Performance Tips and Tricks

(Need to supply some code samples that show the problem and the solution)

Performance Areas:

- General
    - Turning data into class files
    - Binary CSS
    - Use fewer nodes
        - Describe the process that occurs for every node:
            - Picking
            - Sync
            - Bounds computation
            - Computing dirty regions
            - Visits when laying out
            - Visits when applying CSS
            - Embedded maybe 1000 nodes max, Desktop 10x that
                - Probably 200 on embedded, 20,000 on Desktop should be very fast
        - Less memory overhead
            - Each Node concurrently costs 5-7k
                - In part because of state copy for multi-threading
    - Execute less code
    - Reduce method calls
        - Sometimes this is "free", and on desktop it doesn't matter as much, but embedded (and iOS) makes this more important
    - Use FXCollections API
    - Avoid structural changes (compute bound)
        - Lots of work to discover if a SG change is "legal", such as avoiding loops etc.
    - Visibility changes
        - Presently we do a downward tree-walk to set "treeVisible" state on each node
- Threading
    - Do minimal work on the main FX thread
    - Can create (most) things on a background thread
- Graphics
    - Need document to specify how Prism works
        - What OpenGL gets generated
        - What is the path used for rendering
        - What do we do when we need to render images
    - Text, text measurement is always slow.
    - Images are faster than shapes
        - Shapes require rasterization (in CPU!) and passing a mask to the GPU
        - Bounds of the shape to be rasterized has a big impact on performance
        - Images that are not changing are about the fastest thing in the universe
    - To cache, or not to cache? That is the question!
        - Every time the thing being cached changes, it is very expensive to redraw it.
        - But reusing the baked image a zillion times is faster than redrawing!
    - Load smaller images when you don't need a full sized image
        - Loading a single 12mega-pixel image on PI can run out of memory
        - Memory management flags for texture management should be API
    - PI Specific: Configure platform to split between GPU / Main memory
    - Limit the use of things that require rendering into intermediate textures (such as non-axis aligned rects or random shapes or Node types for clip, opacity on things that aren't leaf nodes, effects, blend modes)
    - Look at fill rate
    - Reduce overdraw
    - Be aware of how dirty regions work (maybe opportunity to turn it off?)
    - Explain how region image caching
        - When do we do it, and when do we not
- Animation
    - Specify the use of cache hints during animations
        - DEFAULT
        - QUALITY
        - ROTATE
        - SCALE
        - SCALE_AND_ROTATE
        - SPEED
        - Under certain circumstances such as animating nodes that are very expensive to render, it is desirable to be able to perform transformations on the node without it having to regenerate the cached bitmap. An option in such cases is to perform the transforms on the cached image itself, using one of the CacheHints.
    - Specify the Interpolator for a KeyValue in a Timeline
        - Using Interpolator.EASE_BOTH can provide a much nicer looking animation that looks smoother to the eye than the default LINEAR.

- UI Controls
    - Smart customization: no need to extend from the Control class:

        - Need more functionality than provided by an existing control

        - Not necessary to provide as a library control
        - Extend Layout container and CSS styles
- CSS
    - Avoid selectors that have to match against the entire set of parents
    - Use stylesheets not setStyles
    - Use pseudo-class state, not multiple style classes, for state-based styles (FX 8)
- FXML
    - Proper use of FXMLLoader#load
        - The load() method of FXML loader is a tricky method. Make sure you are not calling the static one. The next code is wrong – it calls static method load() which has nothing common with your fxmlLoader object:

```
FXMLLoader fxmlLoader = new FXMLLoader();
Parent obj = fxmlLoader.load(getClass().getResource("MyApp.fxml"));
Object invalidController = fxmlLoader.getController();
```

Correct way:

```
FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("MyApp.fxml"));
Object obj = fxmlLoader.load();
Object myController = fxmlLoader.getController();
```

How to find performance problems ...

Random (need sorting):

- Manual state sorting in the scene graph
- Making a fast Cell?
- Animation: Control bitmap caching
- Animation: use new API for smooth animations (text rendering hints etc)

Pulse logger should keep track of which cached images are invalidated when.

From John Smith:

> *The biggest thing when doing performance work is identifying the benchmarks. Once we know what we're measuring, it *will* get faster.*

This is one of the most difficult things I found about trying to code performance sensitive stuff for JavaFX. It's the not knowing part of it.

JavaFX features high level features such as effects, css and animation and it's hard to know where performance bottlenecks will be without trial and error.

For instance, you can draw hundreds of thousands of lines really quick, but if you try to draw a path with more than 10000 elements, things start rendering slow - so you can speed your rendering up by using lines rather than paths, but unless you know that or try it, you might get stuck.

Or another instance is selecting the wrong pixel format for a WritableImage can kill the performance of trying to animate a video by twiddling the image's pixels because the frame-rate drops an order of magnitude without the right pixel format.

I also find it hard to know the impact of something like effects or CSS on the GPU or battery life, because it is pretty difficult for me to objectively measure those kind of things.

Should I set up lots of parallel animations, or am I better off having a central pulse style system which does everything on a tick? Without knowing how animations are implemented, e.g. if they use their own thread or if they incur a bunch of other overheads, it's hard to make an objective decision about that.

If your application uses WebView with intensive JavaScript, then you are better off using a 32 bit jvm on windows rather than a 64 bit one, because one will use a JIT JavaScript compiler and the other won't.

The type of effects used make a large performance difference. For example, boxblur is a whole lot quicker than a gaussianblur - as it's supposed to be I guess from reading the wiki pages on what the algorithms incur.

Buffering of canvas commands, and a subsequent pause while initially rendering a canvas with lots of commands can introduce pauses to the application that most api users aren't going to know about until it starts occurring.

I am sure there are many more similar performance impacting tradeoffs which could have been listed here that I don't really know about or understand (such as whether I should rely on dirty region heuristics or node cache hints to optimize rendering performance or should I just snapshot the nodes myself and use the snapshot rather than relying on a platform optimization).

I think a lot of the above is just the nature of JavaFX and goes with the territory - it's a relatively high level library which abstracts you from some of the low level implementation details so that the true cost of some of your api usage choices are hidden from you.

> *Did we try turning cache to true and cache hint to SPEED?*

A simple game I wrote used some basic animation of about 50 nodes with effects applied to them (translucent, blur, sectioned viewports into a large Image) and without caching ran painfully slowly. Setting caching to true and using cache hints (just on the animated nodes) made a massive performance difference (on a macbook air), it was the difference between a game which was playable and a game which was not (i.e. framerate did not drop to single digits and the air's fan didn't spin up).

Turning on caching for select nodes was the easiest and single biggest performance improvement I got for the game.

> *We had an embedded hack-fest a couple weeks ago in which performance on desktop went from 320-800+fps on table view scrolling, which in large measure came down to reducing the number of state switches on the graphics card (and the resulting decrease in the number of OpenGL calls).*

I realize the above statement is to do with internal optimizations, but should I, as a user of the JavaFX API ever have to worry that the way in which I write my user code may result in something like an increased number of state switches on the graphics card?

Or should I just be able to ignore that kind of stuff as an implementation detail, kind of like when I drive my car, I press the accelerator and it goes and I don't really need to worry much about how that happened?

The difficulty for me here is that I don't know what a state switch on a graphics card is and have no way of knowing whether a particular code path is triggering a lot of switches.

It seems like an aim for JavaFX is to not require the developer be a low-level mechanic to make things work.

> *We should never require you to have to follow a 15 point performance plan just to get acceptable performance, or to avoid choppiness*

Nevertheless, an official performance guide would be useful (like Android's: http://developer.android.com/training/best-performance.html).

I put together a short (and necessarily incomplete) guide as part of an answer to a stackoverflow question on obtaining good performance with JavaFX: http://stackoverflow.com/questions/14467719/what-could-be-the-best-way-to-view-millions-of-images-with-java

Thoughts and questions from Zonski:

1) turning cache to true and cache hint to SPEED I haven't seen explained what the drawbacks are to doing this. I assume there is some trade-off for turning this on otherwise it would be on by default?
2) AnimationTimer - how is this related to transitions (like TransitionTimer). They seem to be two very different things - not just at the usage level but the actual underlying performance and implementation - but it's not clear to me exactly the difference. I wanted to work with both AnimationTimer stuff and Transitions in a consistent way (e.g. pause one thing to pause my whole app). When trying to use the API my instinct was that the transitions used an underlying AnimationTimer, so I was looking for a way to setAnimationTimer on the transitions, so they all shared the same instance. I'm obviously seeing it wrong, but it's not clear to me why these things are so different.
3) If I have an app with multiple views (say 20 or 30) and I want to animate between them (slide in, out, etc). Should I have all the views already added to the scene and then just move them off-center (and/or make them invisible), or should I add and remove them at the start and end of each animation. I'd assumed that adding and removing was the way to go for performance, is this correct? The question really is: If a node is in the scene graph but not visible does it add much overhead to rendering, picking, etc? Doing something like Mac's Mission Control would be much easier if I didn't have to add/remove and could just zoom, but is this a bad idea from performance perspective?
4) Is there much of a hit in building a view full of nodes and then throwing them away and rebuilding on each showing of that view? In my apps (browser style) I would build all the views on startup and then reuse these by loading different data into them (e.g. the same instance was used for showing Person1 details as Person2, just loading different values into it). I thought this would be the best from performance, but there was a comment from Richard a while back that suggested it was perfectly ok/performant to throw away and build the page again on each showing? This would be much easier from an animation point of view as animating the transition 'back' or 'forward' between two pages of the same view is problematic when you are reusing the same view for both!
5) Is there much overhead to a view that it is in memory but not actually added to the scene. i.e. if I do end up building a new instance of the view for every "page load" (as per question 4), then it would be extra handy to just keep those views in memory so when the back button is hit I don't have to rebuild them. I assume the memory would build up pretty quickly if there were large tables, images, videos, etc? How does a web browser manage to cache all these pages in history and would the same approach be performant for JFX?
From Pedro:
- use layouts whenever possible instead of binds for laying out nodes
- reduce use of effects. If the effect is static use images instead of
effect
- In javafx 1.3 it would cost more to use stroke instead of a fill. For
instance if you have a rectangle with a stroke with would be more efficient
to draw 2 rectangles with a fill, so the other one would be used to produce
the stroke. Don't know if this still applies.