Optimizing client side performance

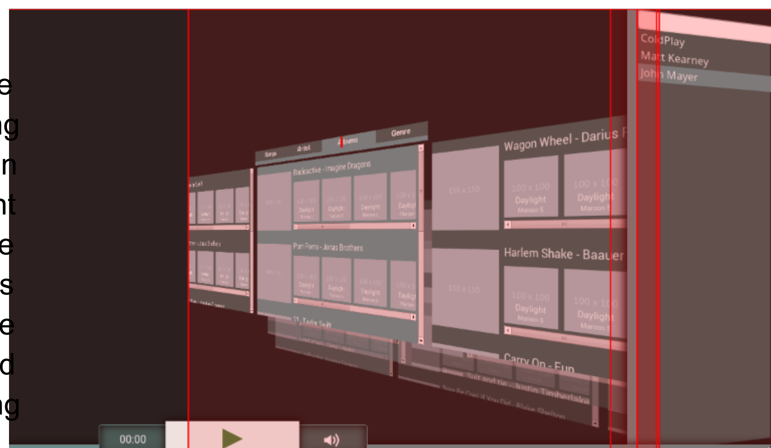
As the main theme of the application is the ability to provide high density information using 3D elements, we felt that a large portion of how favourable the user experience is deals with how smooth the user interaction is with the application. Our main goal with this performance tuning experiment was to find a way to provide lag-free interactions and animations.

Anyone who has done any rasterization algorithms, either being line drawing, text rasterization, anti-aliasing etc., knows that redrawing of any type is expensive, even if done on the GPU. Most window-management systems, including browsers, can smartly evaluate exactly which regions of a particular window needs to be redrawn, to avoid unnecessary rendering. These regions are called paint rectangles, and a visual cue as to what is drawn at any given moment in the browser can be activated using the *Show Paint Rectangles* option in Chrome. Once enabled, you should be able to see red rectangles around regions which are actively redrawn.

Rendering

☑ Show paint rectangles

The below screenshot is from our initial prototype, which does not contain any rendering optimisations.



The screenshot is taken while performing the animation for opening the item list on the right. As you can see, there are a lot of paint rectangles being used while performing the animations. This means, that for every frame of the animation, images, text, colored DIV’s and so on are redrawn, leading to a very jittery animation.

In this short technical overview, we will talk about how we managed to reduce the number of paint rectangles used during animations, and how we created a much more fluid experience.

Reducing Content

One obvious solution is to reduce the actual content being drawn in paint rectangles, without actually sacrificing information density.

From our experiments, the biggest source of lag were the songs being displayed for a search result, along side the text which is being displayed under each image. As the images themselves are being sent to us directly by the LastFM API, we had no control over the quality and compression rate of the images, apart from image size. We decided to go with two different image dimensions to display in the search result page. The larger images are reserved for “first-class” results, being those which match the searched query. The smaller images are

reserved for similar songs matching “first-class” results. This helped reduce lag considerably, as it resulted in a reduction of the amount of image information being rendered.

Another solution to reducing the content being rendered, was to reduce the number of actual results being returned from the search query. We kept the minimum amount of songs needed to fill a search page, and added several buttons to dynamically load additional results.

Loading of additional results would still cause jitteriness after a certain point, but this way the user is aware that he is sacrificing performance for additional information, as he sees a gradual degradation in animation fluidity while loading additional results.

Layering

One performance improvement technique introduced in recent modern browser, is the process of caching the result of rendering certain HTML elements in caches on the GPU, called *layering.*

A layer is essentially a static texture containing the rendering result of HTML elements, which can then be used as a graphical representation of these elements to apply various transformations, with high performance on the GPU. Think of them as *screenshots* of your web page, or regions of the web page. You can then use these screenshots to create various transformations and animations on the original HTML elements, without transforming or rendering the actual elements.

Layers are a very powerful tool, as they can be uploaded to the GPU directly as textures, and then animated with various transformations, which are incredibly fast compared to rastering the initial HTML elements.

The fastest way to do pure CSS animations is to force the browser to separate HTML elements, which you know will be animated independently, into separate 3D layers which can then be hardware accelerated. If done right, no rasterization of HTML elements will be performed, and no paint rectangles will be shown while animating.

One way to force the browser to separate elements into separate layers is to use transitions, more specifically using the CSS property *transition (-webkit-transition).* This way, the browser can figure out which elements will be animated and can separate them into layers accordingly. Compare this to the classical jQuery animations, where the properties of HTML elements are animated through pure JavaScript, where the browser has no way of knowing that you are doing actual animations, it becomes obviously clear why transitions are the preferred way of doing animations.

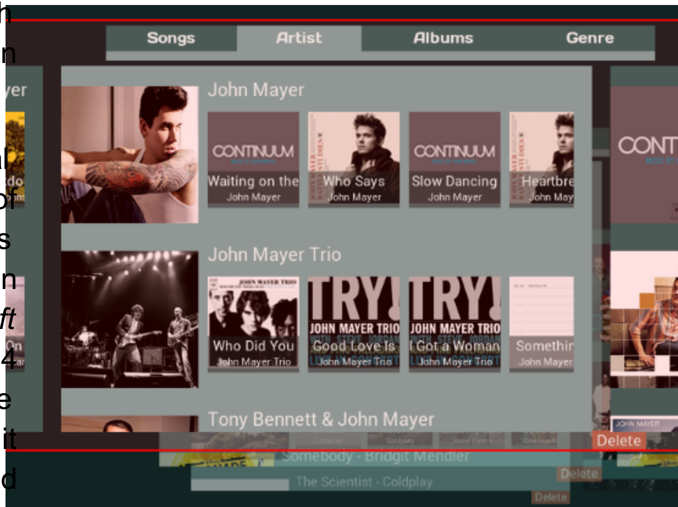
Pure transitions do have one particular shortcoming. Layers are created for each animation begining, and are then stamped back to the main page when the animation ends. This means, that if you only use CSS transitions, you will see a paint rectangle at the beginning of the animation, when the layer is created, and one when the animation ends, when the layer is blended back into the main page layer.

A solution to force the browser to dedicate a layer to a group of HTML elements, and avoid the beginning and end rendering, is to use 3D transformations, more specifically the *transform3D (-webkit-transform3D)* CSS property. If done right, you will see a complete lack of paint rectangles, either at the beginning, end, or during the animation.

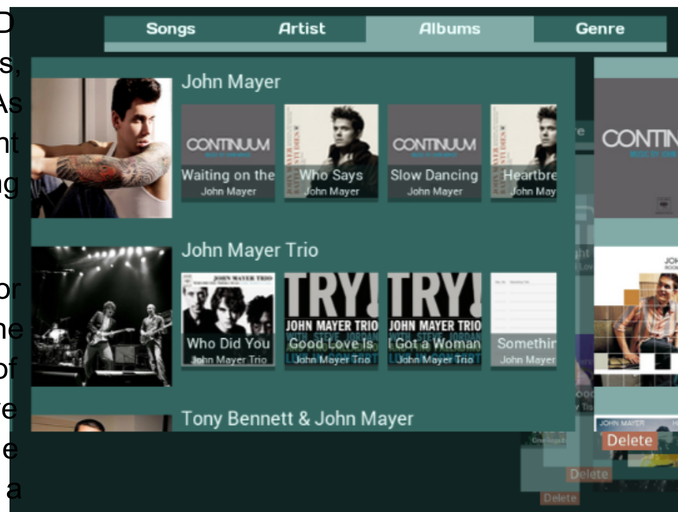
This principle of layering is the basis of all animation improvements brought to the application. We present below a few case studies of concrete examples of where we used layering.

Case Study: Search Page Scrolling

One place where we used layering was with vertical and front-to-back page scrolling in the Search section.



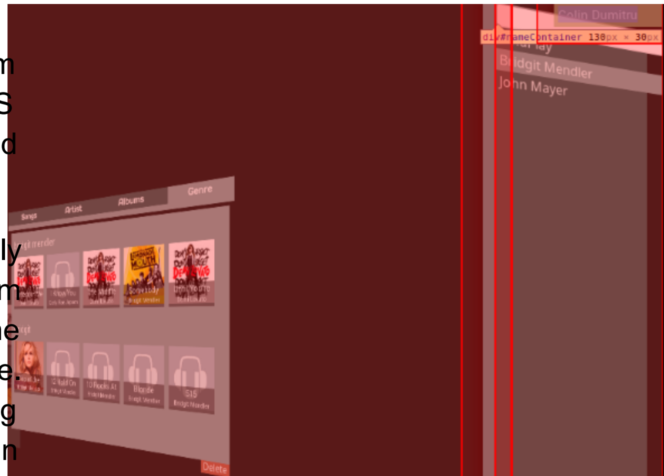
The first screenshot is from our initial implementation of the horizontal scrolling of search pages. We used overflowing tables inside a single narrow DIV, and then animating using jQuery the *margin-left* property to horizontally scroll the entire 4 search pages. As you can imagine, as the animation was done using pure JavaScript, it was very jittery. You can also notice the red paint rectangle which enveloped the entire four pages. This was very wasteful, as no content in any of the four pages was changed, yet they were redrawn for every animation frame.



We moved the animation to using 3D transformation, transitioning on the X axis, and the animation improved significantly. As you can see on the second image, no paint rectangles are seen while animating, making for a very smooth animation.

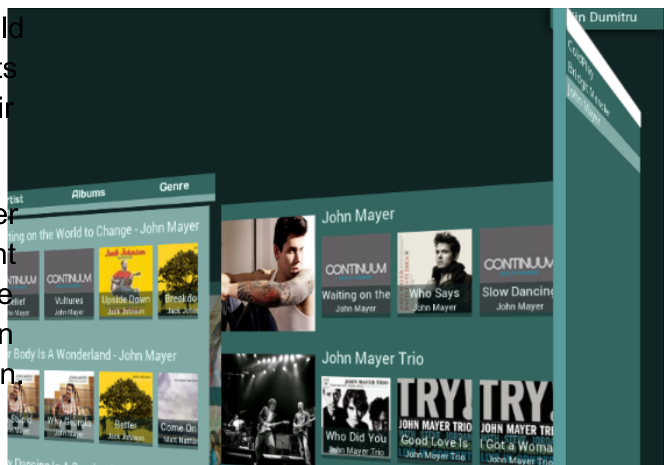
We have noticed however one graphical error with this technique. If you look below the active search page, you can see a section of the background pages missing. We believe that this is an issue with the layer rectangle not being calculated properly, copying a section of the static background while animating, and not refreshing the background properly. This issue is only reproducible on Chrome, Firefox displays the animation correctly. We plan to add a separate option to revert to the legacy animation model, if the user sees this graphical anomaly too distracting.

Case Study: Item List



Another area where we migrated from JavaScript animations to pure CSS transitions, is the item list on the right hand side.

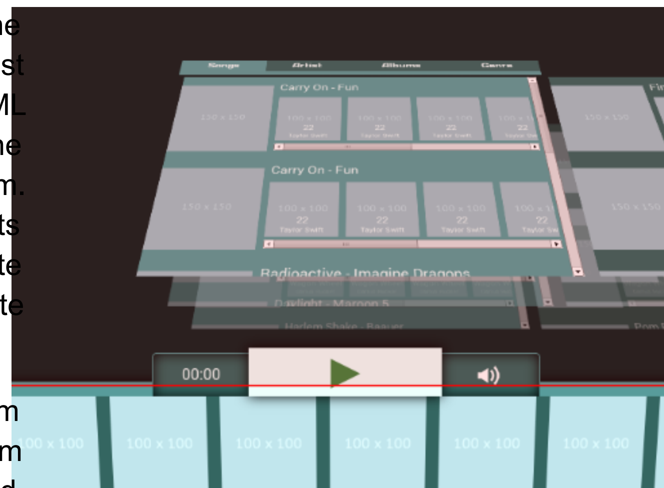
As you've probably guessed from the largely red image on the right, expanding the item list caused a very large amount of jitter in the application, as it is very rendering intensive. Not only the item list itself was being redrawn while animating, but also the main content area in the middle.



In an ideal situation, no re-rendering should be done while animating, as the elements itself do not change, just their transformation.

In the second picture you can see that after correctly setting up layers, all paint rectangles are gone. Just the pre-snapshotted images are transformed on the GPU leading to a very smooth animation.

Case Study: Now Playing List



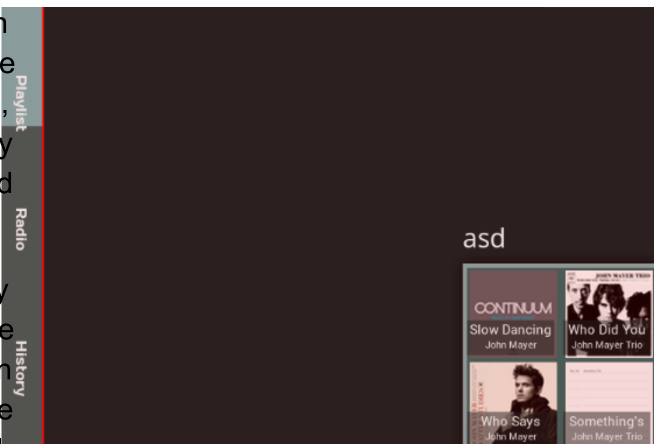
Similar to the Item List above, the transitioning effect for the now playing list involved animating two groups of HTML elements: the content in the middle of the page and the now playing list on the bottom. This means that both participating elements needed additional 3D transforms to animate independently from each other, in separate layers.

The first image on the right was taken from the prototype. As you can see, the bottom bar had no actual paint rectangles displayed, as it already used 3D transforms for the rotation effect. The largest paint rectangle however, was present on the main content area, which, once again, didn't need to be redrawn as the content itself does not change while animating.

The result of applying layering can be seen on the right. As can be observed, once again, no paint rectangles are seen during the animation, thus no actual rasterization is performed, leading to a very smooth animation.

Case Study: Section Scrolling

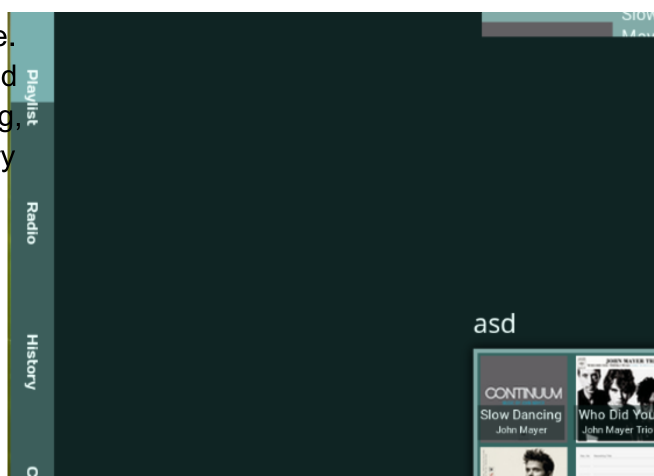
The last region of the application which benefitted from splitting elements into layers, was the section switching animation and scrolling, more specifically the menu bar on the left.



The screenshot on the right was taken before we applied the optimisation technique described in this paper. As you can see, paint rectangles can be observed not only around the menu bar itself, but also around the content area in the middle.

The scrolling effect was done manually using JQuery animations, by overflowing the content area in the middle, and then animating the *margin-top* property of the containing DIV element. As you can probably guess by now, the animation itself was very lag-prone, especially if a high amount of content was being displayed in the main content area.

The final result is seen in the second image. Not a single paint rectangle was recorded during, before, or after the animation, making, likewise the rest of the case studies, for very smooth animations.



Conclusion

As it stands now, we feel that the biggest change you can do to optimize page performance, after the initial page load, is to correctly identify and apply layers to the web application. This lend us a huge leap in performance, which is most apparent on low end machines.

It also made the web application usable on mobile devices and tablets. We will still continue development on the more optimized mobile version, but, as it stands now, both versions are completely usable on high end mobile devices.

You can always view side by side comparison, with and without layering by visiting the prototype page which doesn't have this layering technique applied, and the current development page with the latest improvements.