90% Easy, 10% Impossible

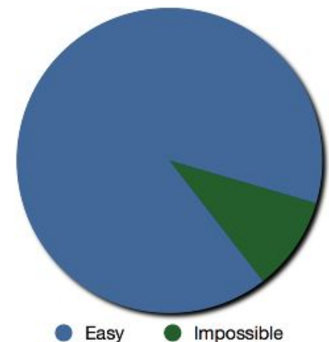
Source URL: http://fdiv.net/2008/09/13/90-easy-10-impossible

[1]

Posted by cwright [2] on 2008.09.13 @ 20:19

Filed under:

Antisocial Story [3] Apple [4] Cocoa [5] Not Apple [6] Social Story [7]



From time to time, I get these insatiable urges to read what other Cocoa developers blog about. Sometimes they're informative, sometimes they're funny, sometimes they read like college textbooks, and sometimes they're just downright terrible, but I read them anyway. It comes and goes in waves, every 2 or 3 months. A couple weeks ago, one such binge happened, and I started reading Aaron Hillegass' critique of NSController (since I was hating it at the time, and wanted to feel justified in hating it). In the critique, the following statement was made:

> "I used to use PowerBuilder, but it made 90% of the application easy to write and 10% impossible."

(From http://www.cocoabuilder.com/archive/message/cocoa/2003/10/26/75323 [8])

This statement was made to illustrate how some other class provided better functionality (NSAssociation), and how NSController was reverting back to how things were with PowerBuilder.

Now, I have to admit: I've not really used PowerBuilder, NSAssociation, or NSController all that much. At all, for that matter (I eventually hacked around bindings without needing NSController at all). But the statement stuck with me.

Lately I've been feeling the same way about many of apple's frameworks. They're handy, don't get me wrong. Learning and developing in Cocoa has been a dream come true – no other environment has been so personally satisfying for me, ever. Period. But there are some seemingly stupid omissions that make me wonder what's really going on, what the engineer was really thinking… were they in a hurry? Did they even think of this one totally legitimate use case? Is this specially crafted to keep me from doing things that "they" don't want me to do?

ColorSync is supposed to be Apple's platform-independent color management system that provides essential services for fast, consistent, and accurate color calibration, proofing, and reproduction. However, it doesn't seem all that consistent (with QuickTime, at least), and I've not seen it used anywhere outside of Mac OS X. I would also argue the "essentialness" of it – for me, it has been nothing but infuriating (Why the hell is this specific Codec cooking my colors!? Why can't I just set a "Don't F'ing touch my colors, Idiot!" flag somewhere, and have things "just work"?)

smokris often asserts that color management is actually an essential part of many workflows. I secretly agree, under the following conditions: That color data comes into my computer from an external digitizer. After conversion, it should be in a device-neutral format that I can do whatever I want with. And when I'm done with it, the only other conversion should take place when I shove that color data to another external device with an external digitizer. That's right: 2 steps. Colors that originate from a non-external source (i.e. I just type in color constants or something) only need to be converted for display – no cooking is necessary while I'm manipulating it. But that's not how it actually works…. CoreImage has its own wild ideas about when conversion needs to take place, as does QuickTime. And then there's still that final display/output step, where yet another conversion seems to take place. Just keep your hands off my colors, except for input, and output.

QTKit is a joke of a path to QuickTime in 64-bit applications, as noted in a previous post [9]. I can feel the venom of that post still in my veins, so I'll leave it at that. So infuriating.

Then there is CoreImage's inability to force a CIImageAccumulator to flatten itself. Instead, it *must* store the whole filter chain, and all associated images needed to render. Never mind that sometimes there are memory limits, and storing thousands of 4MB images is a bad idea if you're in 32-bit land (and still sloppy if you're not, since you'll still end up swapping madly on a modest system). There are internal private, undocumented hacks to work around this, but they've been there since 2005! Why has this been an issue for 3 years?!

And finally, today NSOpenGLContext/CGLContextObj decided to turn against me: While I can allocate, manage, and provide them with where I want them to render off-screen, I cannot do the same for a depth buffer – It insists on managing that itself. Never mind the fact that I'm working with 768MB framebuffers, and similarly-sized depth buffers, and I need to allocate those at the beginning of the app so that the allocation will actually take place (otherwise, memory gets too fragmented due to other frameworks doing their thing, and then the allocation fails, and you can't do anything about it).

I guess I can allocate the whole context right at startup, but really, why is that necessary? And how do I change that on the fly, in the event that I actually made one too big, for safety? I can reuse the too-big color buffer, but not the depth buffer… Why can't I dynamically provide color, depth, stencil, accumulation, and auxiliary buffers on a whim? Since it's a software renderer at that point anyway, it's not like it's doing anyone any services by hiding this functionality.

So, that's Cocoa lately: Applications are 90% easy, 10% impossible.

Links:

[1] http://fdiv.net/2008/09/13/90-easy-10-impossible

[2] http://fdiv.net/user/24

[3] http://fdiv.net/category/antisocial-story

[4] http://fdiv.net/category/apple

[5] http://fdiv.net/category/software-development/cocoa

[6] http://fdiv.net/category/not-apple

[7] http://fdiv.net/category/social-story

[8] http://www.cocoabuilder.com/archive/message/cocoa/2003/10/26/75323

[9] http://fdiv.net/2008/08/26/qtkit-qcheatkit