A Rapidsoft Systems' White Paper

© June, 2009

Mobile Development: Porting Mobile Applications on Brew, J2ME, iPhone, Blackberry and Windows Mobile Devices

Summary: One of the main problems of mobile software development is to deal with multiple platforms. On one hand, we have open standard like J2ME, on the other proprietary controlled standard like BREW, iPhone and Windows Mobile. Both approaches have their own merits since both aim to provide an environment where it is easy to write application once and then run on many handsets using that particular platform. Unfortunately, incompatibility between two environment means that a mobile application producer must re-write an application separately for each platform increasing development cost.

How can we then write software that minimizes porting cost for multi-platform development? At Rapidsoft Systems, by doing multiple projects on various platform we have developed this expertise over the years. This is learning by fire and experimentation since there is no guide exist that tells you how to go about such a task. But in the end, it is all about saving development cost to our customers.

Introduction

If you are like one of many mobile application producers, you face a major dilemma as to which platforms to support. The decision is not easy and it may have to be driven by business and commercial factors. But those are commercial decisions, and have nothing to do with technology difficulties in porting applications across mobile platforms.

In this paper, we talk about some of the reasons why people will like to port applications from PC to mobile platform, and from one mobile platform to another. We also provide some ground rules that can help developers write software that is easy to port across platforms and hence will cut down the development cost.

As we mentioned earlier, BREW and J2ME are very different programming environment. The same is true for iPhone and Windows Mobile. They all provide vary different programming functionality. For example, J2ME has many programming APIs which have no direct or corresponding functionality in BREW or on iPhone. For example, there is an API called Sound, Player and Volume Control in J2ME; whereas there are no libraries available for the same in C/C++. So what can a developer do in such situations.

How can we then write software that minimizes porting cost for multi-platform development? At Rapidsoft Systems, by doing multiple projects on various platform we have developed this expertise over the years. This is learning by fire and experimentation since there is no guide exist that tells you how to go about such a task.

Why Port Software

After spending many hours researching the subject, I thought I'd put down some thoughts about porting games and writing portable software in general. This is not an exhaustive tutorial on the subject but a rather presentation of key elements for writing portable software in general.

Why port software? Especially in the game industry, is there much to gain by catering to minorities? MacOS is sitting on 5 percent of the world's desktops, and while that's a significant number of users, the other 95 percent is almost all Win32 boxes, which means you can get most of the users from writing for one target, and think nothing of the Linux and BeOS users of the world. So what's the point?

The above logic applies to mobile environment even more. If top 50 handsets dominate the mobile market, is there any sense to support low end handsets that do not make top 50 list since the cost to return ratio is not quite as good? Carrier compulsions or the desire to be seen serving a wider market may motivate you to support more and more devices.

Here are some of the oft sited reasons:

- **Choice.** One person may like Linux and Linux command lines, others like MacOS X's Aqua, others like WinXP. Making a portable game lets your users work and play in the environment they like best. The consumer wins.
- **Device Popularity.** If you offer a game only on J2ME devices, but not on iPhone or Blackberry devices or vice versa, you are making certain assumptions and limiting consumer choices. Also, consumer may think your company as an elitist or lacking in support making them use another competitive product.
- **Code quality:** porting to a new platform exposes bugs you never knew existed in the first place. It forces you to remove assumptions from the program. It can make it easier to maintain. It can make it easier to license your code to others, or contract out work on that code. Having a game that runs cleanly across several platforms can actually lower the cost of maintenance if you do it right. It also helps maintain the programmers' sanity when problems like buffer overruns become obvious through different memory managers.
- **Consoles:** You want a Playstation 2 port of a game? Your first stop should be Linux. The migration path is easier than going to the console directly, it's cheaper to develop for, and yes, Linux and the PSX2 use the same compiler.

- **Financial reasons:** If a program is done right from the start, the cost to move it to new platforms is small, but the potential for extra sales increases. Mac and Linux users *do* tell their friends about software they like, and they tend to be very vocal in their praise of the companies that are willing to support their platforms. Games for Linux tend to have an infinite shelf life, whereas their Win32 counterparts tend to land in the bargain bin in a month or so.
- **Warm fuzzies:** I like supporting the underdog operating systems. It makes me feel good. Pyrogon's stated intention is to make quality games that don't cost millions of dollars to make. Having players enjoy their games is priority one, so it makes sense to get them into the hands of people that generally appreciate the games more than the twitchy overclockers that are giving Windows gaming a bad name.

We are speaking of games, but these statements are generally true for any kind of software.

So now that we know *why*, let's explore *how*.

To start, we have to express the Tao of porting: *no code is portable until it gets ported*. Sure, we all write wonderful code, and choirs of angels sing while we type, but there will always be unexpected problems that won't be seen until the source is pushed through a strange compiler on a strange operating system running on a strange processor. The trick is to minimize the amount of non-portability right from the start. This takes diligence and a little bit of know-how on the part of the coder. Knowing what you're doing can literally reduce the porting time by weeks. This knowledge is best gathered through experience, but these guidelines can be a push down the road of that first experience. If any of this seems like common sense, then it just means you've been down that road before.

Here is a summary of these simple coding rules formulated by a wise man.

Rule #1: Think before you code.

Sounds simple, doesn't it? Unfortunately, even in the video game industry (ESPECIALLY in the video game industry!), it seems that many developers jump right in and start coding. This is wrong, wrong, wrong. It doesn't take long for a project to become an unwieldy, hardcoded mass of spaghetti, which leads to the usual set of problems; however, if even maintaining (or, heaven forbid, *enhancing*) the codebase is difficult, it will be twice as hard to make it portable. What you need is a blueprint. Sketch it out, write it down, babble incessantly about your plan to everyone around. Have an attitude that lets people tell you honestly if a plan is stupid, and prepare to revise details or whole subsystems. Better to do this now than find yourself reworking the program during crunch time.

Rule #2: Make abstractions.

If you're writing a Windows game, sooner or later, you are going to have to call a Windows-specific function. Things that can be done portably (like using stdio instead of the win32 API) should be done, but other things, like blitting to the screen, are system-specific. What you should do is take a few minutes and wrap things like DirectDraw in a simple class, and expose their functionality in general ways. Do NOT expose DirectDraw data types, to prevent the urge to bypass the abstraction. If something can't be done through the abstraction layer, then the abstraction layer should expand. Candy Cruncher

does this very thing for audio, video, the "registry", input, etc. For example, you've got a "DisplayDevice2D" class. This class is subclassed into a DirectDraw version, a GDI version, an SDL version (for Unix), and a Carbon version (for MacOS). There are some immediate benefits here, even in single-platform development. Note that two of those subclasses are for Windows; this gives Pyrogon the ability to choose the best balance of performance and stability at runtime for any given run of the game. Theoretically, they could add an OpenGL-based subclass of DisplayDevice2D that renders the sprites as textured quads to increase the framerate more, at their users' discretion. This would be added to the framework, without changing the actual game's code, and would benefit the next game they do, too. Good abstraction makes good design, and the benefits are quick ports to other architectures, more flexibility for the power users, and better support for various hardware on the primary platform. It's a huge win.

Rule #3: Be data driven.

Spend as little time in your program as possible. Branch gaming logic out into scripting languages as quickly as possible. This isn't an argument to write your whole game in Perl (unless you want to, I guess); instead, get the stuff that *must* run fast in C/C++ code (which is almost always the blitters in a 2D game, reference Rule #2) and get the game logic itself out to something you don't need to compile every time you tweak it. I say this not just because it's a good idea, but porting a script interpreter is frequently easier than looking for subtle problems in game logic in C. But what scripting language is best? It depends on what you need. If you need something basic, roll it yourself, but it's better to embed an existing scripting language; there are many that are portable, debugged, and supported to choose from. Perl, Python, and Scheme are just some options. The Pyrogon framework has Lua, which seems to be popular for scripting game logic.

Rule #4: Be sensitive to byte ordering and packing.

If I ever see another game that sends "sizeof (myStructure)" bytes over a network connection, I'm going to scream. I should scream right now, because I will no doubt see this again. Candy Cruncher is not a networked game, but it does run on both Intel (Windows and Linux) and PowerPC (MacOS X) systems, which means that it has to be careful about reading from and writing to files. Between processor types, operating systems, and even compilers, sizes of data types change. I'm talking about something more subtle than the classic C problem of an-int-is-not-always-the-same-size-everywhere, although that's important, too. Structures get packed differently (and not every compiler can understand #pragma pack), data has to be aligned differently, and data gets stored backwards on different processors. If you have to read or write structures to disk, a network socket, or anywhere that a different system may see it, you should send it, one scalar at a time, in an agreed upon format (bigendian or littleendian), and rebuild the structure on the other side of the connection. Do not send floating point numbers ever, if you can help it, since different CPUs have different precisions, and you can only correct for this so far (I can think of at least four ports I've worked on that got bitten by the floating point thing. Be wary.) If you do not do this now, it is nearly impossible to fix it later in a program of any size.

Rule #5: Write what you have to, steal the rest.

I've just told you to write a scripting language and be very careful about how data gets manipulated.

Right now you're probably wondering how any of this is supposed to make your job easier. Hey, I said this takes diligence! However, the secret is really in the open source community. Why should you write image decoders, and audio format decoders, and scripting languages, when they are freely available for the taking? Candy Cruncher takes advantage of several cross-platform libraries: Lua, zlib, and Ogg Vorbis, to name a few. The Linux and Mac Classic ports use SDL, SDL_ttf, and SDL_mixer, not to mention Loki Setup for the installer. This is literally years of development time that can just be dropped into place, and more importantly, all of these libraries are cross-platform to start with, so you don't have to wonder how you'll get that .OGG file to play on BeOS; it just will.

Rule #6: Don't use assembly language.

Just don't. If you must, you better write a C version and optimize based from that, so that there is a *working* fallback. But don't write assembly in the first place. 99.5% of the time you think you need it, you don't. Just say no; Candy Cruncher did.

Rule #7: Listen to your beta testers.

Sooner or later, your port will be ready for external testing (you *are* going to do a beta test, right?) and you will be unleashing your baby into an unfamiliar world. If this isn't your primary development platform, chances are it isn't a platform you know all the ins and outs of. Even Linux users will find that different distributions do things very differently, and every Linux user has her own routines and traditions. Listen to what they ask you for, and give them what you can. One of the beta testers for Candy Cruncher noted that the game's response was a bit jerky on his box, and wondered if we could make it use the X11 cursor instead of drawing a sprite. We added that. Another wanted to have his Unix login name be the default when entering his high score. It was a good idea that never crossed my mind: added. People with keyboard layouts I've never heard of showed up: fixed. People with exotic display targets poked their heads up: tweaked. Odd sound problems on certain distros: debugged. These requests are to be expected, and are relatively trivial to implement, but they lead to happy customers and, again, a more flexible codebase. You do *not* want thousands of demo downloads from users that would have bought the game if only the mouse was a little more responsive. You could *not* have predicted it until someone came along and ran their X-server at an odd color depth. Anticipate possible differences in platforms, but be ready for anything.

Rule #8: Embrace Murphy.

Not every idea is a good one. If something isn't working, chuck it. If a subsystem isn't portable, make it so. If you are modular, and abstract, this makes the code easier to drop into a future product. Like I said, nothing is portable until it's ported, and all the planning in the world doesn't beat Murphy's Law. In such cases, don't be afraid to throw something out and replace it with something that works better. Struggling only makes it worse.

Conclusions

Mobile application distribution has become a lucrative industry with a manifold increase in mobile subscriptions and the number of avid mobile gamers around the world. However, success depends on reducing the cost of development and porting and timely execution of the development and application porting.

Use of external porting can cut down the cost and time to market applications. While it is possible to create an internal porting capabilities for most companies that option will be quite expensive since it requires significant man power efforts that are cyclical in nature. This means using a trusted partner like Rapidsoft Systems to develop and port your applications can be the best solution for you.

For more information and specific questions, please contact us at:

Rapidsoft Systems, Inc,
Mailing Address: 7 Diamond Court, Princeton Junction,
New Jersey 08550, USA

(Princeton) New Jersey, (San Jose) California, Delhi/ Gurgaon (India), Mumbai (India), Chennai (India)

Web: www.rapidsoftsystems.com

Phones: **1-609-439 4775 / 1-609-439-9060 (US East Coast, NJ Office)**
1-408-829-6284/ 1-408-890-2509 (US West Coast, San Jose Office)
Fax: 1-831-855-9743

Email: info@rapidsoftsystems.com