ActionScript 3 optimization techniques

Joa Ebert*

April 26, 2008

Abstract

This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License[1].

This document contains several optimization techniques for ActionScript 3 (AS3). Most of these techniques have been discovered by members of the Adobe Flash community. This paper is a comprehension of existing techniques to achieve faster code execution. There are trivial optimiztions that one can follow during daily work and advanced techniques that require more than just basic ActionScript 3 knowledge.

Contents

*With credits and thaks (in no particular order) to André Michelle, Nicolas Canasse, Thibault Imbert, Claus Wahlers, Mario Klingemann, Ralf Bokelberg, Ralph Hauwert, John Grden, Carlos Ulloa and Michael Baczynski

[1] http://creativecommons.org/licenses/by-sa/3.0/

1 Introduction

A lot of good optimization techniques have been found by a lot of developers. The only problem is that they are located on a lot of different places and get lost. This document contains most of the tricks that have been found so far.

Because not all of these techniques require the same level of understanding they are split into three categories. The "More advanced" category may be used carefully. If you do it the wrong way you could get the opposite result which is slower code execution.

1.1 Testing environment

There are a lot of tables in this documents that cover execution time. They were done on a machine with a Intel Core2 CPU T7200 at 2GHz using 2GHz, 2GB Ram with Windows XP Professional SP2. One important factor is the player which is the release Flash player version 9.0.60.108. The tests were not done using the debug player and they were not running in debug mode which is very important in specific cases. Each test case was running ten times. The result shown is always the average execution time of the ten runs. This way two tests do not conflict with the garbage collection and each test is just responsible for its own memory usage. Before each run there is a ten seconds delay in order to not conflict with the Flash player starting procedure.

Here is an example of a test case and how it was executed. All the code that is not necessary like the display of the results has been removed.

```
package
{
  [SWF(frameRate='255')]
  public class TestCase extends Sprite
  {
    private static const n: int = 1000000;
```

```
    public function TestCase()
    {
      setTimeout( runTest, 10000 );
    }

    private function runTest(): void
    {
      var a: Number = 0;

      for ( var i: int = 0; i < 10; ++i )
        a += test00();

      a /= 10;

      showResults(a);
    }

    private function test00(): int
    {
      var t0: int;
      var t1: int;
      var i: int;

      t0 = getTimer();
      for ( i = 0; i < n; ++i )
      {
        var a: Number = ( 1 as Number );
      }
      t1 = getTimer();

      return ( t1 - t0 );
    }
  }
}
```

2 Basics

In this section you will find techniques that you can easily integrate in your daily workflow. Most of the tips are very common and should just help you avoiding mistakes while migraing from ActionScript 2 to ActionScript 3.

2.1 Native types

There is one simple rule in ActionScript 3. Use integers for iterations. This means if you were using `Number` in ActionScrpt 2 you want to use `int` now in

ActionScript 3.

Iterations	int[ms]	Number[ms]
10^2	0	0
10^3	0	0.1
10^4	0.1	0.4
10^5	1	3.2
10^6	9	24

Slow version:

```
for (var i: Number = 0; i < n; i++)
  void;
```

Fast version:

```
for (var i: int = 0; i < n; i++)
  void;
```

2.2 Typing

If you have to use objects do not forget to type them correct. An example could be a simple three dimensional vector. Usually such an object has a x, y and z property. Bad habit is to use the `Object` class. In ActionScript 3 the AVM2 can make use of defined objects that are not dynamic.

Iterations	Vector[ms]	Object[ms]
10^2	0.1	0.1
10^3	1.3	2.2
10^4	13.3	17.2
10^5	73.4	109.2
10^6	672.8	1033.7

Slow version:

```
for (;i<n;i++)
{
  var v: Object = new Object;
  v.x = 1;
  v.y = 2;
  v.z = 3;
}
```

Fast version:

```
class Vector3D
{
  public var x: Number;
  public var y: Number;
  public var z: Number;
}

for (;i<n;i++)
{
  var v: Vector3D = new Vector3D;
  v.x = 1;
  v.y = 2;
  v.z = 3;
}
```

2.3 Casting

Array access can be speeded up by telling the Flash player what is inside the array. Since arrays can contain any object the player has to figure out what type it is. This step can be eliminated by casting the object.

Iterations	Cast[ms]	Without cast[ms]
10^2	0	0
10^3	0.3	0.5
10^4	3.3	3.3
10^5	23.5	25.3
10^6	173.3	180.9

Slow version:

```
for (;i<n;i++)
  array[i].x = 2;
```

Fast version:

```
for (;i<n;i++)
  Vector3D(array[i]).x = 2;
```

2.4 Promoting issues

If you do something like `i * 2` the expression will be promoted as a `Number`. Array access has been optimized for integer numbers. So whenever you do some calculations make sure that you keep the `int` type by casting again as an integer.

Iterations	Cast[ms]	Without cast[ms]
10^2	0	0
10^3	0.1	0.2
10^4	1.3	1.3
10^5	12.8	14.7
10^6	69.1	77.2

Slow version:

```
for (;i<n2;i++)
  Vector3D(array[i*2]).x = 2;
```

Fast version:

```
for (;i<n2;i++)
  Vector3D(array[int(i*2)]).x = 2;
```

3 Advanced

3.1 Instance re-using

Using a lot of uneeded istances slows the Flash player down in two ways. The first is that a lot of garbage is created which means the garbage collection has to do extra work. The second argument against a lot of unneeded instances is simply that you have to create them which takes an extra amount of time.

Iterations	GC friendly[ms]	Using new[ms]
10^2	0	0.1
10^3	0	1.3
10^4	0.3	12.8
10^5	2.7	135.6
10^6	28.3	1299.3

Slow version:

```
for (;i<n;i++)
  p = new Point(i, i);
```

Fast version:

```
for (;i<n;i++)
{
  p.x = i;
  p.y = i;
}
```

3.2 try..catch and null

Never use `try..catch` if you can use `null`. It is way slower than the comparison if something is `null` or not.

Iterations	null[ms]	try..catch[ms]
10^2	0	2.8
10^3	0.1	29.7
10^4	0.2	301.9
10^5	1.1	3022.9
10^6	10.6	30740.3

Slow version:

```
var o: Sprite;

for (;i<n;i++)
{
  try
```

```
  {
    o.blendMode = BlendMode.ADD;
  }
  catch ( error: Error ) {}
}
```

Fast version:

```
var o: Sprite;

for (;i<n;i++)
  if ( o != null )
    o.blendMode = BlendMode.ADD;
```

3.3 Pairs

TODO

3.4 Lookup tables

TODO

3.5 Bitwise operators

TODO

4 More advanced

4.1 Constructor

Code inside the constructor is not optimized by the Just-in-time compiler (JIT). To use JIT optimized code there is the possibility to call a function out of the constructor. The code inside that function is then optimized again.

The reason why there are no test results here is that I could not find a real rule when it makes sense to do this or not. Usually you expect a faster code executin but if there is no difference because the JIT is not used at all which could happen you have an even slower code executon because of one extra function call which is your initializing function.

4.2 Constructor parameters

TODO

4.3 Linked lists

TODO

4.4 1D loops

TODO

4.5 2D loops

Usually a way to loop through a two-dimensional array is always the same. This array can either by an `Array` or a `BitmapData`. Unfortunately this is easy but not very fast. There is always a jump once the itration by one dimension is done and one value has to be reset.

Instead of jumping from the end point to the begining we just cycle through the two-dimensional object. This has a big impact on performance for large objects.

Make sure to use `x` and `y` careful since the value is only used once and not more often. This example loops from top-left to bottom-right / bottom-left. This depends on the height. If the height is odd you will end at the bottom-right. If it is even you will end at the bottom-left.

As you can see the simple `for`-loop variant is actually faster until 512^2 iterations. So you have to make the right decision when to use which method.

Iterations	Cycle[ms]	Simple for[ms]
128^2	2.7	2.7
256^2	11.1	10.3
512^2	29.7	28.3
1024^2	95.4	106.3
2048^2	355.8	738.6

Slow version:

```
var color: int;
for (var y: int = 0; y < height; y++)
  for (var x: int = 0; x < height; x++)
    color = bitmapData.getPixel(x, y);
```

Fast version:

```
var color: int;

var y: int;
var x: int;
var m: Boolean = true;
```

```
while (true)
{
  color = bitmapData.getPixel(m ? x++ : --x, y);

  if (x == width || x == 0)
  {
    if (y++ == height)
      break;
    m = !m;
  }
}
```

Note: Michael Baczynski[2] found several issues I could not reproduce with this code. If a simple for-loop is faster for you I would like to get a response in that case.

4.6 Type conversion

TODO

4.7 14-bit integers

5 Bytecode level

When writing on a bytecode level even more optimizations can be achieved. This is definitly not something for daily work and only important if you really know what you are doing.

Bytecode is always shown in an *asm* block with pseudo-code notation.

5.1 Registers

TODO

5.2 Lookup storage

TODO Text and table
Slow version:

```
for(;i<n;++i)
  Math.sin( 1.0 )
```

Fast version:

[2] http://lab.polygonal.de

```
__asm{
  getlex Math
  setlocal 1
}
for(;i<n;++i)
  __asm{
    getlocal 1
    pushbyte 1
    callProperty sin, 1
  }
```