Developers guidelines

DEVELOPER WORLD THE FAST TRACK FROM MIND TO MARKET

October 2009

Adobe™ Flash Lite™ 2 and 3

in Sony Ericsson feature phones

Sony Ericsson

Preface

Purpose of this document

This document is intended for Adobe™ Flash Lite™ content developers who want insight in the implementation of Flash Lite in Sony Ericsson phones.

People who can benefit from this document are:

- Software developers
- Operators and service providers
- Content providers

It is assumed that the reader is familiar with Flash development and ActionScript.

At Sony Ericsson Developer World, corresponding information for Flash Lite version 1.1 enabled phones can be found in a separate Developers guideline.

These Developers guidelines are published by:

Sony Ericsson Mobile Communications AB,
SE-221 88 Lund, Sweden

Phone: +46 46 19 40 00
Fax: +46 46 19 41 00
www.sonyericsson.com/

12th edition (October 2009)
Publication number: 1203-8107.12

Sony Ericsson Developer World

At www.sonyericsson.com/developer, developers find the latest technical documentation and development tools such as phone White papers, Developers guidelines for different technologies, Getting started tutorials, SDKs (Software Development Kits) and tool plugins. The Web site also features news articles, go-to-market advice, moderated discussion forums offering free technical support and a Wiki community sharing expertise and code examples.

For more information about these professional services, go to the Sony Ericsson Developer World Web site.

Document conventions

Products

The following Sony Ericsson phones are referred to in this document by generic names:

Generic names Series	**Sony Ericsson phones**	**Flash Lite version**
Aino™	Aino™ U10i, Aino™ U10a	3
C510	C510, C510c, C510a	2
C702	C702, C702c, C702a	2
C901	C901, C901a, C901 GreenHeart™	2
C902	C902, C902c	2
C903	C903, C903a	2
C905	C905, C905c, C905a	2
G502	G502, G502c	2
G705	G705, G705u	2
Jalou™	Jalou™ F100i, BeJoo™ F100i	2
K630	K630i	2
K660	K660i	2
K850	K850i, K858c	2
Naite™	Naite™ J105i, Naite™ J105a	2
T700	T700	2

Generic names Series	**Sony Ericsson phones**	**Flash Lite version**
T707	T707, T707a	2
T715	T715, T715a	2
V640	V640i	2
W508	W508, W508c, W508a, W518a	2
W595	W595, W595s	2
W705	W705, W705u	2
W715	W715	2
W760	W760i, W760c	2
W890	W890i	2
W902	W902	2
W910	W910i, W908c	2
W980	W980i	2
Pureness™	Xperia™ Pureness™ X5, Xperia™ Pureness™ X5i	3
Yari™	Yari™ U100i, Yari™ U100a	3
Z750	Z750i	2
Z770	Z770i	2
Z780	Z780i, Z780a	2

Trademarks and acknowledgements

Document history

Change history		
2007-09-24	Doc. no. 1203-8107.1	First version published on Developer World. Information about K850, V640, W910 and Z750 series
2007-11-06	Doc. no. 1203-8107.2	Second version. Information about K630, K660 and W890 series added
2008-01-06	Doc. no. 1203-8107.3	Third version. Information about W760 series added
2008-02-20	Doc. no. 1203-8107.4	Fourth version. Information about C702, C902, W980 and Z770 series added
2008-05-20	Doc. no. 1203-8107.5	Fifth version. Information about G502 and Z780 series added
2008-07-22	Doc. no. 1203-8107.6	Sixth version. Information about C905, T700, W595 and W902 series added
2008-08-31	Doc. no. 1203-8107.7	Seventh version. Optimising tips added in appendix
2008-09-09	Doc. no. 1203-8107.8	Eighth version. Information about G705 added
2008-10-17	Doc. no. 1203-8107.8 (rev. B)	Eighth revised version. New layout
2009-02-15	Doc. no. 1203-8107.9	Ninth version. Information about C510, C901, C903, W508, W705, W715 and W995 added
2009-03-26	Doc. no. 1203-8107.10	Tenth version. Information about T707 added
2009-04-08	Doc. no. 1203-8107.10 (rev. B)	Tenth, revised version.
2009-08-12	Doc. no. 1203-8107.11	Eleventh version. Information about Aino™, Naite™, T715 and Yari™ added
2009-10-26	Doc. no. 1203-8107.12	12th version. Information about Pureness™ phones added

Contents

Overview

Flash Lite was originally developed by Macromedia™ (now part of Adobe™ Systems Inc.) to run Flash™ based content on the latest generation of mobile phones and other small devices. It first appeared as Macromedia Flash Lite 1.0 in 2003, and was at that time aimed at the Japanese NTT DoCoMo i-mode™ services. Flash Lite 1.1 followed in 2004, and contained a number of major improvements. Both Flash Lite 1.0 and 1.1 are based on a Flash 4 structure, helping to minimise the footprint and processor demands of the player.

In 2006 Adobe Flash Lite 2.0 was introduced. It is based on the Flash Player 7 and supports most of its features, and also have some additional features suitable for mobile development. Part of the ActionScript 2.0 command set is supported. For further details about Flash development in general and the Flash Lite player in particular, see Adobe Mobile & Devices Developer Center found at http://www.adobe.com/devnet/devices/.

Flash Lite in Sony Ericsson phones

There are two main implementations in Flash Lite enabled Sony Ericsson phones:

- The “stand alone” model. Flash Lite exists as a separate media application in the phone. In some Sony Ericsson phones, this model is used for wallpaper or screensaver applications.
- The “browser based” model. Flash Lite essentially runs as a plugin within the resident browser of the phone. This model allows a user to access Flash Lite content via an Internet connection without needing to open a separate application for viewing. It is also the default way of viewing Flash content via the phone file manager. This model is implemented in all Flash-enabled Sony Ericsson phones.

Specifications

Compliancy

Sony Ericsson phones in this document comply with the Flash Lite 2 or Flash Lite 3 standards, with the exception of the following areas which are currently not supported:

- Device video playback
- Streaming sound (MP3, AAC, and so on).

Phone features

Features	Phones		
	K630, V640	**C510, C702, C901, C902, C903, C905, G502, G705, Jalou™, K630, K660, K850, Naite™, T700, T707, T715, W508, W595, W705, W715, W760, W890, W902, W910, W980, W995, Z750, Z770, Z780**	**Aino™, Pureness™, Yari™**
General			
Screen size	176x220	240x320	240x320 Aino™: 240x432
Colour depth	18-bit (262,144 colours)		Aino™: 24-bit (16M colours), Pureness™: 64 shades of grey, Yari™: 18-bit (262,144 colours)
Web browser	Access Netfront, version 3.4		Access Netfront, version 3.5
Browser screen, normal mode	176x176	240x266	240x266 Aino™: 240x378

Features	Phones		
	K630, V640	**C510, C702, C901, C902, C903, C905, G502, G705, Jalou™, K630, K660, K850, Naite™, T700, T707, T715, W508, W595, W705, W715, W760, W890, W902, W910, W980, W995, Z750, Z770, Z780**	**Aino™, Pureness™, Yari™**
Browser, full screen mode	176x220	240x320	240x320 Aino™: 240x432
Flash features			
Flash Lite version	2.x		3.x
Flash in browser	Yes		
Flash in Wallpaper	Yes		
Flash in Screensaver	Yes		
Flash colour depth	16-bit (65,536 colours)		
Heap size for Flash, per content type (MB)	2		
Enabled sound formats for Flash	Midi		
Language support for device fonts and input for Flash Lite 1.1 content	According to i-mode specifications		

Links and references

- Adobe Flash Lite product page
  http://www.adobe.com/products/flashlite/
- Adobe Mobile and Devices Developer Center
  http://www.adobe.com/devnet/devices/flashlite.html
- Sony Ericsson Developer World
  http://developer.sonyericsson.com

Appendix

This appendix contains general advice on application design and programming issues of interest for Flash Lite 2 and Flash Lite 3 application developers, targeting Sony Ericsson phones.

General design considerations

Designing Flash applications for phones presents several challenges:

- **Screen size**. The small dimensions (128x160, 176x220, 240x320 or 240x432 pixels – smaller when the browser is in normal mode) of phone screens should not limit application capabilities, but is a key consideration in application design, particularly when it comes to selecting fonts and images.

- **Navigation**. In most phones, navigation has to be done using keypad controls as opposed to mouse/keyboard navigation in, for example, a computer. On Sony Ericsson phones, some keys are assigned for specific use, for example, “Back” and “Select”. This may need to be considered when designing Flash Lite applications, to avoid usability problems.

- **Menus**. To improve usability, Flash applications should adapt to MMI (Man – Machine Interface) conventions established in targeted phone models.

- **Image formats and sizes**. The limited processor power in a phone requires optimisation of images before using them in Flash Lite applications. In general, phones also have limited colour range compared to computers.

- **Softkeys**. Newer Sony Ericsson phones support up to three softkeys, however Flash Lite is only able to control two of them as the third softkey is mapped to the centre navigation key, the Key.ENTER event. Softkeys can only be mapped when a flash file is played in interactive mode in the browser, not within a web page.

- **Device decoded images**. Flash Lite 2 and 3 offer the possibility to let the native image decoder of the device handle images. To determine what image formats a phone supports, the property `System.capabilities.imageMIMETypes` can be used. Decoding megapixel images is currently too much for the Flash Lite decoder to handle.

- **Network access** from Flash Lite is only permitted for Flash files running in the browser plug-in.

- **Scaling**. Flash Lite is a vector based format and scales very well to different sizes. However, different viewers in the phone scale them differently.

  - The browser scales a stand-alone .SWF file to fit its drawing area. For example, a browser drawing area of 240x240 pixels shows all Flash Lite content in that size, even if the original size is 128x128.
  - Flash files within an HTML document adapt to HTML “Size” attribute values if specified.
  - Screensaver or wallpaper viewers present Flash content in its original size if it is smaller or equal to the screen size. If it is larger it is scaled down to the fit the screen size.

  noScale makes the size of Flash content fixed, so that it remains unchanged even as the size of the player window changes. Cropping may occur if the player window is smaller than the Flash content.

- The documentation provided by Adobe gives examples of existing Flash based content, and provides information on how to design Flash Lite applications for phones. Testing applications on actual phones to verify application behaviour on the targeted platform, is also highly recommended.

Optimisation guidelines

The limited processing power, number of colours, display area and memory of mobile phones require optimisation of Flash Lite applications designed or adapted for this platform. For example, applications must be designed to use memory and the limited CPU power efficiently and lead to small SWF file sizes.

The Zip file included with this document contains a Flash file, with a movie clip which might be useful for monitoring the level of free memory, frame rates per second and other benchmarking values while testing an application during development. The movie clips and associated code can be copied and pasted into a Flash Lite project during development.

Memory management

The Flash Lite player assigns memory for rendering objects such us movie clips, buttons, bitmaps, sound, XML, and so on. At start-up Flash Lite reserves a fixed portion of 256 KB for the static heap and a maximum of 2 MB for the dynamic heap. Project Capuchin applications running in phones with Flash Lite 3 reserve up to 10 MB dynamic heap. Memory is first allocated from the static heap and, if necessary, memory is allocated from the dynamic heap in 32 KB chunks.

When the Flash player is set to run in the background or to pause, only a few chunks of the dynamic heap are released to the system, while the static heap is never freed. When the Flash player runs out of memory the screen most often turns white.

Image rendering optimisation

Vector images vs bitmaps

PNG8 is the preferred image format for multicoloured graphics, instead of vector or traced graphics. PNG8 tends to yield relatively small file sizes and is ideal when enough memory is available. Compressed JPG images may be smaller in size, but the decompression process uses CPU power, which in turn may affect the perceived image quality. PNG8 files are about five times smaller than PNG32 files. Reducing the number of colours in the palette as much as possible reduces the file size. But sometimes PNG files render in a slightly lower quality on 16-bit colour (RGB565) displays. This problem can be reduced by applying some dithering effects to reduce the distortion especially when using gradient colours.

With regard to vector shapes, it is important that they are primitive and do not have too many points or curves in order to reduce the calculations needed by the CPU. In Adobe Flash authoring environments, this can be done by selecting the shape and then selecting *Modify – Shape – Optimize* from the main menu.

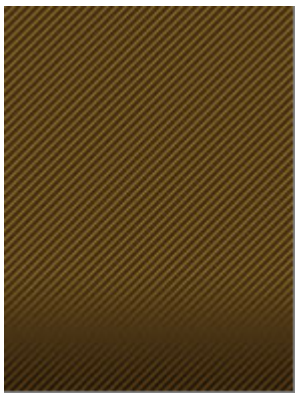
Gradient vector shapes are CPU intensive and should be used with care. Also note that linear gradients renders faster than radial gradients.

Compressing bitmaps

Bitmaps are produced in the form of compressed files, such as PNGs, JPEGs or GIFs. Keep in mind that these formats occupy more memory space than the actual file size when decompressed by the image decoder.

Bitmaps should be compressed to the lowest possible size. This can be achieved by trimming edges and using compression tools, such as Adobe Photoshop™, Adobe Fireworks™ or other third party applications.

Example:



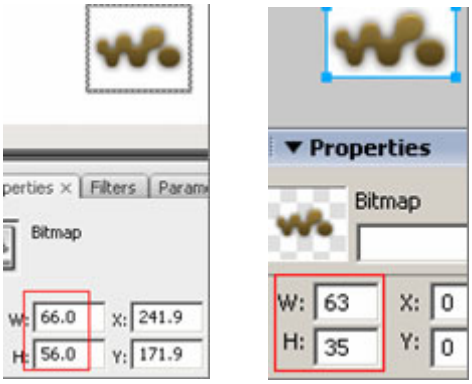
The size of the bitmap above was reduced from 8kb to 2.25 KB. It was converted to a PNG8 bitmap with 68 colours, and a third party application was used to apply more compression without major loss in quality.

Transparency

Using many images with transparent areas and overlapping them is a CPU intensive task and may cause graphics on the screen to flicker.

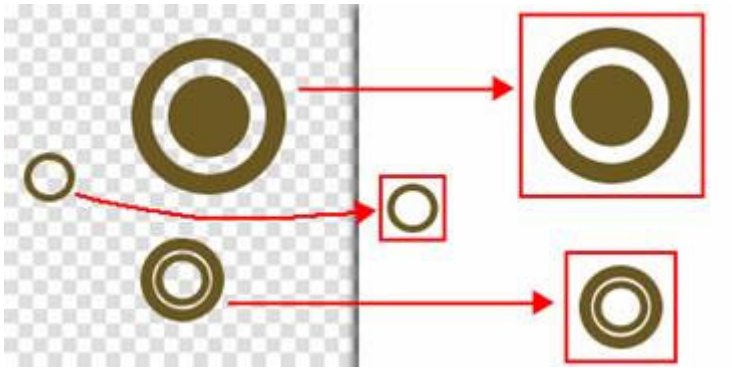
The size of transparent areas in images should be reduced as much as possible. The Flash player always recalculates all parts of images including the transparent pixels. To optimise, transparent areas should be minimised.

Example:



By trimming the png image to the left with dimensions 66 x 56 pixels, the transparent area around the edges was reduced. The image was trimmed to 63 by 35 pixels. With larger images, trimming transparent areas definitely improves performance.

Example:
Large bitmaps with many transparent areas can be converted into separate images, and then each piece can be converted into a separate movie clip. This way the Flash player does not need to render every transparent pixel. The image to the left has a large transparent area, which can be reduced by converting the circles into separate bitmaps and then into movie clips.



When bitmaps are placed on a solid colour background, another solution is to use this background colour instead of transparent areas.

Dot per inch optimisation

Make sure that the DPI used for bitmaps placed in your Flash file is not higher than 72. If you can manage with lower DPI settings than that is going to be even better.

Optimising animations

Maybe what is done in 300 frames on a PC ought to be done in 100 frames or less on a phone. Using fewer frames in animations improves performance and reduces memory usage.

Motion tweening vs. scripted animation

Motion tweening is generally faster than scripted animation, particularly with simple animations. ActionScript tends to consume more of the CPU processing power and time, since the Flash player has to translate the code into native language and then do the necessary calculations to move objects. Which method to use depends on the animation complexity, the phone and the purpose of including the animation. Too much focus on details sometimes consume development time and CPU processing power without giving the desired user experience, particularly when played on small screens.

Combining animation with changes in alpha or other effects related to transparency or text animation definitely slows performance. The number of moving objects and the number of effects that goes along with them should always be kept as low as possible when performance is an issue.

Keeping the animation region small and not covering a large portion of the screen also improves performance, since the Flash player redraws moving objects within a bounding box for each frame as shown in the image below. One way to optimise in this case is to keep the bounding box as small as possible.

When using transition effects, going from one view to another, the background animations and other minor effects that are not visible during this transition should be paused or removed.

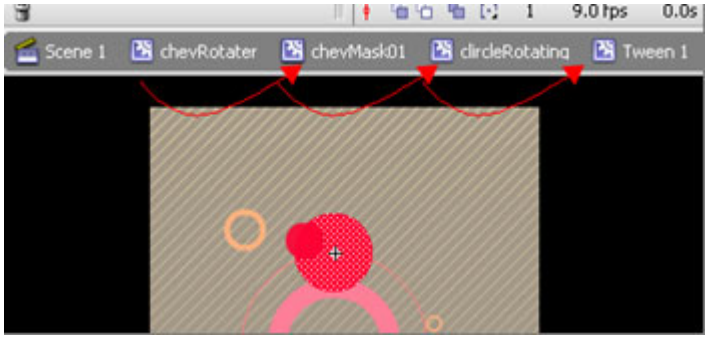
Placing objects on the screen

Object properties like X and Y coordinates, width, height, scaling factor and rotation angle should as much as possible be set to integer numbers instead of decimals to make calculations and rendering faster in the Flash player.

Working with symbols

Nested symbols

Avoid nested symbols on multiple levels as much as possible. Nested symbols require a lot of runtime stack memory and slow CPU performance. In some cases multiple nested symbols can cause the phone to freeze. Three or four levels of nesting should be enough, anything more than that tends to complicate things. The images below illustrates a case where five levels of nesting was reduced to three.



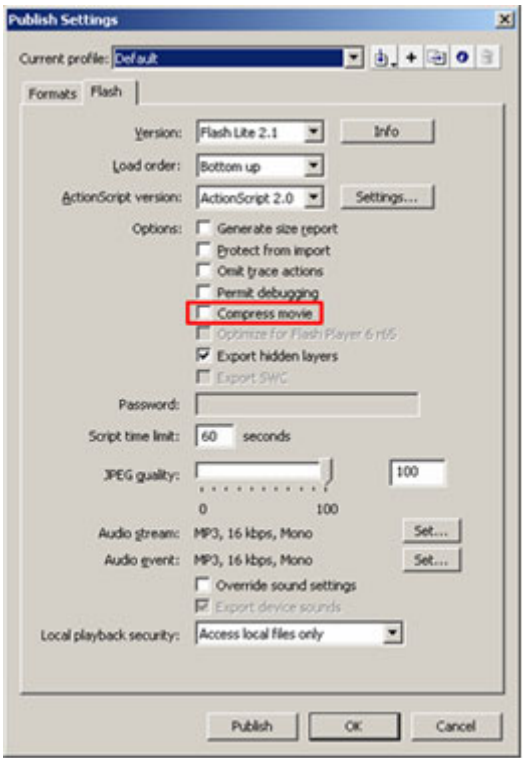
Seven levels of nested movie clips can cause Device Central to display the error message "Stack limit reached". In the phone the image turns white when this error occurs.

Grouped shapes

Grouped shapes should be avoided. If the graphics require duplication, it is better to convert the shapes into movie clips. Movie clips are easier for the processor to calculate, even with several instances on the stage, since each instance of a group is calculated independently.

Compressed symbols

In Adobe Flash, the *Compress movie* option in the *Publish Settings* dialog – *Flash* tab is selected by default. Selecting this option leads to a small SWF file after compiling the .fla file. This setting is suitable for the Internet since the SWF file will download faster. On phones, compressed movie clips use more of the heap memory, therefore this setting should be unchecked.]



Reusable symbols

Symbols that are used more than once should be converted to movie clips. This makes the SWF size smaller, since instances of the same movie clip can be used over and over instead of redundantly creating similar elements. This is very useful both for reusing bitmaps and to avoid copying and pasting vector graphics manually while building the scene.

Working with standard UI components

Avoid using Adobe standard UI components shipped and installed automatically with Flash CS3 or CS4. The issue with these components is that they are not optimised for mobile phones and require a lot of memory. Light UI components for Flash Lite developers can either be downloaded from the Sony Ericsson website at http://developer.sonyericsson.com/site/global/docstools/flashlite/p_flashlite.jsp or can be custom made by the developers themselves.

Working with text

Text animation

Flash Lite 2 and 3 support static, dynamic, and input text fields. From the Flash perspective these types of text are complex vector shapes that require a lot of CPU processing power to draw and render. Using device fonts leads to better performance, since these fonts use less complex vector shapes for drawing each character.

When text animation has to be used it should not be placed over another animation, particularly when working with anti-aliased texts. Animating several anti-aliased texts can slow the performance significantly. One way to avoid this is to temporarily lower the Flash Lite rendering quality from high to medium or from medium to low while animating the text, and set it back to the higher rendering quality when the animation is complete. This can be achieved, for example, by using `fscommand2("SetQuality", "low")`. If the text does not change or can not be animated with the lower setting, using the text as a bitmap should be considered.

Multi-line dynamic and input text

When working with multi-line dynamic and input texts, calculating the line breaks is a CPU intensive process. This is because the line breaks of the strings are calculated at runtime and are recalculated each time the text field is redrawn. Static text fields do not have this problem since line breaks are calculated at compile time.

Embedded fonts

Embedding fonts in the SWF file increases its size because it implies embedding the font file. If it is necessary to embed fonts, only including the needed glyphs should be attempted. For example, if the input text field receives only numbers then only the number characters 0 to 9 should be included.

Embedding fonts also requires a lot of static memory, whereas device fonts use the fonts in the phone and do not require any extra memory.

Sometimes, using dynamic text boxes with fixed width and only embedding the needed fonts in the SWF file can lead to good performance improvement. This implies that static text fields should be replaced by dynamic ones and the fonts to be displayed should be embedded or, as mentioned above, static texts may be replaced by bitmap images. There is a trade off between memory consumption and performance that the Flash developer may want to keep an eye on while using dynamic text boxes and embedding glyphs or using bitmaps to resemble static texts.

ActionScript and some common pitfalls

Using onEnterFrame

Scripted animations, if conditions, math calculations and anything associated with or placed inside an `onEnterFrame` event is invoked each time the Flash player redraws the frame, unless the `onEnterFrame` is deleted.

Example

A particle system creates a star field where the stars are moving towards the viewer. If the required coordinate calculations are continued forever without applying some checks to stop unnecessary parts, there is a risk that the phone will freeze or run out of memory.

```
container.onEnterFrame = function ()
{
     if (vz > 2000)
     {
          vz = vz * friction;
     }
     for (var _loc5 = 0; _loc5 < numClips; ++_loc5)
     {
          var _loc3 = this["clips" + _loc5];
          _loc3.z = _loc3.z + vz;
          if (_loc3.z > 80000 - fl)
          {
               _loc3.z = _loc3.z - 80000;
               _loc3.x = Math.random() * 6000 - 3000;
               _loc3.y = Math.random() * 6000 - 3000;
          } // end if
          var _loc4 = fl / (fl + _loc3.z);
          _loc3._xscale = _loc3._yscale = _loc4 * 100;
          vpX = vpX + (_root.container.cross._x - vpX) / 100;
          vpY = vpY + (_root.container.cross._y - vpY) / 100;
          _loc3._x = vpX + _loc3.x * _loc4;
          _loc3._y = vpY + _loc3.y * _loc4;
     } // end of for
};
```

The lines highlighted in red are serious pitfalls and may cause a crash when the onEnterFrame code has been executed a number of times. The reason is found in the code

```
vpX = vpX + (_root.container.cross._x - vpX) / 100
```

vpX initially equals to `Stage.width / 2`. On a screen with 176 px wide screen, the initial calculation results in: vpX = 1,759. Recalculations will be executed each time the `onEnterFrame` event is triggered, resulting in consecutive vpX values 0.0102544615529, 0.0064104283209, 0.0062876647315 and so forth until the CPU crashes. To avoid this, an `if` condition should be added, checking when the targeted location or size of the particle is reached, and then reset the calculation.

In scripted animations where acceleration is used, optimising the code could also include deleting `onEnterFrame` as soon as the object reaches its targeted location. One solution is to check the change of the X or Y coordinates between two consecutive calculations, and, if the change is zero, delete `onEnterframe`.

Using duplicateMovieClip and attachMovie

If a `duplicateMovieClip` command is allowed to create too many instances on the stage, for example, in particle systems, this may slow performance drastically especially if the duplicated objects are vector shapes. A limit should be placed on the number of instances this command creates.

When the duplicated instances are not needed any more, they should be removed as soon as possible using the `removeMovieClip` command. The same applies to `attachMovie`. When this command is used, calling `removeMovieClip` when the movie clip instances are no longer needed helps to avoid memory leaks and improves rendering time.

Using setInterval

`setInterval` calls a function or does something like updating time at specific intervals. It can also be used to loop as long as the player head reaches the frame where this piece of code is nested. Some interval settings may be valid only for a few seconds and then need to stop. When `setInterval` becomes inactive or is not needed any more, the interval should be cleared with the `clearInterval` command to preserve memory. In addition, when you remove a movie clip from the SWF file, it will not remove any `setInterval()` function running within it. You always have to remove the `setInterval()` function by using `clearInterval()` when you have finished using it.

Flash reserved words

Using variable names, function names and other declarations reserved by Flash must be avoided. In the image below, declared names that may cause problems are surrounded by red frames. The editor has marked them by making the characters blue,

```
69  //----------------------------------------
70  //
71  function init()
72  {
73      for (var _loc1 = 0; _loc1 < numClips; ++
74      {
75          var _loc2 = container.attachMovie("c
76          _loc2.x = Math.random() * 6000 - 300
77          _loc2.y = Math.random() * 6000 - 300
78          _loc2.z = Math.random() * 80000;
79      } // end of for
80      walkman.swapDepths(100);
81  };
82
83  var numClips = 20;
84  var fl = 500;
85  var vz = 3000;
86  var friction = 9.800000E-001;
87  var vpX = Stage.width / 2;
88  var vpY = Stage.height / 2;
89  var TvpX = vpX;
90  var TvpY = vpY;
```

Visibility and alpha

Symbols that are to appear just once or reappear only when needed should be removed from the stage when they are not shown any more. The recommended method is to use `removeMovieClip`, not to change visibility to `false` or set alpha to zero. Setting visibility to false works fine but there will still be one object added to the root array or movie clip array, and it will still be required from the Flash player to assess the movie clip and decide whether to render it or not. If `removeMovieClip` is used, some bytes can be saved and rendering time can be speeded up a couple of milliseconds.

It is better to avoid setting alpha to zero. The downside with this method is that Flash renders the symbols regardless of their alpha. Removing movie clips improves performance since Flash renders fewer objects and memory is freed. A movie clip rendered with alpha set to zero still receives mouse events and is treated as any movie clip. Setting movie clips to be transparent or semi-transparent always affects performance.

If a symbol is to be called frequently, setting its coordinates to off-stage values when it is not needed, is a better method than changing its alpha or removing it from the stage and then attaching it at a later point. Relocating an object is faster than repeatedly drawing and removing it.

Strings and arrays

String and array handling are CPU intensive tasks. String and array manipulation should be avoided whenever possible.

For example, if a string is used only to represent integers, it should be declared as an integer instead. Likewise, it is better to declare a variable type as a Boolean instead of a string and assigning it values like focus = "false", because then you will have to stretch your code for no reason, and you will end up using something like this:

```
else if (action == "lunaChangeFocus" && focus == "true")
```

Classes

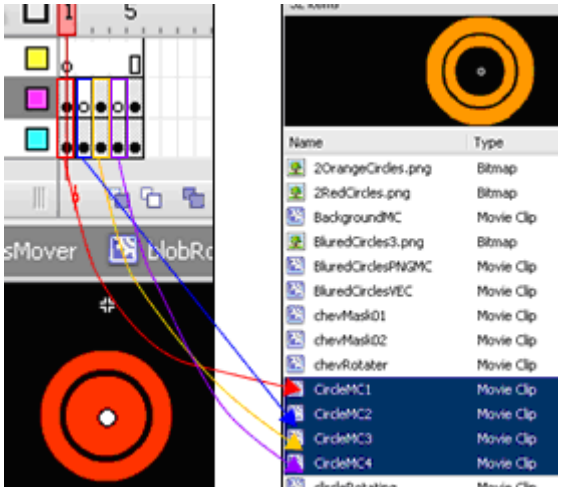
One way to reuse code is to adopt Object Oriented Programming (OOP) techniques which involves using classes, inheritance and polymorphism. To what extent OOP should be used depends on what the developer wants to achieve. Sometimes it is necessary to use classes and inheritance while other times OOP is not needed at all. When using OOP the following tips may be useful:

- Keep the class structure as simple as possible.
- Use compact classes and code.
- Make chaining (namespaces) simple and short. This shortens startup time and lookup time in the hash table.
- Keep inheritance as low as possible. Inheritance complicates method calls and uses more memory. Try to find a balance between class extensibility and efficiency. Whenever possible, place called methods and functions in the same class, compact classes are more efficient during run time.
- Classes which were loaded dynamically as part of an external SWF file during runtime must be manually deleted from memory after unloading the SWF file, otherwise the byte code of these classes will remain in memory. Try the `delete` statement and specify the full path of your class names, for example `delete folder1.folder2.SomeClass`.
- The length of variable, property and method names also affect the string hash table and look up time.

Issue with gotoAndPlay or gotoAndStop

When `gotoAndPlay` or `gotoAndStop` are used, the Flash player renders everything between before displaying the targeted frame. Thus the invisible frames are loaded when not necessary. This consumes memory and drains CPU power, particularly if what is in between is complex or heavy content, for example, shape tweening or anti-aliased text along with a number of different frames.

One way to overcome this problem is illustrated in the image below. Content differs across all the five frames. Each frame can be converted into a separate movie clip (the highlighted items in the library) and be attached to the stage when needed. Even if the targeted symbol needs to be duplicated, all nested content must not.



Working with variables

In general, the Flash player works faster with local variables like the ones placed inside functions than with globally declared variables. The var declaration should be used whenever possible, because as soon as the function is exited, these variables are destroyed and some memory is freed.

Global variables should be reused as much as possible and set to null when not needed anymore so the garbage collector can take care of them.

Variable declarations should be as short as possible. For example,
`var FramesPerSecond:Number = 0;`
can be replaced with
`var fps:Number = 0`

According to Adobe, Flash developers should use strict data typing in their variables whenever possible because it helps in the following ways:

- Adds code completion functionality, which speeds up coding.
- Generates errors in the Output panel so there will be no silent failure when the SWF file is compiled. These errors helps in finding and fixing problems in applications.
- To add a data type to variables, the variables must be defined using the `var` keyword. In the following example, when creating a `LoadVars` object, strict data typing should be used:

  ```
  var paramsLv:LoadVars = new LoadVars();
  ```

  Strict data typing provides code completion, and ensures that the value of `paramsLv` contains a `LoadVars` object. It also ensures that the `LoadVars` object will not be used to store numeric or string data. Because strict typing relies on the `var` keyword, strict data typing cannot be added to global variables or properties within an object or array.
- Helps users understand your code.

- Do not overuse the `Object` type. Data type annotations should be precise to improve performance. The `Object` type should only be used when there is no reasonable alternative.
- Keep variables as short as possible while retaining clarity.
- Make sure variable names are descriptive, but do not go overboard and use overly complex and long names.
- Only use single-character variable names for optimisation in loops. Optionally, single-character variables can be used for temporary variables in loops (such as i, j, k, m, and n). Use these single-character variable names only for short loop indexes, or when performance optimisation and speed are critical. The following example shows this:

```
var fontArr:Array = TextField.getFontList();
fontArr.sort();
var arrLenght:Number = fontArr.length;
for (var i:Number = 0; i< arrLenght; i++) {
this.attachMovie("idName", "newName" + i, i);
this["newName" + i]._y = spacing + 10 * i;
this["newName" + i].textField = fontArr[i];
}
```

Note: Strict data typing does not slow down a SWF file. Type checking occurs at compile time (when the SWF file is created), not at runtime.

Figures from redundant calculations should be stored in variables or tables

Trigonometric calculations for games or animations can slow the application, especially if they are extensively used, for example, calculating sine, cosine and arctangent values to move objects in an orbital or wavy way. In such cases, developers may use the `Math` class and calculate the different values for rotations or x and y coordinates using the `sine` and `cosine` methods. Sometimes the same calculations are needed repeatedly during run time, which means that developers repeatedly have to access the `Math` class and invoke the calls for `sine`, `cosine` and `arctangent`, and the CPU then has to recalculate things while keeping up with rendering the scene. One way to save CPU performance is to create a table with constants during compile time or a manually predefined table and then use it during run time without causing the CPU to do calculations repeatedly. In a similar way, the value of `Math.PI` can be stored in a constant variable and then be referenced in the code to avoid both accessing the `Math` and getting the long PI value (3.14159265358979). It is better to use `var myPI:Number = 3.14.`

Example of storing `Math.PI` in a variable and sine radiant values in a lookup-table during compile-time:

```
Var _PI:Number = 3.14;
var sineTable:Array = new Array();
for (var angle:int = 0; angle < _PI*2; angle ++)
{
     sineTable[i] = Math.sin(angle);
}
```

As mentioned above it is also possible to use a constant table with manually fed-in values. Make sure in both cases that the calculations or data retrieval does not delay the start-up time of your application.

Anonymous function calls

Syntax like:

```
musicMode = function(){
     _root.walkman = true;
...
...
}
```

can be replaced by the following, more efficient and faster code:

```
function musicMode()
{
_root.walkman = true;
...
...
}
```

The same applies to the code:

```
myObj.eventName = function( ) { … };
```

The following is more efficient:

```
function myFunc( ) { … };
myObj.eventName = myFunc;
```

Loop iterations

Loop iterations, such as for, while, do-while, frame based loops and enterFrame slow the processor, particularly when there are many things to do or to be checked by the `If` condition. The number of loops used and the amount of code each loop encloses should be carefully considered. There are many techniques and examples on the web explaining how `for` loops or `while` loops can be optimised in better ways as in loop interchange, peeling, fusion or loop combining, and so on.

Frame based loops should be stopped using `stop()` or `gotoAndStop()` as soon as they are not needed.

Event listeners

Flash Lite 2.x allows developers to build listeners that detect events, for example, which keys have been pressed. When these listeners become of no use, for example, when a new scene is loaded or when the movie clip is unloaded or removed from the stage, the listener data residing in memory is not released unless that part of the ActionScript is explicitly cleared. To free some memory when listeners are no longer used they should be removed with the `removeListener` command.

Limitations with regard to the display area

Publishing content at the correct resolution

When publishing content, it should be set to the correct screen resolution of the targeted phone. Although Flash movies stretch to adjust to screen dimensions, rendering scaled movies do affect performance. When creating a project, the settings in Device Central should be compatible with the targeted device.

Note: To prevent Flash content from scaling to fit a larger screen than that of the targeted phone, the following piece of code can be used:

```
Stage.scaleMode = "noScale";
```

The screen area is the boundary

Scripted animations with coordinates way off the stage, or symbols that are calculated to be off the screen, use CPU power for invisible items, while the Flash player may be rendering other things at the same time.

For example, the function below falls in this pitfall. The external display area on some phones is 176 by 159, but in the lines highlighted in red, ActionScript generates X and Y coordinates at values between 3000 and -3000.

```
function init()
{
    for (var _loc1 = 0; _loc1 < numClips; ++_loc1)
    {
        var _loc2 = container.attachMovie("clips", "clips" + _loc1, 10 + _loc1);
        _loc2.x = Math.random() * 6000 - 3000;
        _loc2.y = Math.random() * 6000 - 3000;
        _loc2.z = Math.random() * 80000;
    }
    walkman.swapDepths(30000);
};
```

Symbols, graphics, coordinates and similar properties should be restricted to fit in the display area as much as possible. For example, if the screen is 176 by 159 pixels then a movie clip with dimensions of 500 by 500 pixels does not make sense. It only causes more burdens on the CPU while animating, rescaling or rotating the objects. If an object should cover the entire screen at the beginning, and gradually shrink down to its real size, an initial size of, for example, 180 by 160 pixels should be more proper. The object should then quickly be shrunk to its intended size.

Keeping content clean and tight

Unused objects or code

Unnecessary pieces of code should be removed because they are processed and translated by the Flash player into the native language.

Empty frames, layers and movie clips should be removed, since they may add up to the final size of the SWF file or use valuable parts of phone memory.

Cleaning is strongly recommended. ActionScript affects performance to a large degree and should be kept to the minimum amount possible. Commands like `trace` used during the test phase must be deleted in the end. It is also possible to select the *Omit Trace Actions* checkbox on the *Flash* tab in the *Publish Settings* dialog.

Variable declarations that are not used anywhere in ActionScript must also be deleted to avoid unnecessary processing.

Duplicate work

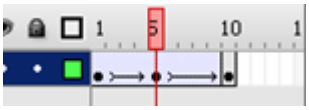
The lines highlighted in red shows an example of the same command executed twice.

```
_quality = "high";
Stage.scaleMode = "noScale";
fscommand2("SetQuality", "high");
```

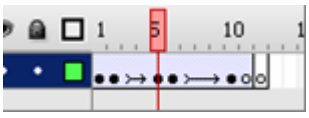
Simplifying the content

Only the necessary key frames, symbols, ActionScript, variables, library items, and motion tweening should be used, in the simplest way possible.

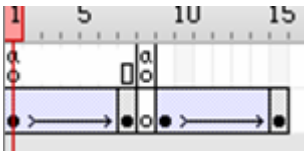
For example, a project contains a motion tweening fading a movie clip in and out over 10 frames. This can be implemented by plotting three key frames, one at the beginning (frame 1) where alpha is set to 100, a middle frame (frame 5) where alpha is set to zero, and the last frame (frame 10) where alpha is set to 100. This gives a simple fade out – fade in effect.



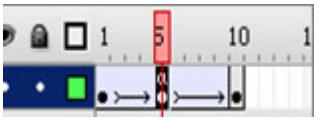
During development and testing developers sometimes insert additional frames in between, frames that are not responsible for doing anything and can be removed before finalising the project.



Another case were it may seem proper to add an additional frame is when the developer wants the player head to stop on the fifth frame where the symbol is completely invisible because that is where the alpha is set to zero. It may be tempting to tween the object to the 4th frame, set the alpha to zero and then insert a blank key frame on the 5th frame and place the `stop()` or `gotoAndStop` command on that frame.



A simpler and more straightforward solution would instead be to place the `stop()` command on the 5th frame where alpha = 0 and nothing is visible.



Simplifying the code

One action to take is to trim the code and make it as simple as possible.

For example, instead of using two functions to create yellow and red blobs, only one function needs to be created to do that, unless there is a good reason for using two separate functions. In other cases, attaching yellow and red blobs at the start time from the library to the stage may not require any function at all.

Another example is where a couple of objects are nested within a movie clip and some common property, for example the RGB value, of these objects are to be changed on all the objects. It could be a parent movie clip called `Container`, with nested movies `mc1`, `mc2`, `mc3`. To change the RGB for the child movies individually using advanced RGB to transform from one colour to another can be done using ActionScript:

```
Container.mc1.setRGB(something);
Container.mc2.setRGB(something);
Container.mc3.setRGB(something);
```

Since all the nested movie clips are to be changed to the same new colour, a better solution could be to simply change the RGB of the parent `Container` to the desired colour, because subsequently the children will take that colour.

Keeping the library small

Only needed objects should remain in the library. Extra items should be removed, particularly the ones with linkage identifiers not used anywhere in ActionScript code. Such objects may add to the final SWF size.

Highlighted optimisation tips

- Place similar objects, bitmaps, vectors and text fields together on neighbouring levels so Flash do not need to repeatedly switch between different rendering methods. This makes the rendering engine perform better.
- Avoid extensive use of masking, simultaneous animations, too many frames and perfectly smooth tweening. Sometimes effects can be achieved with a smaller amount of frames and no tweening. Some animations works better when created manually "frame by frame". This may require a more work, but it can help the phone to avoid the math calculations associated with path defined motions.
- Keep ActionScript code as simple and compact as possible.
- Keep in mind that the math behind the interactivity, special effects, particle systems and animation are some of the bottlenecks to consider when performance is an issue.
- Experiment and test different solutions to find out what is most effective.