explicitly specifying the database name prevents the runtime from having to check each connected database to find the matching table. It also prevents the possibility of having the runtime choose the wrong database. Follow this rule even if a SQLConnection is only connected to a single database. Behind the scenes, the SQLConnection is also connected to a temporary database that is accessible through SQL statements.

💡 *Use explicit column names in SQL `INSERT` and `SELECT` statements.*

The following examples show the use of explicit column names:

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

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

Compare the preceding examples to the following ones. Avoid this style of code:

```
-- 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
```

Without explicit column names, the runtime has to do extra work to figure out the column names. If a `SELECT` statement uses a wildcard rather than explicit columns, it causes the runtime to retrieve extra data. This extra data requires extra processing and creates extra object instances that aren't needed.

Avoid joining the same table multiple times in a statement, unless you are comparing the table to itself.

As SQL statements grow large, you can unintentionally join a database table to the query multiple times. Often, the same result could be achieved by using the table only once. Joining the same table multiple times is likely to happen if you are using one or more views in a query. For example, you could be joining a table to the query, and also a view that includes the data from that table. The two operations would cause more than one join.

Efficient SQL syntax

Use `JOIN` (in the `FROM` clause) to include a table in a query instead of a subquery in the `WHERE` clause. This tip applies even if you only need a table's data for filtering, not for the result set.

Joining multiple tables in the `FROM` clause performs better than using a subquery in a `WHERE` clause.

Avoid SQL statements that can't take advantage of indexes. These statements include the use of aggregate functions in a subquery, a `UNION` statement in a subquery, or an `ORDER BY` clause with a `UNION` statement.

An index can greatly increase the speed of processing a `SELECT` query. However, certain SQL syntax prevents the database from using indexes, forcing it to use the actual data for searching or sorting operations.

Consider avoiding the `LIKE` operator, especially with a leading wildcard character as in `LIKE('%XXXX%')`.

Because the `LIKE` operation supports the use of wildcard searches, it performs slower than using exact-match comparisons. In particular, if you start the search string with a wildcard character, the database can't use indexes at all in the search. Instead, the database must search the full text of each row of the table.

Consider avoiding the `IN` operator. If the possible values are known beforehand, the `IN` operation can be written using `AND` or `OR` for faster execution.

The second of the following two statements executes faster. It is faster because it uses simple equality expressions combined with `OR` instead of using the `IN()` or `NOT IN()` statements:

```
-- 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
```

Consider alternative forms of a SQL statement to improve performance.

As demonstrated by previous examples, the way a SQL statement is written can also affect database performance. There are often multiple ways to write a SQL SELECT statement to retrieve a particular result set. In some cases, one approach runs notably faster than another one. In addition to the preceding suggestions, you can learn more about different SQL statements and their performance from dedicated resources on the SQL language.

SQL statement performance

Directly compare alternate SQL statements to determine which is faster.

The best way to compare the performance of multiple versions of a SQL statement is to test them directly with your database and data.

The following development tools provide execution times when running SQL statements. Use them to compare the speed of alternative versions of statements:

• Run! (AIR SQL query authoring and testing tool, by Paul Robertson)

• Lita (SQLite Administration Tool, by David Deraedt)

Chapter 9: Benchmarking and deploying

Benchmarking

There are a number of tools available for benchmarking applications. You can use the Stats class and the PerformanceTest class, developed by Flash community members. You can also use the profiler in Adobe® Flash® Builder™, and the FlexPMD tool.

The Stats class

To profile your code at runtime using the release version of the runtime, without an external tool, you can use the Stats class developed by mr. doob from the Flash community. You can download the Stats class at the following address: https://github.com/mrdoob/Hi-ReS-Stats.

The Stats class allows you to track the following things:

- Frames rendered per second (the higher the number, the better).
- Milliseconds used to render a frame (the lower number, the better).
- The amount of memory the code is using. If it increases on each frame, it is possible that your application has a memory leak. It is important to investigate the possible memory leak.
- The maximum amount of memory the application used.

Once downloaded, the Stats class can be used with the following compact code:

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

By using conditional compilation in Adobe® Flash® Professional or Flash Builder, you can enable the Stats object:

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

By switching the value of the DEBUG constant, you can enable or disable the compilation of the Stats object. The same approach can be used to replace any code logic that you do not want to be compiled in your application.

The PerformanceTest class

To profile ActionScript code execution, Grant Skinner has developed a tool that can be integrated into a unit testing workflow. You pass a custom class to the PerformanceTest class, which performs a series of tests on your code. The PerformanceTest class allows you to benchmark different approaches easily. The PerformanceTest class can be downloaded at the following address: http://www.gskinner.com/blog/archives/2009/04/as3_performance.html.

Flash Builder profiler

Flash Builder is shipped with a profiler that allows you to benchmark your code with a high level of detail.

Note: Use the debugger version of Flash Player to access the profiler, or you'll get an error message.

The profiler can also be used with content produced in Adobe Flash Professional. To do that, load the compiled SWF file from an ActionScript or Flex project into Flash Builder, and you can run the profiler on it. For more information on the profiler, see "Profiling Flex applications" in Using Flash Builder 4.

FlexPMD

Adobe Technical Services has released a tool called FlexPMD, which allows you to audit the quality of ActionScript 3.0 code. FlexPMD is an ActionScript tool, which is similar to JavaPMD. FlexPMD improves code quality by auditing an ActionScript 3.0 or Flex source directory. It detects poor coding practices, such as unused code, overly complex code, overly lengthy code, and incorrect use of the Flex component life cycle.

FlexPMD is an Adobe open-source project available at the following address: http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD. An Eclipse plug-in is also available at the following address: http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD+Eclipse+plugin.

FlexPMD makes it easier to audit code and to make sure that your code is clean and optimized. The real power of FlexPMD lies in its extensibility. As a developer, you can create your own sets of rules to audit any code. For example, you can create a set of rules that detect heavy use of filters, or any other poor coding practice that you want to catch.

Deploying

When exporting the final version of your application in Flash Builder, it is important to make sure that you export the release version of it. Exporting a release version removes the debugging information contained in the SWF file. Removing the debugging information makes the SWF file size smaller and helps the application to run faster.

To export the release version of your project, use the Project panel in Flash Builder and the Export Release Build option.

Note: When compiling your project in Flash Professional, you do not have the option of choosing between release and debugging version. The compiled SWF file is a release version by default.