Writing Portable Code

Didier Malenfant
Naughty Dog, Inc.
dids@naughtydog.com
http://www.naughtydog.com

Introduction

The concept of portability is very often overlooked in current projects. At best code is 'ported' at the end of the development cycle from one platform to another with great pains. Features or overall quality usually suffers. Does that kind of porting nightmare sound familiar? Even if two versions are being developed concurrently, a lot of the time one version is privileged and ends up progressing faster than the other.

Writing Portable Code is a totally separate process from porting code, the first making the second a lot easier. Although porting will be discussed briefly in this paper, it will mainly be to explain the consequences of choices made while writing code in the first place. Porting badly written code is hard and unfortunately there aren’t a lot of things one can do to make this process easier. Actually there is, make code portable in the first place.

So how hard is it to write portable code? It’s actually a lot easier than people think and the best thing is you can benefit from this even if your code never ends up getting ported to anything else. The practices it promotes go hand in hand with writing better code altogether. It won’t solve world hunger but it might make you a better programmer.

One word of caution: Making your code portable might not make your game portable. We don’t see a lot of fighting games on PC or a lot of real-time strategy games on consoles and there is a reason for that. Game design and concepts might not lend themselves very well to your porting target. While this might not be a huge issue to your decision-makers, it’s something to keep in mind. This is why I will limit myself to a discussion about porting game engines rather than porting games. Keep the difference in mind.

Why should you care?

Is it really that big a deal? It could be. Like so many other things it depends on your situation and your environment. A lot of the time it does require some extra work and it might be a hard sell on your producer or teammates if they don’t see the need for it. Time is money in game development like in anything else and it will always be a lot faster and easier to write code tied up to one platform. Where the win might be is when the time comes to make the code available on another
platform and you are under pressure to deliver, fast.

I personally developed the taste because of my fascination with weird alternative platforms. I might come across as a freak but when we used SGI Indys to develop an N64 game I was working on, it was a total blast to port my personal routines to it over a single weekend and have all my applications and libraries working under IRIX. I never used this for anything afterwards but this is an example of what drives my interest in the matter.

Personal hobbies apart, think about your ability to port part or your entire game engine to one of those next-generation platforms we all keep hearing about. The faster you can get the current version of your code up to speed, the more time you’ll have to work on enhancing it to make the most of that particular platform rather than just delivering a carbon-copy product. Hardware manufacturers are always out there looking for people able to show off their new platform or feature and that OEM deal, or that first generation console title, might make all your efforts worthwhile.

Another side of portability is that software gets used in unexpected ways, in unexpected places. Development cycles are now under so much pressure that different teams in the same company might end up reusing code from one project to another, sometimes from one platform to another. The more general your code is, the better. As we will see, writing modular code is a good way to write portable code. This has the interesting side effect that high-level code end up being easier to share and more useable in different situations. Environments also change, compilers evolve, APIs and libraries change and you will be better prepared if your code is written with portability in mind.

Last but not least a portable program is a better program, and it will make you a better programmer. This might sound like a bold statement, but the issues that are involved in writing portable code are very similar to the ones involved in writing solid code. It makes you make design and code architecture choices that will all make your code also better designed and better constructed. As a bonus, code that gets reused on multiple platforms and projects is likely to be better tested.

There is no such thing as a fully portable program and I won’t be able to present you with the magic potion that will make you port anything to anything. There will still be some extra work involved and even if just about anything might be achievable, it might not be worth the effort in your case. Writing portable code is a concept that should drive your choices and the way you approach the problems you’re faced with. There is no perfect solution but we can aim for it and still get the benefits it brings us.

**Work environment and tools choices**

Too many people get locked into one platform. A lot of the time this happens
without them noticing it. The complaints we hear about one platform being inadequate or underpowered can very often be traced to a difference in the tools or the architecture. The main problem with getting used to one platform is that it can greatly affect your productivity when the time comes to port your code to another target. The more you know about the differences that exist, the better you will be equipped to make choices about which tool or feature to use in your projects.

The IDE you use might not be available everywhere, there are currently very few of them that are multi-platform. If that IDE handles project management you could be in trouble when another platform requires you to use make files. This can be a particular annoying problem if the IDE you use tries its best to prevent you from generating or using make files. Also in this case, who is going to generate the dependencies for all your source files? Hard coded dependencies inside your make files really isn’t something you want to get into with current large size projects.

The solution is to use widely available tools like GNU make and the make depend command which will retain their functionality when ported to another platform. The source code for these is also widely available which makes them a breeze to customize. Your favorite text editor could also become your worse nightmare if you cannot use it in you new environment. Luckily many good text editors are available on multiple platforms like Visual Slick Edit or Crisp. They also often integrate building process inside the editor just like any other IDE out there would. If you are hard bent into finding the same editor on a lot of platforms you might want to give Emacs a try. It’s free and it’s available on virtually everything.

Remember that not every operating system out there handles their file system in the same way. Think about case-sensitivity for a start. A lot of development is being done on the Windows platform, which is completely case-insensitive when it comes to its filenames. When porting to another operating system like UNIX this can become a major issue if your include statements don't use the correct case. Also drive letters do not exists outside of the Windows world and therefore are best left out of include statements altogether. One of the best solutions to this particular problem is to add a platform specific include path to the compilation environment, leaving source files free to use relative paths in their includes. As we will see later on this is a better solution than using pre-processor conditional statements selecting between different include statements.

Version control can also become a problem. If the program you use is based on a file database system then most often than not this can be hosted on any operating system that supports the SMB file sharing protocol. The server or clients programs can be a bigger problem, as most of them are platform specific. Again alternatives exist like the RCS/CVS system, which not only exists on multiple platforms but also handle multiple file checkouts and FTP-like access to a remote database. A lot of open source projects are developed using CVS, which makes this software very stable and error-free.
One of the biggest tools in a programmer’s toolbox has to be the compiler and this can also be the source of major headaches. Compilers have different sensibilities to some warnings, and we all have encountered a program that compiles with no errors on one platform, only to be flooded by warnings on another. The solution to this is to enable full warnings on every platform you use. Some compilers offer a pedantic option that will make sure your program abides by strict ANSI rules. This might sound like a really annoying solution, and I have encountered many programmers who have the habit of turning off annoying warnings rather than fixing them, but this is the only way to stick to a standard you know is likely to be available everywhere. You may also want to build your code on a different compiler, such as GNU, just to sanity check every now and then that your code actually compiles without warnings or errors.

In general make sure you stay in the mainstream of the language you use and don’t use weird constructs or libraries that, although probably very clever and useful, could come back to haunt you when time comes to port the code to another platform. An example of this is the order of evaluation of some constructs in C that isn’t defined in any standard and is therefore NOT guaranteed. Compilers specific constructs like pragmas or variable parameters macros are also a bad idea unless you manage to hide them in a system specific file. Libraries can be a tough choice and you should make sure that this ‘Standard’ library you decided to use is indeed a standard. Sometime even this isn’t enough, as standards aren’t sometime implemented everywhere or correctly.

One of the issues that come up sometimes in console development is: Is it ok to use C++? The answer is: It’s getting there slowly. Some early compilers for consoles used to generate horrible code for C++. To make matters worse, the best code constructs like the ones using virtual tables were the one producing the worst code. The reason behind this was simple, Japanese programmers didn’t use C++ and therefore little effort was put into producing good results. Nowadays with next-generation platforms things are getting better, but the biggest problems remains: C++ hides a lot of its generated code from you, so keep an eye on the code generated for your programs.

When porting code to another platform, sometimes half of the battle is getting everything to compile in the first place. If you plan ahead and make this process faster you’ll have a main loop up and running faster which really makes the rest of the work go a lot smoother. Empty function calls can then be replaced, one by one, until the full product is up and running on the target machine.

**Portable data types**

At the lowest level of code portability lays the data types you commonly use throughout your program. Longs, shorts and ints are standard type for both C and C++ languages so they probably won’t cause you any problem when porting
them to another platform right? Wrong.

Ask people around you how long is a long type and listen to all the answers you will get. The ANSI standard only actually specifies that a long is ‘at least’ 32 bits and that the following relations are always true:

\[
\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \\
\text{sizeof(long)} \leq \text{sizeof(float)} \leq \text{sizeof(double)}
\]

At least is the magic keyword here and what it means is that you can’t rely on longs to be a certain size. You can only rely on them being able to hold ‘at least’ 32 bit values. This has implications on a lot of commonly written code, like initializing a long variable to 0xffffffff to set all its bits or set it to -1. If the long is 32-bits it works, if not you have a problem that can be sometime very hard to track down. Be aware of this as some of the future next-generation platforms might use exotic things like 64-bit wide longs.

Another case when type sizes can cause a problem is when using them for structure declaration. You cannot expect that a structure will have a certain size or, due to alignment issues on some platforms, that the size of the structure is the sum of its parts. One solution to this is to define your own types with names like int8, int16 and int32. These can be defined in a system depended file and you can even use some compiler constructs to generate an error during compilation if sizes are incorrect. A safe assumption is that types are aligned on their own size (32 bits are 4 bytes aligned and so on…), which enables you to build your structures with the correct number of padding bytes if necessary rather than leave the compiler to pad out structures behind the scenes. I usually make all my structures at least 32 bits aligned.

Float32 and Float64 might be a good idea too. In general using your own types, even if they default to the standard type, enable you in the future to easily replace them by more suitable ones on other platforms. An example of this is a fixed-point class replacing floats in a Playstation program, the Playstation having to emulate floating-point operations in software. One rule could be not to use char, short, long, float or double throughout your code, replacing them by your own size-explicit types.

An exception to this is the int type. Plain simple int can also vary in size on different platforms but this can actually be used to your advantage. Ints are usually the same size as a machine word that makes their use very efficient for counters and any local variable where, unlike in structures, size isn’t the primary concern. Using your own int16 type on a 32-bit machine will cost you wasted cycles in masking off the unused part of the 32-bit word or worse, multiple load for the same 32-bit word when the two 16-bit halves are accessed. For structures this might not be an issue, as low-level ones will allow you to write optimal assembly code to perform only one load and cache the unused half for later use.

Another solution can be to use size-explicit types for persistent data and
externally imposed formats like hardware registers, and use unsized ones for everything else. If you decide to go that route keep in mind the minimum sizes described by the ANSI standard and make sure that new platforms and compilers abide by it.

An interesting type to remember is size_t, which defines a type to hold memory sizes on a given platform. This will become more and more useful going forward into more and more 64-bit environments.

Bit fields are generally a bad idea, especially if used for persistent data or hardware registers. A lot of people still find them very useful but keep in mind that they tend to be extremely machine dependent.

The last problem type is char which can either default to signed or unsigned depending or the compiler you use. Consider the following piece of code:

```c
Char ch;
Ch = 0xFF;
If (ch == 0xFF)
{
    ...  
}
```

Type promotion rules will make this program fail on some platforms since both 0xFF and ch get promoted to int and the result therefore depends on character sign-ness. Again defining your own int8 and uint8 can help solving this problem if you have a way of checking at compile time that they are indeed correct.

If you want to go the final mile in type portability madness you should probably also stop using chars to represent strings. This is probably some of the nastiest type of changes you can have to do late in a project, when time comes to support Unicode characters. As we will discuss later, encapsulation can enable to hide strings into a totally abstract data type, which will enable you to change the representation without changing the code using this type.

Finally we wouldn’t be complete without mentioning the endian-ness issues between different platforms. Some platforms store their data in a big-endian order (most significant byte first), some in little endian order (least significant byte first). This is mostly an issue when exchanging data between two different platforms and can be resolved by writing platform specific accessing functions or use ASCII for your data files which doesn’t suffer from this problem. For best results, try to avoid using functions that convert fields while accessing them. Instead network or file data should be convertible to the native format upon loading/receiving the information.

**Data abstraction**
Data representations might be different from one platform to another. Also a certain representation might offer better performance on one platform while being totally inadequate for another. This forces us to hide the way some data is stored to make sure high-level code has no knowledge of the representation used and therefore stays fully portable.

The lowest level of this is the endian issue we mentioned earlier and the solution we found for this will also apply for different types of data. Access functions enable us to encapsulate different data while providing the same functions to read or modify it. Object-oriented languages call this data encapsulation and provide a lot of functionality to make this task easier. You can even declare some variables private making sure that no code but the member functions have access to it. C++ even enables you to re-define operators, which is called overloading, to make in some case your newly defined type act the same as the default ones. A good example of this was the fixed-point class used to replace floating-point numbers.

The fact that other languages, like C, don’t directly support object-oriented programming doesn’t mean it can’t be done. One of the most basic things that C++ hides from you is passing a pointer to the object a member function works on. You can do this yourself and write sets of functions that work on a specific structure type. If your high-level code never accesses the members of that structure directly you will be able to easily change the content and the inner-workings of those structures without changing the code using them. Using a get/set mechanism for each variable might seem very clumsy but it is the only choice when working with machine/platform specific data.

This also makes it easier to instantiate and work on multiple copies of the same object, which wouldn’t be possible with global variables.

A good example of this is a FILE data type. FILE is commonly defined as part of the POSIX standard to hide different representations of file handles on different platforms. The file handling functions all take a pointer to a FILE structure and you as a user never really get to know or care about what this structure contains. It also enables you to have multiple files opened at once. 3D objects can also be good candidates for this as their representation can vary so much from one platform to the other. If you can find the right set of functions to handle and draw them, you will never have to know how they are actually stored in memory.

The secret to this is to try to think in terms of objects you can encapsulate things into. This is the base of object-oriented programming and forces you to think about what kind of interfaces you want to provide to these objects. Never provide direct access to the structure members, instead if speed is an issue keep access methods but make those inline. It can also help to think in terms of actions on the variables rather than directly modifying them. Using rotations or translations rather than accessing the matrix’s internal values can also modify a transform. This will still work if you change the internal representation to, say, quaternion. In general function names using verbs will make your life easier.
If too much data needs to be exchanged between the application and the object, or if too many conversions take place, you should rethink the level at which the encapsulation takes place in your design and try to place it at a higher level. This will enable you to move all these transitions inside a single object and probably optimize them greatly. On the other hand be careful not to end up with a game object, which tries to be everything for everybody. Granularity of encapsulated objects extremely important, you want to strike the balance between portability on the one hand and well-optimized-for-the-platform on the other. As we will see in the next chapter, the larger your granules are the more chances you will have for optimization inside your API.

**Indirection layers**

You can think of this as data abstraction for functions. In the same way as hiding data can help portability, hiding some low-level processing behind a common set of functions is also a major part of writing portable code. Some people will refer to this as the 90/10 rule of cross platform development, 90% of platform independent code for 10% of platform specific code. Numbers published by John Carmack about his experience on Quake3Arena certainly shows this to be true and easily achievable.

Your goal is to avoid putting platform specific code in the middle of your programs. Any low-level code is usually platform specific and should be hidden behind function calls. For speed issues these calls, if to small functions, can be made inline while still preserving the benefits. The question then is: How low is too low-level? Usually if an indirection function is called a lot, this is a good sign the indirection level could be too low. This choice is crucial to good performance since it will affect the ability of a specific implementation of a function to optimize its inner workings.

An example of this is the choice between a portable DrawTriangle() or DrawObject() function in a graphic library. The first one might enable you to have more common code sitting on top of it, but the second one leaves more freedom to optimize platform specific code and would probably be the better solution in a game development environment. This also means that it can be a good idea to pass extra information to the platform specific code, even if a specific implementation does not use all of it. Try to think ahead and not put too many restrictions on your code. Experience with many environments and platforms will enable you to make better choices in this area.

Input and output functions are usually the best place to start implementing indirection layers. Don’t forget that graphics code is a kind of output and can be a very sensible area to work on in your games. Try to remember that functions like memset() and memcpy() can be extremely optimized on different platforms since their purpose is simple but can be executed in so many ways. This is the kind of ratio between functionality and performance that you should try to
achieve in your own libraries.

**API design**

The next step to design your indirection layers is to package them into portable libraries, and this brings us to another side effect of portability concerns. Carefully designing the API of your libraries can help make your code a lot more portable and especially easier to port to a new platform.

The first thing to be careful about is namespace clashes. Some platforms might contain functions that are named like yours and this could force you into a painful round of search and replace. Both C++ and Java have solutions to this, C++ with the ANSI namespace keyword and Java with its concept of packages. In C you can always use a naming scheme such as MYLIB_functionname(). It is a good idea to use a similar scheme to name your own data types and structures.

Try to avoid using conditional compilations like with the #ifdef directive in C. Although this might seem like a convenient solution to portability issues, it tends to make your code hard to read. Most important it effectively splits the code into two different programs with all the problems that this brings in terms of testing and maintaining, but doesn’t bring any of the advantages of doing so explicitly. Storing different version of a function into different files might create redundant code but will improve readability and make sure that weird combinations don’t happen when some keywords are defined and not others.

As we mentioned before system dependencies should be hidden behind interfaces and this is usually a good point at which to focus your efforts on clean interfaces that provide enough information for future implementations. Take for example a string-handling library. Ideally you would want this to define not only a string and a char data type, but also a set of functions to handle strings, calculate their lengths or copy them. This has to be provided without any knowledge of how strings are stored to support future extensions to Unicode for example. Ideally you want to make sure that you provide enough functionality and flexibility so that other implementers on different platforms won’t feel the need to go behind the scenes of your interface. Doing this for all your different components is definitely not a simple task from a design point of view. Again experience makes this a lot easier.

Try to be consistent, whether it’s in the way your functions are named or if the type or order of the parameters they take. It’s never a good thing to have to look up the documentation every two seconds to use an API, even more so if there isn’t any documentation. If you provide system specific functions, make it easy to guess how they are called from one platform to another. Also make sure that you mark them clearly as system specific functions so that nobody wonders why they aren’t provided on another platform. Some parameters can be provided but ignored in some implementations. Again these should be clearly marked as such, naming them hints for examples, to save your users from unpleasant
surprises.

Most important make sure that no implementation of you routines have hidden side effects that some users or platforms might rely on for the high level code to work properly. Each function should do exactly what it is named suggests, no more, no less. If your subsystem has different states then the state it is in after a particular function call should be obvious and consistent across implementations. If for any reason you need to have some hidden functionality or feature you should document this extensively to make porting by another person easier. In the end, keep in mind that ported code is new code and as such should always be thoroughly tested.

Basic utility functions

One of the first class of portable functions you will probably consider is file input/output. So many things can be different on each platform that it is usually a good idea to go beyond the FILE structure and provide means to open and read files from hard-drives, CD-ROMs or game cartridges. Filenames can be replaced by file indices for space efficiency and a request/callback scheme can be implemented for future support of asynchronous loading on some platforms.

As mentioned previously rendering is also an output and accessing it or setting it up can be made very easy. In the next part we will look at a screen setup library that enables you to open rendering screens on multiple platforms using the same interface. Basic drawing functions like lines, rectangles or font routines can also be helpful, especially for debugging information. You can also write a common internal bitmap format to enable you to load various formats and still handle them with the same set of functions. I personally started working on a material-handling library, which hides textures, blending modes and single or multiple pass operations from the objects a material is used on. This will hopefully not only enable portability but also retain performance and make quick prototyping of texture effects easy.

Math and most geometry functions can also be encapsulated to make the most of hardware acceleration, which is popping up on various platforms. Again in this case you should give the library something to work with and encapsulate vertex buffers rather than single points information. The more the platform specific library can optimize the better.

Sound is also good choice. With the support for 3D sound slowly becoming standard, you can still write a library that would in some cases emulate 3D sound support or maybe ignore it altogether and offer various degrees of support on various platforms.

Memory management should also be part of your bag of tricks. Consoles usually can lack in this area, or for some of them support can actually be buggy. Memory on consoles is usually scarce and fragmentation can kill you since running out of
memory is not usually an option. A lot of this can be avoided by implementing a very simple stack based heap system, which totally eliminates fragmentation. On PC one of the problems people encounter has to do with the operating system's virtual memory implementation. If it suddenly decides to page out some of the memory used by your game the frame rate might come to a crawl. A few solutions exist for this, none of them perfect. One of them is to allocate a buffer large enough to hold your game but smaller than the smallest pageable amount and allocate your memory exclusively out of this buffer. Again a library can hide all this specificity.

If you plan on translating your game to a language that uses a different alphabet then you should definitely have some string handling functions before development start. Porting string support can be a nightmare especially going from a standard 8-bit to a 16-bit Unicode character format.

Finally endian-handling functions for different processors will definitely come in useful. As long as you decide which way your files are stored if they are binary, you can write a set of functions that will 'read' memory locations as big-endian or little-endian. Of course this means every platform will have one of the two sets performing absolutely no operation and just passing the value along.

**Portable file formats**

Another area where portable code is becoming more and more useful is for tools development. Many common processing tasks for games, like level conversions and such, are very often offloaded to separate machine. In most cases you need to run these tools on a lot of very cheap machines or, at the other end of the spectrum, you need it to run on a operating system which will make the most of your multiprocessing capabilities. Either way writing your tools with portable code will enable you to use them on whatever the best platform turns out to be for your situation.

One of the keys of writing portable tools is to write and use portable file formats. Most tool pipelines nowadays are made of multiple stages, which can be on different platforms. This requires a portable exchange format together with portable routines to read and write it. We already mentioned that endian-ness issues can cause a lot of trouble but data type size differences can also make you do a lot of converting before you can start working with the data.

One easy way to make this all happen is to use ASCII as the base of your file format. The benefits of writing your files as plain text are numerous. It is portable, easy to generate and easy to parse. One of the best things you can do with text files is process them using the many scripting languages available like Perl or Awk. This is something most people on PCs nowadays will usually not think about since PC applications are usually built as huge vertical monolith instead of reusing separate modules. Think about it, being able to pop an editor window opened to check or modify a file is a great help.
Some people say that storage quickly becomes a problem. Realistically this becomes less and less of an issue since storage prices are enabling us to store gigabytes and gigabytes online for lower and lower prices every day. If you really have storage problems, have no fear. Text happens to compress extremely well and this could be used to archive less-often used or backup files.

While text has no endian-ness problem, it does have a line terminator problem. Windows platforms like to terminate their lines with both a carriage return and a line feed character whereas Unix platforms, for example, usually only use a carriage return (the line feed being implied). This is usually not a very big deal and can be handled at the parser or the file processing level with ease.

One of the other great advantages of using text as a file format is that it enables you to modify your memory structures while staying backwards compatible. The reader itself can fill missing fields from previous formats with default values.

In most cases, text isn’t a great idea for run-time data. You might not be able to spare the memory wasted by parser and processing routines or you might not want to have your data easily parsed by everybody else. In this case you can very easily write converters to output platform specific binary format but still use text for exchange and storage of you game data on your network.

Is Java the future?

Write once, run everywhere is Java’s motto. This ideal is possibly an unreachable goal but as in other things, the process of aiming for it brings many good things. One mistake that people make is to misunderstand the difference between ‘Java: The language’ and ‘Java: The runtime environment’. Many of the issues people bring up about Java have to do with the virtual machine or the libraries rather than with the language itself.

Object oriented programming is a great thing but in the process of trying to stay compatible with C, C++ has brought along a lot of the problems involved with C. A lot of C++’s best features like templates, operator overloading or multiple inheritance turn out in the long run to be curses rather than blessings. In short it is far too easy to write bad C++. Java does away with all this and started with a blank page to try and design a better object-oriented language.

Java source code is compiled into machine-independent byte code, which is then executed on a virtual machine or JVM. The JVM is currently emulated in software on all target platforms but some projects are underway to build Java processors in the future. You can add native code to a Java program by using the Java Native Interface or JNI. This separation of low-level native code and high-level portable code is exactly what we’ve been discussing here.

A few game companies are already using this system to take advantage of the
commercial JVM’s features like Just In Time compiling or JIT. Unfortunately JNI can be very cumbersome to use and in some cases just plain inadequate. Argument passing from native functions to Java functions is slow and should be kept to a minimum. Here again the point of separation between low and high level is crucial. Another issue is security if you allow people to download user-defined code, say, over the Internet. This can be a great feature for your game and can allow people to not only expand your game’s data but write their own mechanisms too. The downside is that people would also be able in some case to run their own security managers on the commercial JVM and therefore bypass any security measure you set up.

So if after looking into it you decide that commercial JVM and JNI is not for you, you can still benefit from all the JAVA language has to offer. The solution: write your own JVM. Be warned, this is not for the faint of heart and can turn into a major undertaking to get right. The advantages are that you can define and use your own interface between native and Java code. This is likely to suit your needs better and most likely will be faster. Your JVM can also use your own garbage collection scheme, which can be tailored for game development. There is no particular standard that needs to be followed when designing a garbage collector scheme for your JVM. You are free to be extremely creative and come up with a solution that will provide the best run-time performance for your situation. You could also have no garbage collection at all if your memory requirements allow for it.

Both these areas can provide you with a lean and mean scripting language machine. Of course there are downsides, mainly that you will not benefit from the work of others in the JVM field and will have to maintain and update yet another piece of code.

A lot of people mention speed as the main reason why Java will never make it into the world of game development. Look into it for yourself. There are games being developed right now using Java as their scripting language. Issues like speed of execution never stay an issue for very long in the world we live in and you should make sure you’re ready to make the right choices for your future projects. Be prepared.

**Conclusion**

I hope this will have given you a taste of what to look for when writing portable code. Remember that beyond all the techniques and tools of the trade, there are far more important issues to writing good, modular and portable code. While making this paper I have had the chance to talk with many people who agree with me that, as a developer, your code is your most precious resource. Together with keeping the rights to it when dealing with publishers, making sure a lot of your code is portable will make you better equipped for the future and in effect will make all your hard work and trouble worth something.
Expand on the ideas presented here and find out what works best for you. Just like anywhere else, the more experience you get the better.

About the Author

Didier Malenfant has been programming for over a decade and has played games on every platform he was exposed to, starting with a Commodore PET and a game called flies a long long time ago.

He has a passion for performance coding and porting his code to whatever looks like fun. He has experience on writing code for and porting to/from the Commodore Amiga, PC, Sony Playstation 1 & 2, Nintendo 64, Gameboy, Sega Saturn and Linux boxes.

His credits include working on games like UEFA/Ronaldo Soccer, Clayfighter 63 1/3, VR Baseball 97, Heart of Darkness, Wild9, RC Stunt Copter, Sacrifice and Crash Team Racing at companies like Power And Magic, Interplay, Shiny Entertainment and Naughty Dog, Inc. He is currently working on Naughty Dog’s first Playstation 2 title.

Bibliography

[Maguire 93]

[Kernighan & Pike 99]

[Klein & Regier 98]

[McConnell 93]

[Gamma, Helm, Johnson & Vlissides 95]
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, (1995).

[Kreimer 99]

[Huebner 99]

[Saladino 99]