Six Tips for Developing Easily Ported Games

By John Till

For a game developer, one of the greatest feelings in the world comes from watching someone happily playing your game. You can see how much this player is enjoying it, so why isn't everyone else playing it? The simple answer is that not everyone can, because not everyone has the same hardware or runs the same operating system. Millions of game consoles are sold each year [1]. New PCs, laptops, netbooks, and smart phones are released with amazing frequency [2]. Tablets are quickly establishing their place in the market. Standing between you and a world full of potential players are hardware and software variations of epic proportions. How can your awesome game possibly reach the masses? These six tips should help:

Tip 1: Be Flexible with Your Presentation Layer

When I first started developing Pocket PC* games, I had no intention of porting them to other platforms. That was a bit short-sighted on my part, because I was writing and compiling them in Microsoft Visual C++* on Microsoft Windows* for testing purposes anyway. The code had to be re-compiled with Microsoft eMbedded Visual Tools* (available at no cost) before the games would even run on the actual devices. Unfortunately, I had also made the mistake of locking myself into a portrait-oriented, Quarter VGA (QVGA: 240 × 320) resolution, as that's what all of the devices were using at the time. Oops.

The moral of this story is that building your game for a fixed resolution and orientation is generally not a good idea. There are just too many devices with varying displays. Your presentation layer needs to be flexible and separate from the data layer. Before the rise of smart phones and tablets, nearly every display was landscape oriented. Because handheld devices can be turned on their sides, they too can become landscape devices. This makes porting a lot easier.

There are plenty of strategies for handling different resolutions. In the 2D game world, if you design your game for a fixed resolution (like 800 × 600 pixels), the user can typically run it full screen, in which case the monitor handles any scaling. The game could also be run in a window, but if you want the window to be resizable or you want to run it on a device that doesn't allow 800 × 600, then who's responsible for scaling? One popular technique used in OpenGL* is to draw your graphics on a single texture of your desired resolution, and then show that one texture scaled to the full size of the screen (or window). Even though it's only a 2D game, you can take advantage of a 3D engine to ensure that the entire image will be there regardless of device resolution.

You need to take some other factors into consideration, as well. What if your fonts become illegible as a result of scaling, or your objects don't look right at a different resolution? In such cases, where you need "pixel precision," you may have to alter what you show on the screen. For example, assume your 800 × 600 game (A in Figure 1) is running on a typical netbook with a 1024 × 600

resolution. If you have a game board sitting on top of a background, you could stretch just the background to the full resolution (B in Figure 1), leaving the game board unchanged. You could also use an alternate background designed for widescreen displays (C in Figure 1), or you could have two 112 × 600 images that could be placed on either side (D in Figure 1).

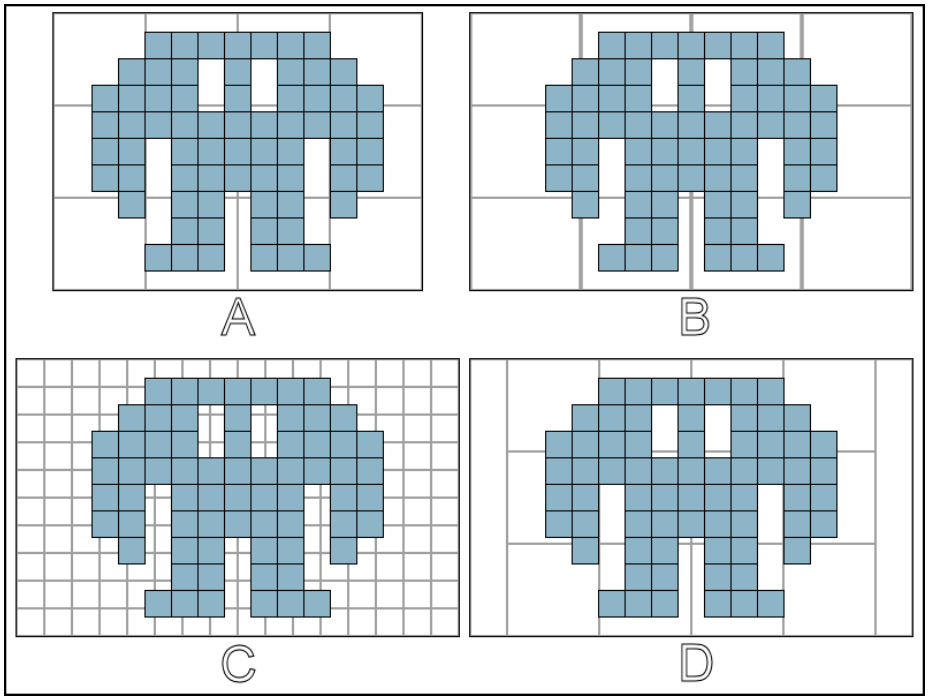

Figure 1. Altering the dimensions can make your game look better.

Regardless of whether you alter the display to compensate for a less-than-ideal resolution or orientation, be sure to use variable positioning for your visual elements. Doing so not only provides the greatest level of flexibility, it allows you to implement cool features like moving the screen elements on the fly in response to feedback from a device's accelerometer or smoothly transitioning between landscape and portrait orientations. Some devices may look much better with a particular layout, so being able to easily reposition your elements is incredibly useful.

Tip 2: Use Middleware That's Already Multiplatform

Unless you're writing everything for your game from scratch, it's rather likely that you're going to integrate someone else's technology or middleware solution. This could be anything from an entire game engine [3] to path-finding logic to script interpreters to multimedia libraries for video and audio. There are a lot of options out there [4]: Let's take a look at a few.

Computer Graphics

In the realm of computer graphics, two 3D technologies tower over the rest: Microsoft Direct3D* [5] and OpenGL* [6]. Both offer great visuals and are supported by graphics card manufacturers, but only one of them is an open standard. This doesn't make a huge difference if you're only developing for Windows, but that's a lot of code to rewrite if you decide that you want to port your game to a platform that doesn't support Microsoft DirectX*.

Audio Libraries

Audio libraries are in the same boat as their visual counterparts. Each platform typically has its own proprietary application programming interface (API) that you can take advantage of, but then you're stuck having to rewrite your function calls for the next platform. Thankfully, several multiplatform alternatives are available that will likely suit your needs, such as OpenAL* [7] and FMOD* [8].

Does this mean you should completely avoid proprietary libraries? No. Some of those libraries may provide additional quality, performance, features, and power or cost savings that you just won't get from a multiplatform library. You'll have to weigh the advantages against the disadvantages. In the end, your game should offer the richest possible experience for the user's platform.

Also keep in mind that not all platforms are equal, and in some cases you can save development time or reduce licensing costs by taking advantage of a less feature-rich solution. For example, you don't need to implement 7.1 surround sound on a platform that only offers stereo output.

Tip 3: Consider Mobile Users

Every day, thousands of new smart phones, laptops, and tablet computers are turned on for the first time. Unfortunately, they can only function as long as their batteries allow them to. Gamers in today's society aren't tied to power outlets. If you keep that in mind, the mobile ports of your game can benefit greatly. Of course, this means compiling your game as a native application, which is less portable than an interpreted program (see Figure 2).

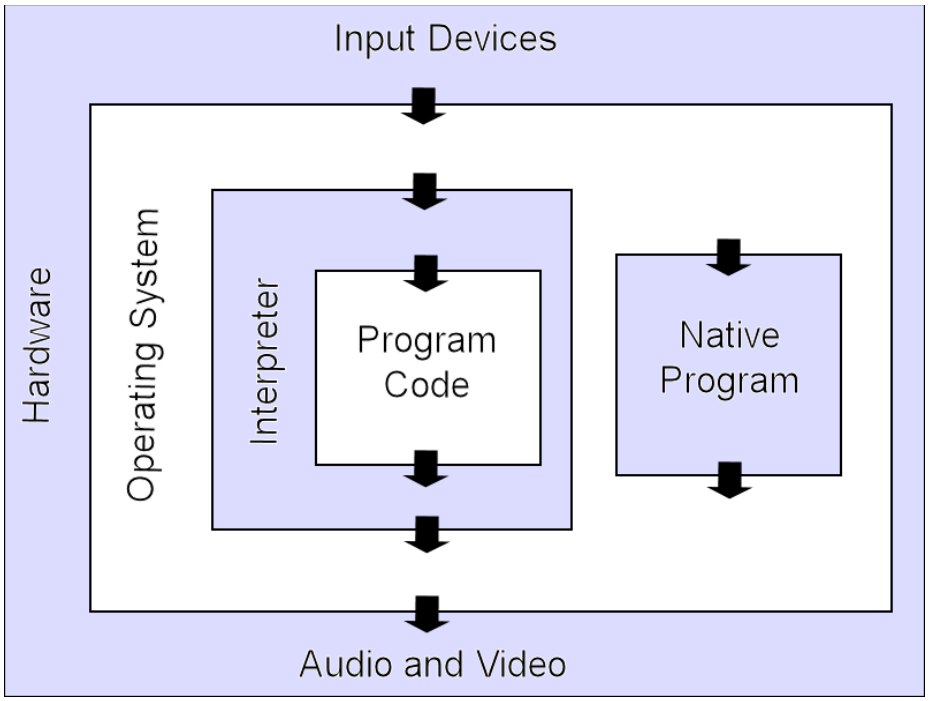

Figure 2. Native applications require less processing than interpreted programs.

Interpreted programs have to be converted into machine language commands by another program, called an *interpreter.* The interpreter also acts as a middleman between your game and the host operating system for handling the user's input and the game's multimedia output. This results in additional CPU cycles, which can slow down performance and deplete the battery faster than an equivalent compiled program would.

Because a native application is not dependent on an interpreter, it can offer the player higher frame rates, increased responsiveness, and less battery drain. Also, because it eliminates a layer of complexity, native applications can even result in fewer technical support incidents.

However, an interpreted program has a couple of serious advantages: quick development time and the potential for the program to work on any platform for which a compatible interpreter has been written. If the interpreter has already been ported, you may not have to do anything to your game to get it to run on that platform. That's incredibly convenient for you, but your mobile users will have to pay for that convenience.

When you play an Adobe Flash*-based game [9], you'll notice that it runs in a plug-in in a browser window (see Figure 3). Every key press or mouse action has to be translated multiple times before it reaches the game logic. On a desktop machine, it's a small price to pay. Flash is compatible with a lot of platforms, and Flash programs typically only need to be written once. However, Flash is still not supported on all platforms and may require the user to install the Flash plug-in first, and the content itself needs to be sent from a server, which could take some time to download, depending

on the user's Internet connection. For mobile devices, wireless signals can quickly deplete batteries.

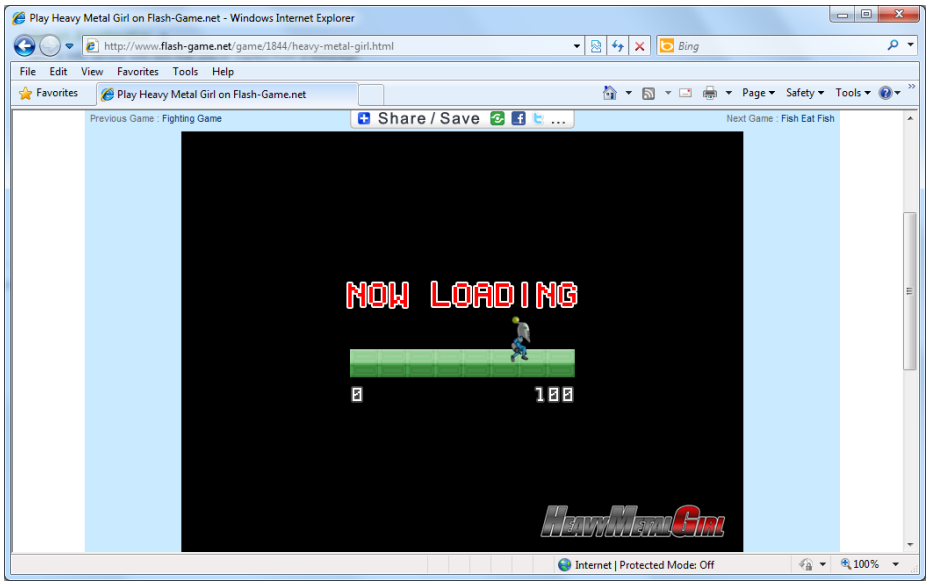

Figure 3. Adobe Flash-based games can be played in most Web browsers.

Whatever you decide to use, keep in mind that mobile users have a limited playtime before they need to recharge their devices. Any strides you take to conserve power while maintaining a smooth and exciting game experience will earn you their appreciation. Compiling your game so that it runs natively is definitely a step in that direction, but you should expect it to take longer to achieve your porting goals.

Tip 4: Allow for Multiple Input Methods

Keyboards, mouse devices, touch pads, touch screens, accelerometers, joysticks . . . These are the typical input methods you'll be faced with when porting your game. It may look like an intimidating list at first, but it can be simplified into a few categories. For example, if you're using a touch pad on a laptop or netbook or a touch screen on a tablet or smart phone, your actions are generally converted into mouse inputs: X + Y + Event. Gestures are a relatively new exception that not all devices support. Keyboards, accelerometers, and joysticks may or may not be present depending on the device, whether it's docked or undocked, and so on.

Simply put, there are so many hardware variations that you typically can't get by with programming for just one type of input unless you're porting to a very particular set of devices. But how do you decide which inputs your game should handle? Table 1 provides a decent guideline.

Table 1. Common device inputs based on hardware category

	Keyboard/hardware buttons	Mouse/touch pad/touch screen	Accelerometer	Joystick
Desktop	X	X		
Laptop/netbook	X	X		
Tablet		X	X	
Smart phone		X	X	
Console	X			X

Granted, there are plenty of desktops with joysticks attached, smart phones with built-in keyboards, and so on. The requirements of the game should ultimately determine which inputs you decide to implement.

Tip 5: Plan for Localization

If you want your game to go global, then you need to face the reality that not everyone speaks English. Even if you don't speak any foreign languages yourself, you can still plan for it rather easily. Localization can usually be handled by focusing on three specific locations: text strings, art assets, and vocals.

For most strings that show up as labels, mouse-over text, long descriptions, etc., you can use a simple text file that contains an index and a translation. When your game loads, it can read that file into an array, and then you can recall the proper translation when you need to display it.

Storing text in the art assets doesn't require a lot of work on the coding side, where a programmer can simply load a different image, but having the artists recreate the same assets multiple times (once for each locale) is something you want to avoid, if possible. If the artists are already maintaining multiple variations of the assets—for different devices, quality levels, etc.—the amount of work they need to do can grow prohibitively large.

Vocals can really help immerse your players in your game, and the easy way around having to record them in multiple languages is to do what movies usually do: Show subtitles. These are good to have anyway, because they give players a chance to see exactly what the characters are saying, even if they have their sound turned off or need to pull their attention away from the game for a moment.

Text usually doesn't take up a lot of space, but image and sound files do. Because most users will only want to play the game in their native tongue anyway, they likely won't want all of the localized art assets and vocals needlessly taking up extra space. That space can be especially

precious on a mobile device or solid-state disk (SSD). If you can't avoid having the translated assets, you should at least be able to package the localized versions separately. This will save space, decrease the download time for your game, and make for a better user experience. It will also allow your game to hit the market sooner, because you won't need to wait for all of the translations to be completed.

Tip 6: Choose a Development Environment That Makes Porting Easy

Most integrated development environments (IDEs) originally weren't created with the intention of being used for multiplatform development. Today, however, there are quite a few tools that will compile your game for desktop, mobile, and even console platforms with minimal or no modifications to the source code. Two popular offerings are Microsoft XNA Game Studio* [10] and the Torque* [11] development tools.

*XNA Game Studio**

Taking advantage of Microsoft Visual Studio*, C#, and the Microsoft .NET* Framework, XNA Game Studio may seem limited by its proprietary nature at first glance, but when you consider that the games you can make with it are capable of running on the Windows operating system (for desktops, laptops/netbooks, and tablets), Microsoft Xbox 360*, Microsoft Zune* media players, and the upcoming Windows* Phone 7 series of devices with almost zero changes to the source code, that's a huge market of players just waiting for your awesome game to come their way. Figure 4 shows the XNA Game Studio interface.

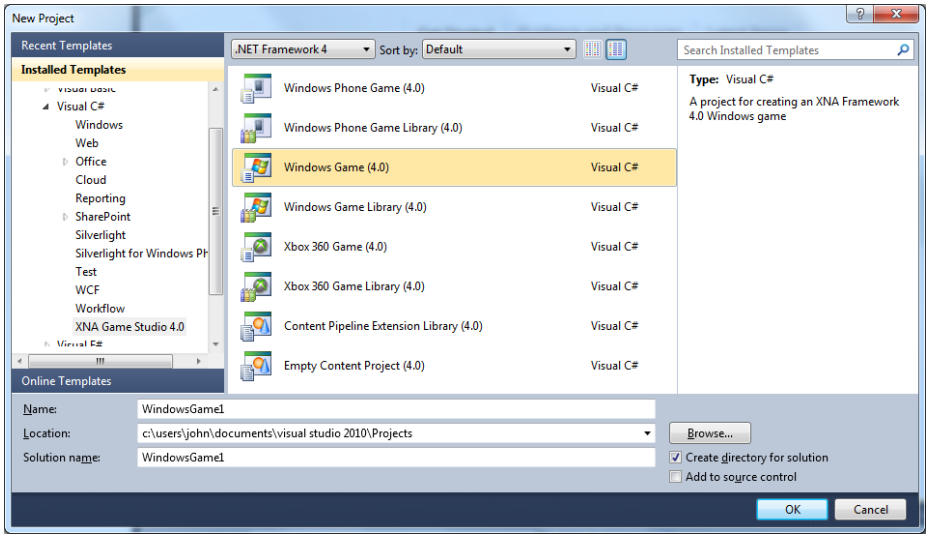

Figure 4. Create. Compile. Run. Microsoft XNA Game Studio templates make it easy.

How much does it cost? This is one of XNA Game Studio's greatest strengths: It's free to download and develop your games. You can also release your Windows games without sending a dime to

Microsoft. Publishing for Xbox 360, Zune, or Windows Phone 7, however, requires that you register as a developer on the appropriate distribution portal (which is about US$100 per year).

How easy is it? An experienced developer will pick it up fairly quickly. New developers will want to familiarize themselves with C# programming first, but there are plenty of online tutorials and books written for it. Overall, the platform has matured very nicely since its release in early 2006.

Torque (2D, 3D, X, etc.)*

Originally built as a 3D engine for the game *Tribes 2,* the Torque development tools have evolved significantly over the past nine years. They're currently split into a variety of separately licensed products, each focusing on a specific market or platform. The code, however, is largely portable across the Torque IDEs. For example, a game written in Torque 2D for Windows can be compiled with iTorque 2D*, which targets an entirely different set of devices. Torque's greatest strengths are found in the number of platforms it supports, its large community, and its developer-friendly interface (see Figure 5).

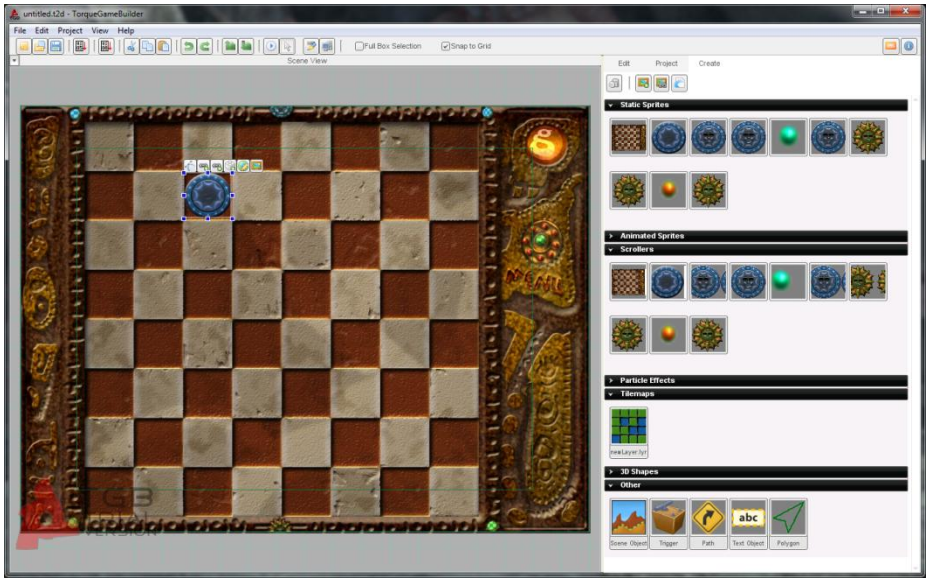

Figure 5. Torque provides templates and drag-and-drop development.

How much does it cost? You'll need to visit the Torque Web site for current pricing, but you should expect to pay at least US$100. You can also download a free 30-day trial version of Torque 2D*.

Although XNA Game Studio and Torque are only a small sample of the multiplatform development tools available, they should give you a good idea of what to look for in a multiplatform IDE: easy application development, support for plenty of popular platforms, and minimal or no re-coding necessary.

Summary

Not everyone owns the same hardware. Not all devices run the same operating system. That doesn't mean that your game can't be played by everyone. With careful planning and the right tools, your game can be enjoyed by millions of players all over the world.

References

[1] Video game charts: http://www.vgchartz.com

[2] Tech news sites: Engadget (http://www.engadget.com) and Gizmodo (http://www.gizmodo.com)

[3] Survey by Mark DeLaura at Gamasutra: http://www.gamasutra.com/blogs/MarkDeLoura/20090302/581/The_Engine_Survey_General_results.php

[4] Middleware solutions: http://www.gamemiddleware.org

[5] DirectX* Developer Center: msdn.microsoft.com/en-us/directx/default.aspx

[6] OpenGL*: http://www.opengl.org

[7] OpenAL*: http://connect.creativelabs.com/openal/default.aspx

[8] FMOD*: http://www.fmod.org

[9] Flash* Games: http://www.flash-game.net

[10] XNA Game Studio* Creators Club Online http://creators.xna.com

[11] Torque* Development Tools: http://www.torquepowered.com

About the Author

John Till is the independent developer responsible for all of the games at Pocket Adventures.com. His games have been acclaimed in Pocket PC Magazine*'s Annual Software Awards and have ranked in the top ten lists at major distribution portals like Handango and PocketGear. John holds a Bachelor's degree in Computer Science from Purdue University and has more than 12 years of experience working in IT.*