Optimizing Adobe AIR for Code Execution, Memory and Rendering

Sean Christmann

EffectiveUI

11/16/08

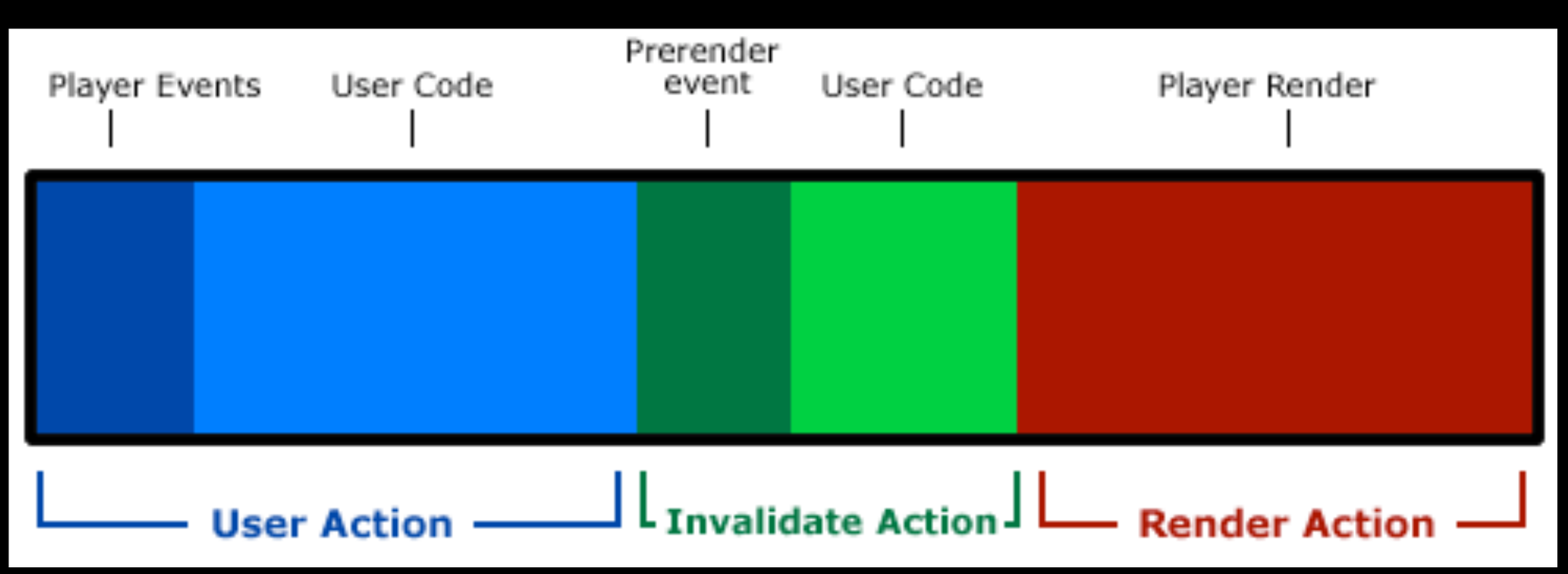
Adobe

What we'll be covering

- Understanding AIR performance is about understanding the Flash Player
  - Dissecting the Flash 'Frame'
- Main topics
  - Execution speed of user code (Flash | AIR)
  - Managing memory (AIR only)
  - Rendering Performance (Flash | AIR)
- Q & A

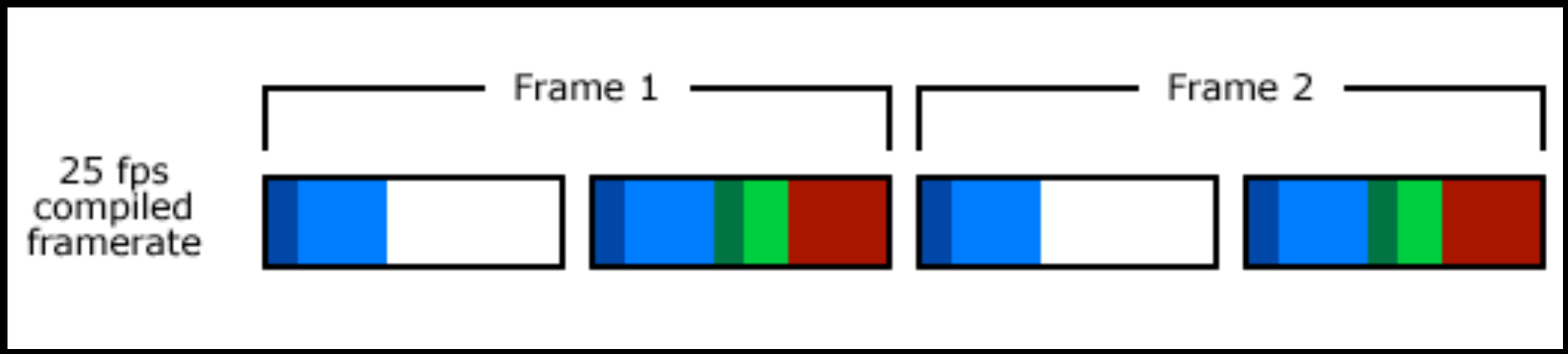
Flash event loop

- Flash player is delegated event loops to run on.
- Event loops range from 5 ms to 20 ms intervals minimum, and expand to a maximum depending on code or rendering time.
- Event loops are processed in the following order:



Flash frame generator

- A frame is generated by stitching one or more event loops together and marking the final loop to process a render.
- Event.ENTER_FRAME will mark the beginning of a frame and Event.RENDER will mark the end or a frame.



Flash frame optimization

- Understanding the process of frame assembly will go a long way in helping perceived performance.
- Separating heavy tasks to run across multiple frames will keep your UI responsive. Users will respond more favorably to a loading icon then a locked interface.
- Same goes for splitting up heavy rendering jobs.
- Flash doesn't support creating arbitrary threads to offload processing off the main event loop.
  - Although Flash 10 provides access to a separate processing thread by way of the ShaderJob. If you can abstract your logic into a PixelBender workflow, you have your own thread to send work to.

Code Execution - Universal AS3 Optimizations

- Typing will bring you the most significant performance impact.

```
var p:Point = new Point();
p.x = Number(1);
p.y = Number(2);
```

- Typing prevents runtime evaluation of method signatures which can slow down the VM.
- This applies to web service data too, if you can cast generic JSON or XML results in to static typed objects, this can have a major influence on application execution.

```
[
  {"_type_":"RecordClass", "name":"Joe"},
  {"_type_":"RecordClass", "name":"Bob"}
]
```

Code Execution - Universal AS3 Optimizations

- Code with 'JIT awareness'
- JIT = Just In Time compiler
  - VM will compile AS3 bytecode to native machine code for the target platform
- JIT won't compile code within constructors, keep these lightweight

Code Execution - Pick your objects

- Primitives offer the best bang for your buck
  - Flash supports the int and uint primitive.
  - Primitives are unboxed values meaning they aren't hidden behind an object, which makes them fast to access.
- Bytearrays allow for faster access of primitives over Array
  - Bypasses the need to box values for storage.
  - Stores floating point numbers and strings without their boxes.
  - Flash 10 <Vector> class uses this same theory.
- Regex is an order of magnitude slower then String functions for searching text.
  - Use regex for validation, use string methods for searching.
  - Call out to webkit if you absolutely must use regex for matching large text.

Code Execution - Pick your data model

- AMF processes on the main event thread, not on the URLLoaders thread.
  - AMF deserialization can't be spread over multiple frames.
  - Consider handling the AMF data by hand and reading out just the objects you need
  - Change to JSON or XML and process in chunks.
- Avoiding using data binding on data transfer objects or AMF objects
  - Binding is your friend for UI reflection, and your enemy everywhere else.

Code Execution - Pick your UI

- Avoid unnecessary parenting in Flex
- Follow the Flex component model
  - createChildren()
  - commitProperties()
  - updateDisplayList()
- Only use Datagrids as a last resort, make sure you can't implement in a regular List first.
  - Also avoid Repeaters for scrollable data

Memory - What's your size?

- AIR requires ~17MB of private memory to launch with 1 window.
- Every window adds 4 bytes per pixel to initialize the stage.
  - 1024x768 window = 3MB
- Assume ~4KB per display object added to memory footprint
- AIR lazy loads extra modules (SQLite and WebKit) and keeps them in memory until the app quits.
- Data models and class memory typically have a much smaller impact on overall memory compared to DisplayObjects.

Memory - Where's your size?

- Profiler only shows memory used by the Flash VM
  - This means you'll only see Class data, and Object allocation, and additional packages like skins.
  - Profiler won't show the Rendering memory, AIR utils like SQLite, system memory like windowing, as well as Socket data.
- Profiling with system tools
  - Activity Monitor on Mac, Process Explorer on PC (requires extra download)
  - Private memory column, or private bytes column will show the actual real memory used by your application
  - Shared memory column represents system libraries shared across all apps like windowing or file access libraries. Don't worry about Shared memory.

Memory - Managing growth

- Use disk space and SQLite to avoid unnecessarily high app memory.
- Reuse objects to maintain a memory plateau
  - DisplayObjects
  - URLLoader objects
- Large single objects fragment less across memroy pages then multiple small objects.

Memory - Cleaning up

- Flash and AIR use a mark and sweep garbage collector
- Garbage collector can be invoked directly in AIR

```
flash.system.System.gc();
```

- Make sure you clean up Asynchronous objects directly, nulling these objects doesn't unhook them from the Flash player.
  - Timer
  - Loader
  - URLLoader
  - File/SQLite

Memory - Cleaning up

- Where things start falling apart
- Garbage Collector may need some help getting started

```
private var gcCount:uint;
private function startGC():void{
        gcCount = 0;
        addEventListener(Event.ENTER_FRAME, doGarbageCollect);
}
private function doGarbageCollect(evt:Event):void{
        flash.system.System.gc();
        if(++gcCount > 3){
                removeEventListener(Event.ENTER_FRAME, doGarbageCollect);
        }
}
```

- Memory fragmentation is pretty severe in AIR
  - Objects may garbage collect, but memory page won't be released back to system
  - You can't rely on ever getting back to slim memory profile after extended application use.
- AIR can be unreliable at releasing webkit successfully

Rendering - Managing your frame

- Keep your framerate set at 60 fps or lower.
  - Current LCDs locked at 60 hz refresh rate, doubtful to change anytime soon
- Avoid multiple display manipulations per frame, code against ENTER_FRAME events instead of Timer events
- Use <mx:Container creationPolicy=”queued”/> to defer object creation over multiple frames.

Rendering - Stages

- Rendering composed of 3 stages
  - Layout
  - Rasterization
  - Compositing
- Rendering bottlenecks will occur in any of these 3 stages, you may have to make educated guesses to determine which stage to optimize.

- Flash has arguably the best renderer in the industry
- http://www.craftymind.com/guimark/

Rendering - Layout

- Calculate text positioning and measurements.
- Parent child coordinates for all DisplayObjects.
- Determine object size of graphics layer
- Elements that are offscreen are determined for culling.
- Pass objects in view off to rasterizer.

Rendering - Rasterization

- Rasterizer passes over vector positions and generates bitmap for output.
- Rasterizes only objects and rects that layout manager has passed it.
- Renders fonts to view.
- Applies antialiasing to edge and stroke lines.
  - Fill edges are single edge antialiased, strokes are double edge antialiased.
- Objects marked invisible are passed over. Remember that alpha = 0 is not the same as visible = false

Rendering - Compositing

- Compositing brings is all together to render to blit to final Stage context
- Compositing aggregates date from Layout pass and Rasterization pass to formulate final image.
- CacheAsBitmap() will store composite chain for reuse later.
  - Use only when you know if will help, don't assume ahead of time, you can get burned from overusing cacheAsBitmap.
- DrawBitmap() is a great way to composite offscreen elements for later use.
- Drawing to a single graphics context allows you to bypass compositing.

Rendering - Bitmap or Vector

- Bitmaps are constant draw time for a given size, vector is variable.
  - Bitmaps skip the rasterization stage of the renderer.
- Bitmaps require more memory up front, but can be reused over time for better memory consumption.
- Bitmaps allow for baking in post processing effects, skip the need for run time filters.
- Vectors scale and rotate better and will generally be a better aesthetic fit.

Rendering - Additional Tips

- Create DisplayObjects ahead of time to increase perceived speed of transitions.
- Use ScrollRects instead of Masks
  - Masks have to render everything to identify view, scrollRects clip render to bounding box.

AIR on PC and OS X

- Window Transparency
  - PC takes a ~10% performance hit when rendering windows with transparency
  - Mac get transparency for free with Apples window compositing technology
- Event Loops
  - Observations show PC processes more event loops per second then mac, this may lead to faster event dispatching, but probably wont effect rendering time.
- Windowing architecture
  - PC apps are geared toward a ‘windowed instance’ architecture. Apps live and die by the presence of a window so memory usage remains lower per app.
  - Mac apps promote ‘server instance’ architect. Apps remain alive in dock regardless of whether windows are open. This requires more testing around memory lifetime.

Q & A

- Presentation files and PDF available at

http://www.craftymind.com

Adobe