John Carmack Archive - .plan (2005)

http://www.team5150.com/~andrew/carmack

March 18, 2007

Contents

Chapter 1

May

1.1 Cell phone adventures (May 27, 2005)

I'm not a cell phone guy. I resisted getting one at all for years, and even now I rarely carry it. To a first approximation, I don't really like talking to most people, so I don't go out of my way to enable people to call me. However, a little while ago I misplaced the old phone I usually take to Armadillo, and my wife picked up a more modern one for me. It had a nice color screen and a bunch of bad java game demos on it. The bad java games did it.

I am a big proponent of temporarily changing programming scope every once in a while to reset some assumptions and habits. After Quake 3, I spent some time writing driver code for the Utah-GLX project to give myself more empathy for the various hardware vendors and get back to some low-level register programming. This time, I decided I was going to work on a cell phone game.

I wrote a couple java programs several years ago, and I was left with a generally favorable impression of the language. I dug out my old ”java in a nutshell” and started browsing around on the web for information on programming for cell phones. After working my way through the alphabet soup of J2ME, CLDC, and MIDP, I've found that writing for the platform is pretty easy.

In fact, I think it would be an interesting environment for beginning programmers to learn on. I started programming on an Apple II a long time ago, when you could just do an ”hgr” and start drawing to the screen, which was rewarding. For years, I’ve had misgivings about people learning programming on Win32 (unix / X would be even worse), where it takes a lot of arcane crap just to get to the point of drawing something on the screen and responding to input. I assume most beginners wind up with a lot of block copied code that they don’t really understand.

All the documentation and tools needed are free off the web, and there is an inherent neatness to being able to put the program on your phone and walk away from the computer. I wound up using the latest release of NetBeans with the mobility module, which works pretty well. It certainly isn’t MSDev, but for a free IDE it seems very capable. On the downside, MIDP debugging sessions are very flaky, and there is something deeply wrong when text editing on a 3.6 ghz processor is anything but instantaneous.

I spent a while thinking about what would actually make a good game for the platform, which is a very different design space than PCs or consoles. The program and data sizes are tiny, under 200k for java jar files. A single texture is larger than that in our mainstream games. The data sizes to screen ratios are also far out of the range we are used to. A 128x128x16+ bit color screen can display some very nice graphics, but you could only store a half dozen uncompressed screens in your entire size budget. Contrast with PCs, which may be up to a few megabytes of display data, but the total game data may be five hundred times that.

You aren’t going to be able to make an immersive experience on a 2” screen, no matter what the graphics look like. Moody and atmospheric are pretty much out. Stylish and fun is about the best you can do.

The standard cell phone style discrete button direction pad with a center action button is a good interface for one handed navigation and selection, but it sucks for games, where you really want a game boy style rocking direction pad for one thumb, and a couple separate action buttons for the other thumb. These styles of input are in conflict with each other, so it may never get any better. The majority of traditional action games just don’t work well with cell phone style input.

Network packet latency is bad, and not expected to be improving in the foreseeable future, so multiplayer action games are pretty much out (but see below).

I have a small list of games that I think would work out well, but what I decided to work on is DoomRPG - sort of Bard's Tale meets Doom. Step based smooth sliding/turning tile movement and combat works out well for the phone input buttons, and exploring a 3D world through the cell phone window is pretty neat. We talked to Jamdat about the business side of things, and hired Fountainhead Entertainment to turn my proof-of-concept demo and game plans into a full-featured game.

So, for the past month or so I have been spending about a day a week on cell phone development. Somewhat to my surprise, there is very little internal conflict switching off from the high end work during the day with gigs of data and multi-hundred instruction fragment shaders down to texture mapping in java at night with one table lookup per pixel and 100k of graphics. It's all just programming and design work.

It turns out that I'm a lot less fond of Java for resource-constrained work. I remember all the little gripes I had with the Java language, like no unsigned bytes, and the consequences of strong typing, like no memset, and the inability to read resources into anything but a char array, but the frustrating issues are details down close to the hardware.

The biggest problem is that Java is really slow. On a pure cpu / memory / display / communications level, most modern cell phones should be considerably better gaming platforms than a Game Boy Advanced. With Java, on most phones you are left with about the CPU power of an original 4.77 mhz IBM PC, and lousy control over everything.

I spent a fair amount of time looking at java byte code disassembly while optimizing my little rendering engine. This is interesting fun like any other optimization problem, but it alternates with a bleak knowledge that even the most inspired java code is going to be a fraction the performance of pedestrian native C code.

Even compiled to completely native code, Java semantic requirements like range checking on every array access hobble it. One of the phones (Motorola i730) has an option that does some load time compiling to improve performance, which does help a lot, but you have no idea what it is doing, and innocuous code changes can cause the compilable heuristic to fail.

Write-once-run-anywhere. Ha. Hahahahaha. We are only testing on four platforms right now, and not a single pair has the exact same quirks. All the commercial games are tweaked and compiled individually for each (often 100+) platform. Portability is not a justification for the awful performance.

Security on a cell phone is justification for doing something, but an interpreter isn't a requirement - memory management units can do just as well. I suspect this did have something to do with Java's adoption early on. A simple embedded processor with no MMU could run arbitrary programs securely with java, which might make it the only practical option. However, once you start using blazingly fast processors to improve the awful performance, a MMU with a classic OS model looks a whole lot better.

Even saddled with very low computing performance, tighter implementation of the platform interface could help out a lot. I'm not seeing very conscientious work on the platforms so far. For instance, there is just no excuse for having 10+ millisecond granularity in timing. Given that the java paradigm is sort of thread-happy anyway, having a real scheduler that Does The Right Thing with priorities and hardware interfacing would be an obvious thing. Pressing a key should generate a hardware interrupt, which should immediately activate the key listening thread, which should be able to immediately kill an in-process rendering and restart another one if desired. The attitude seems to be 15 msec here, 20 there, stick it on a queue, finish up a timeslice, who cares, right?

I suspect I will enjoy working with BREW, the competing standard for cell phone games. It lets you use raw C/C++ code, or even, I suppose, assembly language, which completely changes the design options. Unfortunately, they only have a quarter the market share that the J2ME phones have. Also, the relatively open java platform development strategy is what got me into this in the first place - one night I just tried writing a program for my cell phone, which isn't possible for the more proprietary BREW platform.

I have a serious suggestion for the handset designers to go with my idle bitching. I have been told that fixing data packet latency is apparently not in the cards, and it isn't even expected to improve much with the change to 3G infrastructure. Packet data communication seems more modern, and has the luster of the web, but it is worth realizing that for network games and many other flashy Internet technologies like streaming audio and video, we use packets to rather inefficiently simulate a switched circuit.

Cell phones already have a very low latency digital data path - the circuit switched channel used for voice. Some phones have included cellular modems that use either the CSD standard (circuit switched data) at 9.8Kbits or 14.4Kbits or the HSCSD standard (high speed circuit switched data) at 38.4Kbits or 57.6Kbits. Even the 9.8Kbit speed would be great for networked games. A wide variety of two player peer-to-peer games and multiplayer packet server based games could be implemented over this with excellent performance. Gamers generally have poor memories of playing over even the highest speed analog modems, but most of the problems are due to having far too many buffers and abstractions between the data producers/consumers and the actual wire interface. If you wrote eight bytes to the device and it went in the next damned frame (instead of the OS buffer, which feeds into a serial FIFO, which goes into another serial FIFO, which goes into a data compressor, which goes into an error corrector, and probably a few other things before getting into a wire frame), life would be quite good. If you had a real time scheduler, a single frame buffer would be sufficient, but since that isn't likely to happen, having an OS buffer with accurate queries of the FIFO positions is probably best. The worst gaming experiences with modems weren't due to bandwidth or latency, but to buffer pileup.