



Adobe Flash Multiscreen Gaming: Best Practices

Recommendations for creating online gaming content for Flash Player 10.1

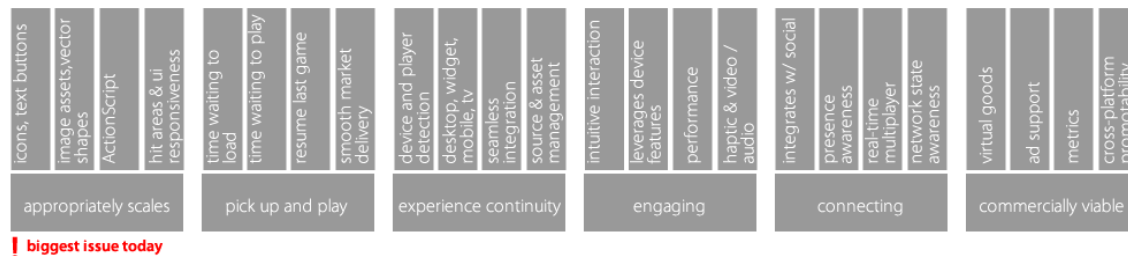
[v5] updated on June 7, 2010

Compiled by Allen Ellison (aellison@adobe.com)

| | |
|---|----|
| High-Level Game Design | 2 |
| Getting Started | 4 |
| Resources | 8 |
| Author Once: Deploy Everywhere? | 10 |
| Do's and Don'ts | 16 |
| Rendering Performance | 17 |
| Data Structure Management | 18 |
| Game Logic, Event Notifications & Sequence Management | 19 |
| Coding Recommendations | 20 |
| HTML Considerations | 24 |
| External Recommendations | 24 |

High-Level Game Design

Introduction



The purpose of this document is to help you understand the issues with porting or developing new games for Flash Player 10.1 on multiple platforms – specifically for mobile devices.

Aside from the information you'll need to start developing, I've identified six important themes that are relevant to mobile Flash games:

- Mobile-Tailored**– when a user navigates to your site, whether to a portal page where they see a list of games, or directly to the URL for a specific game that a friend might have forwarded to them, you want to deliver the right experience based on the device that is being used to access your content. You want to scale the experience appropriately – in terms of size, frame rates, hit areas for buttons, creating the right interaction experience based on whether they're using a mouse (desktop) or touchscreen (mobile), utilizing the DPAD or trackball. In addition, since mobile devices have slower CPUs and less memory available than your desktop, your gameplay experience will benefit immensely by optimizing run-time performance. Also consider, that while your game and your brand might have certain stylistic or behavioral patterns that you want to preserve, the device that you're on may have certain established conventions that users may expect your content to adhere to. While this is less true for web-based content, this becomes more of an expectation when your game is offered as a downloadable application. Other considerations: sleep mode, volume controls, and acoustic quality.
- Pick Up and Play** – game play statistics suggest that mobile game play is significantly different from desktop game play in that mobile game play is more opportunistic, done while waiting for an appointment, on the bus, or in between other activities. Users expect the game play to start immediately and prefer games with a shorter duration and may have to stop playing at any time. When they resume their game play experience, they might expect to, or be pleasantly surprised, if they can pick up where they left off. Unlike a desktop,

you can usually assume that a single person is using the device – therefore you don't need to ask them to set the difficulty level every time they play for example.

- **Experience Continuity** – consider the possibility that users might want to interact with a gaming experience across multiple platforms and devices. Studies show that, given a choice, users prefer to interact with games on their desktop – it's a larger, richer display that has better performance and typically better sound. But when you also consider televisions, tablets and mobile phones – other possibilities emerge such as the ability to play the game separately on mobile phones, while seeing a shared view of the game on the television (think "Battleship"). Even with one player games that you offer both on the desktop and on mobile, and potentially even on mobile as an application on a marketplace – you should consider whether you want to offer the identical experience, an extended experience, or a complementary experience.
- **Engaging** – what qualifies as engaging on the desktop might not be so on mobile. On mobile, content will better qualify as engaging if the experience is tailored to that device, the content is personalizable, and the game leverages the unique capabilities of that device: the accelerometer, the trackball, geolocation (if available), proximity detection, etc. To the extent that users get distracted by the mechanics of interacting with your content, if they have a hard time reading text, hitting a button, scrolling to see the entire game, or trying to figure out how to play the game, they will quickly become disengaged.
- **Connecting** – mobile is all about connecting people on the go – connecting them with content, with individuals, with communities. While there is a niche for single-player, non-committal games, users will be looking for games that allow them to interact with someone in their immediate surroundings, someone that already exists within their self-defined communities, and one or more people from larger communities that they participate in. Connectivity can cover the spectrum from simple leaderboards to non-realtime MMORPG's to turn-based multi-player games to real-time head-to-head competitions. Users might discover other players via social networks, game lobbies, physical proximity, or self-selection using something like Bump¹.
- **Commercial Viability** – aside from the technical details about porting or creating a game from scratch for mobile using Adobe Flash, you will want to consider your monetization strategy. Using the basic remedies of advertising subsidization, one-time application transactions, in-application micro-transactions, or extending a desktop subscription into the mobile domain; or

¹ www.bump.ac --

some combination of the above. If you offer a game on the web for free (or ad-supported) and you want to offer the same game on the marketplace for a fee, consider what the value-add will be that will allow users to justify purchasing the application: do you provide additional functionality or capability, do you provide fresh content, additional levels, etc. Do your research, understand your market, and be informed about the demographics and audiences associated with the devices you're marketing to. Economics will dictate that you pick a specific device or two to go to market with. Your developers will want clear criteria on their first couple of projects as to what device(s) they should be developing against. This is valid because of the different screen sizes, pixel densities, capabilities and behaviors that are unique to each phone (as an example, the Motorola Droid has a keyboard that slides out, and when it does, it locks the screen in landscape mode...this is important if your game requires text entry).

Getting Started

- **Have a Flash-enabled mobile device on-hand.**

I don't recommend trying to use the Android emulator although you can use a service like DeviceAnywhere.com to basically time-share a device (think of it as the mobile device Borg or the mobile Matrix). This device, at a minimum, must have Android O/S version 2.2 (Froyo) installed on it. If you have any questions about how to acquire this device, you should contact the person or organization that made this document available to you.

- **Confirm that Adobe Flash Player 10.1 is installed.**

Navigate to adobe.com/software/flash/about – which, if you have the Flash Player installed will display the Flash Player version number, which should be 10.1.x.y where x is ≥ 61 . Alternatively you can go into Android's **settings** | **Applications** | **Manage Applications** | **Adobe Flash Player 10.1** and see the version number displayed there as well.

If the Flash Player is not installed, you can try navigating to get.adobe.com/products/flashplayer – this should launch the Android Marketplace and prompt you to download and install the Flash Player.

- **Install Adobe Flash CS5**

You can use either Flash Builder or Flash CS5. You can use the older versions of each (Flex Builder or Flash CS4) if you make sure to include flashplayer10_globalswc (the file is called playerglobal.swc but is located in a zip file called flashplayer10_globalswc) in the project. If you're porting an

existing game, chances are that you will want to use Flash CS5 (or CS4).

If you're using Flash CS5 or Flash Builder 4, then you're ready to go. Otherwise you will want to use the playerglobal SWC that ships with Flash CS5 or Flash Builder 4, and there is also one available on Adobe's website at

<http://labs.adobe.com/downloads/flashplayer10.html#additional>

Identifying Mobile Game Candidates

Not every desktop Flash game will translate well to mobile, and by identifying the level of difficulty of optimizing a game for mobile, you'll be able to optimize more games in less time, and at less cost.

STRUCTURAL CONSIDERATIONS

- Consider putting all of the visual elements of your game under a single MovieClip or Sprite --- this will make it easier to deal with orientation issues that might arise.
- Consider your screen real estate and how it's allocated. When on a desktop it'll be natural to expose many controls at the same time, but when on mobile you will want to minimize the amount of on-screen content. Also, one-page instructions might need to become multi-page instructions but with a simpler layout.
- Identify which devices you want to target. I've identified three basic groups of screen sizes other than desktop. If you decide for a target within each group, and then handle the exceptions that will arise within each group, you should be in good shape. For each grouping, I've included a minimum font size recommendation as well as a minimum hit area recommendation. These are only guidelines – for example, the font guidelines assume that you're using a pixel-savvy font, that there is significant contrast and that the background is not noisy. If any of those assumptions are false, you'll get a better user experience by going even larger.

PRE-GAME-PLAY

- Even before going full-screen, the content should fit on-screen appropriately regardless of the device viewed on, and it's acceptable for example, on a Droid, to have 50 pixels of extra space distributed between the top and bottom (if the game was designed from an aspect ratio perspective to fit on a Nexus One for example)

- If there is a desktop SWF and a mobile SWF, the user should be directed to the appropriate experience based on the device detected,
- If the user doesn't have FP 10.1 installed, then they are presented with the option to install/update the Flash Player 10.1, and acting on that option takes the user to the appropriate page in the appropriate marketplace for that device – all of which will happen automatically if the site redirects the user to get.adobe.com/products/flashplayer .
- No more than two taps (three if you could the initial selection tap) should be required before game play begins
- if Game > 400K, then a pre-loader should be part of the experience.

GAME ASSETS

- Assets can be loaded dynamically, especially if you are using any type of asset content management system,
- Assets designed and/or tweened in Flash CS4 or CS5 should be optimized for mobile, which means to simplify the vector shapes and reduce the number of on-screen elements and/or reduce the number of frames in an animation.
- Any links to external content (e.g. more games) should not attempt to launch content in new window, but replace existing content (might also consider leaving full-screen mode before referencing URL)

GAME-PLAY

- full-screen mode: go automatically to full-screen mode on selection, pause the game if leaving full-screen mode, and hitting resume re-invokes full-screen mode, is okay to leave full-screen if text input is required. Consider locking your content in landscape or portrait mode via the techniques outlined in the Flash Sizing Zen document.
- game 'cells' or anything that is tappable should be no smaller than the minimum hit area size for that class of device (see diagram, above).
- if porting an existing game, this means that the game might have to see a change in layout or have a simplified game board to make sense on mobile

- If the user must receive UI hints (what objects they can pick up for example, should use finger-down type hinting or (select-object-with-feedback then a secondary interaction specifies the action on that object) as opposed to mouse over hinting)
- Sound/audio — should be a way to mute sounds, but volume control should be un-necessary, sounds should be appropriate for mobile device (for example, one game has an explosion sound that just sounds like static on a phone).
- Ideally, lock in the orientation into either landscape or portrait (see Flash Sizing Zen for more information).
- Otherwise, if the game is designed for landscape or portrait, then the user should receive some type of prompting if they are in the least favorable orientation. The easiest way of doing this is to place some off-stage graphics to prompt the user to change the orientation.

Assessing the Difficulty of Optimization

The amount of time you should expect to spend on optimizing content for an average title for different types of activities.

- Resizing the game for mobile: 70%
 - Simplifying vector graphics and timeline-based animations: 20%
 - Re-laying out content that needs to be rescaled for readability or touchability: 30%
 - Dynamically loading content: 10%
 - Other rescaling activities: 10%
- Refining full-screen/sizing/orientation behavior including pause/resume: 10%
- Customizing the user-interaction model to the capabilities of the device: 5%
- Flash Player, Device Detection and Preloaders: 5%
- ActionScript changes for EnterFrame and Timer instantiations: 5%
- Miscellaneous: 5%

Resources

Rather than repeating good advice that exists elsewhere, this is a list of what I would consider imperative resources to check out. I would give them the same emphasis as if I had included them in this document.

Adobe : Mobile and Devices Developer Center

<http://www.adobe.com/devnet/devices/>

- Flash Player 10.1 for Devices Optimization Center

<http://www.adobe.com/devnet/devices/fpmobile/>

- Tools for Optimizing Web Content for Mobile Delivery

<http://www.adobe.com/cfusion/entitlement/index.cfm?e=fpmobiletools>

- Optimizing Performance for Mobile Devices
- Optimize Performance on Video, Graphics Hardware
- Omniture Media Tracking Implementation Guide
- Design Tips for Creating Mobile RIA's
- Author Mobile Content for Multiple Screen Sizes
- Detect Flash Player with SWFObject
- Deliver Video for Flash Player 10.1 on Devices
- Optimize and Deploy a Video Player on Devices
- Reference Video Player for Devices
- Profile Performance for Flash Player 10.1 in ActionScript
- Recommendations for Encoding H.264 video for Flash Player 10.1 on Mobile Devices
- Miscellaneous Source Code Examples
- Remote mobile device testing with DeviceAnywhere

http://www.adobe.com/devnet/devices/articles/device_anywhere.html

- Optimizing Performance for the Flash Platform

http://help.adobe.com/en_US/as3/mobile/index.html

Adobe Flash Platform

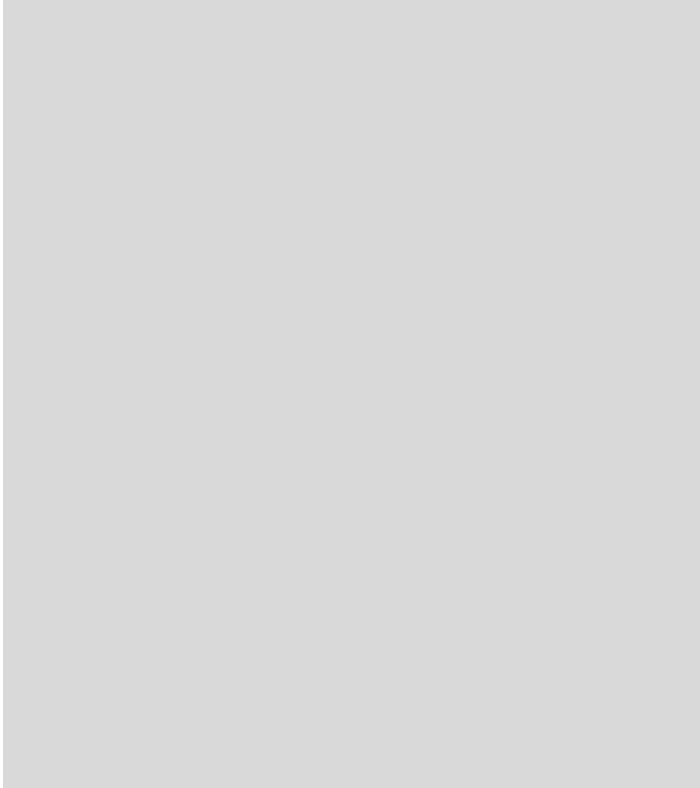
Mobile

[Optimizing Performance for the Flash Platform](#)

[Building ActionScript 3.0 Applications with the Packager for iPhone Preview](#)

Optimizing Performance for the Flash Platform

- ⊕ Introduction
- ⊖ Conserving memory
 - Display objects
 - Primitive types
 - ⊕ Reusing objects
 - Freeing memory
 - ⊕ Using bitmaps
 - Filters and dynamic bitmap unloading
 - Direct mipmapping
 - Using 3D effects
 - Text objects and memory
 - Event model versus callbacks
- ⊕ Minimizing CPU usage
- ⊕ ActionScript 3.0 performance
- ⊕ Rendering performance
- ⊕ Optimizing network interaction
- ⊕ Working with media
- ⊕ SQL database performance
- ⊕ Benchmarking and deploying
- ⊕ Legal notices



Author Once: Deploy Everywhere?

When moving beyond the desktop, where the ability of the system to express, size of the screens and the relative resources available to Flash are roughly equivalent, special considerations need to be made when deploying the same or equivalent content on devices that have a fundamentally different interaction model (mobile=touch interface, small screen and maybe a keyboard and whereas television input might be limited to a 5-button remote but offer up a large display area and fast h.264 decoding but at the same time being sluggish when it comes to animating content). As well, while mobile devices are challenging from a resource perspective, televisions are even more so.

What's not immediately clear is at what point offering up different SWF's becomes important. Does it make sense to offer up a different SWF for every single device or to re-leverage a single SWF universally? Certainly, both are possible and at the same time either extreme has costs associated with it.

The downside of offering a different SWF for each and every device out there is that the on-going costs associated with maintenance and updates increase significantly, as well as organizational issues related to source version control.

And while it's possible to offer a single SWF for all target devices, doing so could have consequences in the short-term. In the long-term, a lot of these issues will be resolved by the use

- Design-time optimizations that are made to satisfy mobile performance issues or real estate challenges won't translate well back into the desktop experience,
- Since the interaction models are fundamentally different, certain aspects which are helpful on desktop (such as rollover) actually make the usability worse on a mobile device and provide misleading visual cues to the user,
- The image assets that are appropriate for a mobile device (small, takes up less bandwidth) won't be appropriate for a high-definition 50" LCD. So, you're stuck with 3 options: embed small image assets and pixelate the heck out of them when scaling up, embed large image assets and scale the down, incurring the filesize bloat, or using an asset management system to deliver the right-sized assets,
- The best practice that appears to be emerging is to target one SWF for desktop, another for mobile devices and another for set-top boxes.

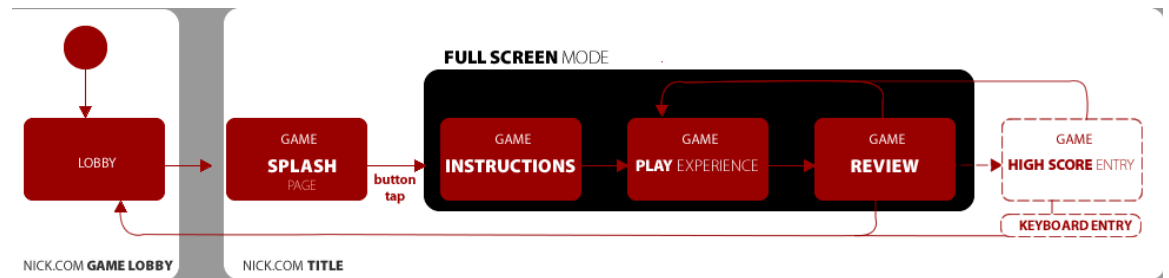
Gaming Display Recommendations

- There are two fundamentally different display/interaction modes for Flash on mobile devices
 - Embedded – content is likely to be more exposed to the potential for gesture ambiguity, D-PAD and trackball issues, as well as orientation and content sizing issues. This is the default display mode for Flash in the browser. Even if the user zooms into the Flash content, they will most likely be able to see some HTML content in the periphery.
 - Full Screen – similar to the same mode on the desktop, the user must initiate full-screen mode (by tapping a button within your movie). Flash will go full-screen, inform the user of the same (and some hint how to escape it) – but with the caveat that you basically can't use the device's keyboard (neither the physical keyboard nor its virtual one). In this display mode, only Flash content is displayed.

The user may inadvertently exit full-screen mode. Since game-play may become increasingly unpleasant in embedded mode, you might consider pausing the game.

My recommendation is that you present the game initially in the HTML context (you have relatively little choice here, as you can not automatically go into full-screen mode). This is a great opportunity for ad placement.





Once the user advances past the splash page (or alternatively, past the game instructions), you can, on a button click, go to full screen mode (assuming that `AllowFullScreen="true"` in the OBJECT/EMBED tag).

You can also make the content go full-screen automatically the first time the user attempts to interact with the content by setting the `fullscreenOnSelection="true"`.

User Interactions

Take into account the types of input devices that are available, and what's not available that you might be taking for granted.

- Mobile devices typically don't have a cursor and don't natively support the concept of hovering, therefore any games that rely on a button's "over" state, that use traditional hinting, or otherwise rely upon any type of "hover" functionality will need to be re-thought.
- Physical keyboard: maybe but not in full-screen mode
- Device's virtual keyboard: will only work with TextFields, not in full-screen mode
- cursor keys: yes if device has DPAD or trackball
- hold-to-repeat: maybe
- simultaneous keys: probably not
- accelerometer: only if mobile and only if AS3

- geo-location: not in the browser
- proximity detection: not for FP10.1
- Screen – yes
- audio – yes
- haptic feedback – not in the browser
- dedicated light indicators (such as trackball tri-color lights) – not in the browser

5-Point Input Controls

- The DPAD and trackball both emit arrow-based events: 37 (left), 38 (up), 39 (right), and 40 (down)
- Both controls automatically align with the device orientation. In other words, when in landscape, pressing the left-most DPAD button (or pressing trackball to the left) will generate 37(s).
- On the Motorola Droid, opening the keyboard tray forces the orientation to be in landscape, regardless of how the phone is physically oriented.
- The DPAD will emit a single `onKeyDown()` and `onKeyUp()` event for each button pressed. In the latest version of the Flash Player 10.1.61.69, pressing the CENTER DPAD button is the same as hitting ENTER.
- The trackball will emit multiple `OnKeyDown()`s and `KeyUp()`s for each thumb-swipe, thumb-zing, or whatever you want to call the action where you spin the trackball. It will approximate the vector of the spin. So for example, if you spin up at 45 degrees, you will probably see the events clustered together: up, right, up, right, up, right, up, right or if you spin straight up, you should see something similar to up, up, up, up, up (anywhere from 4-8 on average), pressing the trackball down is the same as hitting ENTER.
- I was able to confirm that AS2 logic using `Key.addListener()` is working as expected, used this URL for testing: http://flash-creations.com/notes/asclass_key.php

Accelerometer

The accelerometer is only available in AS3 applications. While it might be available in a future release for AS2, it won't be for 10.1.

Also, when contemplating how you might use an accelerometer in an application, consider that you can't lock content in portrait mode when in a browser. However, see an interesting hack in the orientation section, below.

You can specify the accelerometer notification interval – and upon each accelerometer event that gets generated, you'll get three numbers that correspond to acceleration in the X, Y and Z axes.

- Note that if the phone is perfectly level and is still, you should get values (0, 0, 1) – because the Earth is always exerting a gravitational pull equivalent to 9.8 meters per second² (1G).
- **muted** is always false for Android devices (no way to disable accelerometer)

Here's the bare-bones code snippet:

```
import flash.sensors.Accelerometer;
import flash.events.AccelerometerEvent;

var gravity:Number = 9.8 // 1G = 9.8 meters per second
var xAcc:Number, yAcc:Number, zAcc:Number;
var xVel:Number, yVel:Number, zVel:Number ;
var friction:Number = 2;

function init():void {
    if(Accelerometer.isSupported && !Accelerometer.muted) {
        var acc:Accelerometer = new Accelerometer();
        acc.setRequestedUpdateInterval(100); // in
milliseconds

acc.addEventListener(AccelerometerEvent.UPDATE,handleAccele
ration);

acc.addEventListener(AccelerometerEvent.STATUS,handleAccele
rometerStatus);
    }
}
```

```
function handleAcceleration(event:AccelerometerEvent):void
{
    xAcc=event.accelerationX;
    yAcc=event.accelerationY;
    zAcc=event.accelerationY;

    xVel=calculateVelocity(xVel,xAcc,gravity,friction) ;
    yVel= calculateVelocity(yVel,yAcc,gravity,friction) ;
    zVel= calculateVelocity(zVel,zAcc, gravity, friction)
;

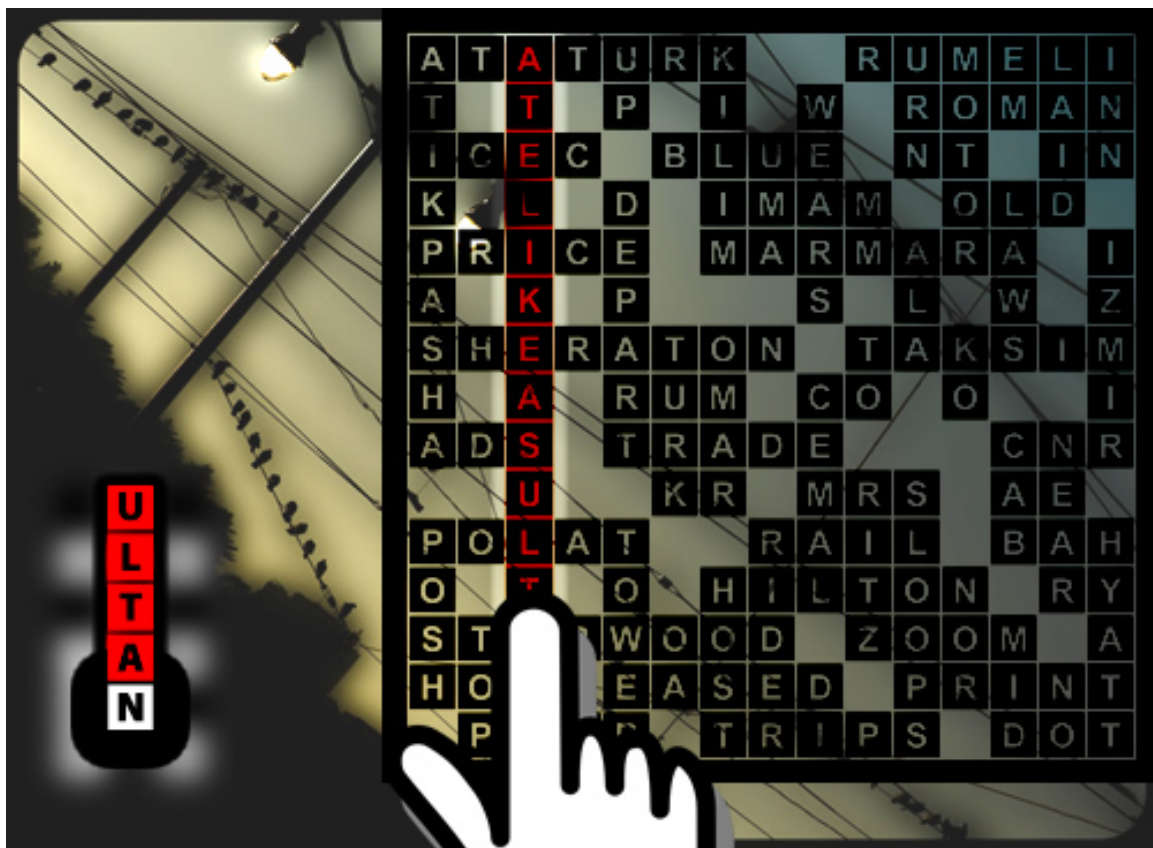
    ...
}

function
handleAccelerometerStatus(event:AccelerometerEvent):void {
// double check the isSupported and muted flags
}
```

Do's and Don'ts

Drag and Drop

- Don't require the user to drag objects that are small or require that their placement be precise – especially in the absence of additional feedback than that which is typically offered on the desktop.
- Do use an auxiliary zoom lens that shows what the content under the finger looks like.



- Do use visual and/or auditory feedback to let the user know that they are either over a valid drop area, or where the object's placement would be if they let go immediately.

Rendering Performance

(note: this section should be compared with the content found in the Resources section, above – some content may overlap)

- Hardware accelerated rendering not always available, and is required for pixel bender to work
- Use standard and alpha blend modes -- but avoid others, particularly layer
- Avoid morphing shapes as these have to be re-tesselated on each frame (expensive).
- Complex or finely articulated shapes are more expensive to render than large simple shapes
- Avoid large changes in scale
- Avoid over-drawing (when relying on hardware acceleration) although it's fine when rendering using software.
- set background color of stage instead of putting a large rectangle in the background.
- Avoid default bitmap fill mode
- Simplify vector graphics
- Use CacheAsBitmap for content that doesn't change frequently but avoid it for content that does change frequently.
- Avoid filters
- Carefully picking bitmap sizes if content will need to be sized down....**mipmap rule**
- but ideally, the content would always be 'right-sized',
- turn smoothing on if reducing image scale (does impact performance)

Data Structure Management

- Old School Polygonal Data Structures (available as source):
<http://code.google.com/p/polygonal/downloads/list?can=4&q=&colspec=FileName+Summary+Uploaded+Size+DownloadCount>
- Consider using polygonal labs data structures --highly optimized
- Newer Highly-optimized polygonal data structures (only available as a SWC):
<http://code.google.com/p/polygonal/>
- Object Pooling: refer to <http://lab.polygonal.de/2008/06/18/using-object-pools/>

Game Logic, Event Notifications & Sequence Management

(note: this section may overlap with content found in Resources)

There are two common performance sinkholes in Flash content:

- onEnterFrame event listeners – it is not uncommon for there to be a lot of unnecessary processing that happens on each frame, which can quickly become very CPU expensive.
- Timers – timers are also frequently abused. Try to keep the number of Timer objects to a minimum, be preferential to using them in objects that are likely to be around for a while (for example, shy away from creating a Timer class for each game avatar, as tempting as it might be). Challenge yourself to see if you can accomplish everything you need with a single Timer class instantiation.

Recommendations

- Implement a Timer manager class
- Single Timer instance
- Multiple Timer subscriptions -- but guaranteed to fall on even boundaries.
- Timer resolution is configurable
- Organize timers in groups so that they can be managed as a group.
- Consider replacing object-based event listeners with iteration loops.
- Avoid using many timers
- onEnterFrame vs. Timer
- Dynamically throttling the frame rate
- If animating multiple objects, perform delta operation in a single pass

For example: in a re-make of the Asteroids game, you could create the ship and four big asteroids (n=5) -- placing a Timer object on each game object. When each Timer emitted a tick you could update the position of its parent. However, that results in [n] separate rendering events. It would be more efficient to make a single rendering pass in order of dependencies (if dependencies exist) updating all of the object positions and triggering just a single display refresh. This should result in a higher effective frame rate.

- Consider using an open source Tween library, such as TweenLite or TweenMax
- Use event notifications to react to keyboard events, don't create loops that check to see if a certain key is pressed or not.

Coding Recommendations

(note: this section should be compared with the content found in the Resources section, above – some content may overlap)

Remember that the best practices for developing a game are not always going to agree with, and in fact will sometimes be in contradiction to best practices for other types of applications.

There is a natural tension that exists between some of the recommendations in this section that seek to optimize run-time efficiency and other, equally valid forms of efficiency, such as the amount of time it takes to write, troubleshoot, and maintain the code. Well-accepted OOP practices such as black-boxing functionality may have long-lasting payoff, but there is a small yet noticeable performance tax that results.

However, one best practice is always recommended, regardless – and that's to **use strict (or strongly-typed) AS3** in your projects. Not only does it make troubleshooting easier but because of the way Flash's virtual engine works, strict AS3 code actually runs more efficiently than its counterpart AS2 or loosely-typed AS3.

Referencing

- Eliminate weak references: e.g. `asteroid["velocity"]` should be `asteroid.velocity`;
- Coerce untyped data (such as configuration data from an XML file) into the anticipated datatype before attempting to access.
- If using a data element more than a couple of times, use a local variable to bypass the cost of a lookup.
- replace explicit or implicit Object datatype instantiations such as **`asteroid = {x:x_pos,y:y_pos}`**; with a more explicit class definition: `asteroid = new Asteroid(x_pos,y_pos,asteroid_type)`

Arrays

- use ByteArray instead of Array for byte/short/int values.
- if you store value objects in an array, coerce them back into the original value object class before attempting to access any of its members, for example, replace `x_pos:int = asteroids[i].x;` with
`x_pos:int = Asteroid(asteroids[i]).x;`
- Better yet, if all of the elements in an Array are the same data type or same class, then use the Vector class.

Comparisons

- use bytes and bit operators to manage game object boolean states (canBeDestroyed, absorbsShot, activelyFiring, isCloaked, etc.)
- Use hard-coded constants in lieu of the 'const' data type: while it decreases the readability/maintainability of the code, it is also more efficient. (public var type:uint =3; instead of public const FORCEFIELD_TYPE:String="standardForceField"; as the latter form results in a string comparison (comparatively expensive)

Math

- Use long-hand notations for functions such as `Math.min()`, `Array.push()`, `Array.shift()`, etc. to avoid the function call penalty:
 - replace `var n=Math.abs(n)` with `if (n < 0) n = -n;`
- Use `a++` (four times) or `a+=4` in lieu of `a=a+4;`
- Use `a=a-4` in lieu of `a--` 4 times or `a-=4;`
- Use multiply instead of division (`b*.5` instead of `b/2`)
- Shift left and shift right in cases where you need to multiply or divide by a power of 2
- Coerce to `uint` instead of using `Math.floor`, and if appropriate `int` is even faster.
- Recast `int` calculations as Flash runtime will sometimes cause the outcome of calculations to become a `Number`.

```
var x_pos:int;  
...  
x_pos = x_pos * acceleration;
```

should be

```
var x_pos:int;  
...  
x_pos = int(x_pos * acceleration);
```

- Instead of

```
y = Math.sin(Math.PI/180*i);
```

use fast inline trigonometric functions

<http://lab.polygona.de/2007/07/18/fast-and-accurate-sinecosine-approximation/>

Iteration

- Replace for loops with while loops. For example, instead of:

```
for(var i:int=0;i<1000;i++) { data[i]=func(i);}
```

do

```
var i:int=0;
while(i<1000) {
    data[i++] = func(i);
}
```

- instead of

```
var v1:Number=10;
var v2:Number=20;
var v3:Number=30;
```

do

```
var v1:Number=10, v2:Number=20, v3:Number=30;
```

- Remember that **removing a child from a display list** (e.g. `stage.removeChild(demon);`) **neither destroys the object nor does it stop the object from consuming resources**. So adding and removing asteroid sprites that have an `onEnterFrame` listener or use a `Timer()` object may continue to consume resources until the timers are stopped/destroyed and you remove the `EnterFrame` listeners that may have been created, even if all explicit references to the object have been lost.
- Replace anonymous function closures with method closures. That is, replace

```
var f:Function = new function(event:Event) {trace("handler");}
```

with

```
function f(event:Event):void {
    trace("handler");
}
```

- haXe is a potential solution to the inherent conflict between performance and maintainability of a game: <http://haxe.org/doc>

HTML Considerations

See **Flash Sizing Zen** document as well as **Player and Device Detection recommendation** document.

External Recommendations

Tip: When applying a blur filter with ActionScript, using values for blurX and blurY which are powers of two (such as 2, 4, 8, 16, and 32) can be computed faster and offer the benefit of a 20–30% performance improvement.

One trick I learned while making some flash games is that a higher framerate on the fla file does not translate into smoother gameplay. In fact, you want the framerate as low as possible so that the main loop has adequate time to compute. I found a solid 18 fps actually looks better visually than an inconsistent 24-30 fps.

Following is from:

<http://www.bigspaceship.com/blog/labs/flash-performance-tips-part-i/>

Masks Are Bad

Well, they're not all bad. Masks can be exceptionally useful, as we all know by now. They're the #1 performance killer though. When you mask something, you force the player to decide what needs to be rendered and hidden every single frame. So how do you get around without masks? With a little bit of patience and tricky layering (such as making the background the foreground with a giant hole cut out for where your viewable area is) you'll be able to get the same end result without the performance hit.

Alpha PNGs and Video

Same deal as the masks. Sometimes its unavoidable, but you're still asking the player to figure out what to render under the alpha. Sometimes we'll make an alpha video at half size and then scale it in Flash. You'd be surprised how good it still looks.

Oh, and with regards to alpha video: Try experimenting with PNG sequences in place of the alpha video. Video usually performs a little better, but its always worth looking into.

Frame Rate

Despite all discussions otherwise, there is no magic framerate. We use 25 or 30 because (as far as I know) we like it best. At some point we tested and determined one was slightly better than the other, but generally speaking this is not going to be the primary cause of a site running slow. I wouldn't generally advise going higher than 30 though, just because you're asking the player to render an awful lot awfully fast...

cacheAsBitmap and BitmapData

Where possible use `cacheAsBitmap` to rasterize vector items. You'll force Flash to draw the symbol one time and then never again. On the flip side, if you're scaling or rotating a symbol NEVER set it to `cacheAsBitmap`. Then you force Flash to render AND recapture the raster every frame, making the process slower instead of faster.

With sites like the [Da Vinci Code](#) and [Nike Air](#), we would take a dynamic screenshot of the section, draw it into an empty movieclip and then manipulate the screenshot to animate it out. This is far, far faster than animating many elements out, or animating over top of many elements. I highly recommend this practice.

`_visible` is better than `_alpha`

`_alpha = 0` and `_visible = false` are totally different things. Alpha determines the opacity of a clip. Visible determines whether or not the clip should actually be rendered by the player. If you're setting something all the way invisible, use the `_visible` property.

`onEnterFrame` and `setInterval`

When you're through with these processes, clear them from memory with `onEnterFrame = null`; and `clearInterval(myInterval)`; respectively. Leaving these around when you're not using them is like leaving the telephone off the hook when you're done with a call.

Pre-define your math

Got a sine wave you're about to draw? Is it the same sine wave every time? Hard code the numbers into an array. By doing the math for Flash, you're saving some complex processes in advance. I even experiment with using a Tween that I `nextFrame()` thru to get all of the entries into an array before hand.

Game Persistence

The following questions are obviously true for AIR, but it's interesting to think about whether or not the same issues apply to in—browser content.

"What these results show is that people want mobile games to be 'pick up and play', and use them to fill time, as opposed to committing a large amount of time to playing them. It is therefore important that game designers facilitate this method of game playing, and make it easy to pick up and play games. The main elements of this are:"

- *fast start up time (from app load to actually playing)*
- *low number of extraneous menus to navigate before playing*
- *app resumes from where it left off*
- *app shuts down quickly, but doesn't lose progress*

-- <http://www.stevebromley.com/blog/2009/10/23/mobile-games-should-start-quickly-lets-get-down-to-business/>