# Experiences in Portable Mobile Application Development

Antti Kantee and Heikki Vuolteenaho

Helsinki University of Technology

**Abstract.** In the software world portability means power. The more operating environments you can support out of the same code tree means more potential users for your software. If done right, additional platforms can be supported with little extra maintenance cost. If done wrong, maintaining additional platforms will become a veritable nightmare.
This paper describes experiences undergone when creating truly portable software. Our software is a real time rendered 3D map and messaging application, which runs on UNIX (Linux, Mac OS X, NetBSD), Windows 98/2000/XP, Windows CE and Symbian Series 60. It is Symbian which makes this mix of platforms interesting and challenging. However, with the knowledge of potential problems, we found that this set of platforms is totally manageable for a portable mobile 3D application.

## 1 Introduction

Traditionally, in the UNIX and C world, portability has come to stand for the ability of a software to deal with differences imposed by the underlying CPU architecture, such as byte order, pointer size or alignment constraints [4, 5]. Other usual suspects for hindering a porting process are standard library or system interfaces either missing or behaving differently. By carefully programming against POSIX and ISO C provided interfaces and avoiding making assumptions about the compiler or underlying hardware, it is possible to achieve a fairly high level of portability, even between UNIX and Windows.

However, when a completely different kind of system, Symbian, is introduced into the picture, the rules change. All assumptions which used to hold in the UNIX and Windows environments may no longer be valid. This does not necessarily make things more complex or difficult. The major factor of difficulties for having Symbian within the sphere of portability of a software is basing key design elements on non-valid assumptions.

This paper describes the issues encountered in developing a mobile 3D application written in C. In Chapter 2 we describe issues specific to Symbian while Chapter 3 concentrates on issues affecting all platforms.

### 1.1 The software: mLOMA

mLOMA [13] (mobile LOcation aware Messaging Application) is in its essence a 3D map application optimized for mobile devices and built on top of OpenGL [9]and GLUT [6]. The mLOMA client can be used to browse a real time rendered 3D scene with a framerate acceptable for interactive use. It features a route guidance system and support for GPS location tracking. A server component is also provided. If a network connection is available, clients can receive up-to-date information on the model and interact using the server. Users can track each others' locations and communicate using messages. Messages can be public or targeted to individual users and they can be attached to any points in space or the model.


**Fig. 1.** mLOMA client running on Pocket PC and Symbian Series 60 platforms

Since mobile terminals do not feature 3D acceleration in hardware and are limited both in terms of available CPU power and memory, the implementation must try to limit resource consumption to a minimum. This is in part done by doing a PVS precalculation on the 3D scene [13] and the rest is accomplished by non-wasteful C programming.

### 1.2 Portability

For defining portability, we first separate the whole idea of portability into two different categories: code portability and concept portability. Concept portability refers to the ability to implement an idea on a variety of platforms. For example, a user interface requiring a cursor is not completely portable to all mobile computing platforms, since some platforms lack a pointer device. On those platforms it is possible to emulate a pointer device, but this will affect usability and is therefore visible to the end user.

Code portability is the ability of software to run common lines of code between the various platforms it is portable to. The code lines which cannot be shared result from differences in the various platforms either in system interfaces or from the hardware. Code portability involves crafting interfaces which abstract the underlying platform functionality where it is different. Abstracing does, however, come with a price of call indirection and increased coding effort, and therefore should be carried out only where necessary. We use the term *machine independent* (MI) to describe code which runs on all platforms and the term *machine dependent* (MD) to describe code which runs only on a certain platform. Software with code portability will have a high MI/MD ratio in terms of lines of code.

Implementing a certain functionality multiple times for different platforms when not really necessary is in its essence confusing code portability with concept portability. The resulting user-perceived functionality will be the same, but the cost of maintaining several different implementations is much higher and will probably lead to broken platforms as code evolves [7, 11]. It is easy to see why, since as the number of lines of code shared between platforms goes down, the portion of the codebase that can be tested on a single platforms goes down as well.

### 1.3 Symbian

Symbian is an operating system designed primarily for mobile phones and other mobile devices. Conserving limited resources is a priority, and several programming practices used on Symbian encourage it. This makes working with Symbian in a multi-platform project a challenging task.

While fully understanding Symbian requires closer attention, this paper does not cover the architecture of Symbian and such studies can be found in dedicated literature [3, 17].

## 2 Porting to Symbian

The mLOMA client was originally written for Linux desktops, Windows desktops and Windows CE PDA devices. Symbian Series 60 support was not originally planned. However, once capable mobile terminals became available, support was required.

### 2.1 GLUT

Symbian lacks a platform-provided GLUT [6] implementation. GLUT, tersely put, works as an event handler in between the application and console (windowing, input devices). Generally, implementations never come out of the main event loop until they detect the quit command being issued. However, due to the active object scheduling scheme used in Symbian applications [12], we cannot run continuosly in the main loop. Instead, we need to periodically relinquish control of execution and generate events to regain control.

A subset implementation of GLUT for Windows CE had been done earlier in the project, since GLUT was not available for it at that time [1]. However, this implementation is mostly incompatible with the Symbian programming restrictions and in addition was built on top of the normal application-transparent preemptive scheduling principle.

We ended up with two separate GLUT implementations. While this is in disagreement with our portability rule set forth in Chapter 1.2, it is important to note this as an acceptable and even encouraged exception to the rule. First of all, code lines are not shared because there are not very many lines to share: approximately 415 of the total 496 lines in the implementation are completely specific to Symbian. Second, the GLUT interface is very unlikely to change and therefore require platform-specific maintenance.

### 2.2 Writable global data in DLLs

Symbian GUI applications are built as DLLs and Symbian does not allow writable global data in DLLs [3]. There are two choices: build an EXE instead of a DLL or get rid of all global writable data. The first option makes building a traditional Symbian GUI very complicated [21].

Each thread can store exactly one word of global writable data in a slot called Thread Local Storage (TLS). We put all our global variables inside a (rather large) struct and push the struct pointer to TLS. Accessing the TLS is slower than a regular function call, in our testing it was roughly 20 times slower. Because of this, we often pass the pointer as an extra parameter in often-used function calls. However, we noticed that passing "a pointer to globals" was detrimental for the interface development within the client. Especially junior programmers had the habit of crafting interfaces with nothing but that pointer passed.

Symbian tools are not helpful in locating global writable data in the program, as they do not even specify the offending module:

```
ERROR: Dll 'MLOMA[102048D8].APP'
  has uninitialised data.
```

Symbian developers have found ways to extract the offending source modules and variables [18], but they are not very practical. A much better way of locating

[1] However, GLUT|ES is now available for Windows CE.

modules and code fragments in violation of this restriction is to use a typical UNIX command sequence:

```
find . -name \*.o                   \
  | xargs nm -o --defined-only      \
  | awk '$2 !~ /[tTrR]/{print $0}'
```

If the filter encounters a symbol type that is not *text* or *read-only data*, it prints the module and symbol name. After this, it is easy to use a text editor to search for the culprit symbol in the offending module.

Notice, that for this to work, `nm` must support the object format of the objects it is supposed to examine. It is most natural to run this on a UNIX development platform against UNIX objects, although it should be possible to use a UNIX-hosted toolchain, such as the one provided by the GNUPoc project, for running it against Symbian object files.

### 2.3 Stack size

In C programming it is customary to allocate memory for local operations from the current stack frame, from where it will be automatically freed when upon return. In most environments it is safe to assume at least tens or hundreds of kilobytes of stack space, making allocating fairly large objects from the stack possible.

Symbian has a comparatively small default stack size (8kB). Large allocations from stack are therefore impossible. On the device, running out of stack will lead to a crash, but the emulator build fails on purpose if it runs into a dangerously large (>4kB according to our tests) stack frame [2]:

```
MAIN.obj : error LNK2001:
  unresolved external symbol __chkstk
```

To remedy this problem, all large allocations had to be moved from the stack to the heap. It involved some code restructuring, but was mechanical work.

### 2.4 Texture loading

The mLOMA client needs to load JPEG and PNG images to show textures on the 3D map. On platforms other than Symbian the open source libraries libjpeg and libpng are used for this purpose. However, these libraries have not been ported to Symbian. Porting them is problematic at best because of the writable global data limitation discussed in Chapter 2.2. Symbian does have a native API for image loading and we use that instead.

The Symbian image loading APIs are asynchronous (non-blocking), while on the other platforms they are synchronous (blocking); the client was designed

[2] Notice that running out of stack is still possible in case of a deep enough call recursion without any single stack frames running over the warning limit.

fairly heavily on synchronous interfaces meaning that it expects the image to loaded once the image loading call returns. We used a nested active scheduler loop to effectively make the loading process appear synchronous [1], although this is strongly discouraged [16]. We ran into several problematic situations because of this. Normally application code handling an event runs without interruption (non-preemptively). But while the image loading function is blocking (using nested scheduling), the nested scheduler is free to schedule other active objects requiring attention. This causes for example reentrancy problems, as we enter GLUT through the active scheduler (Chapter 2.1). Several workarounds were introduced into the code, but, needless to say, these problems were extremely difficult to locate and the resulting bug symptoms may occur only in rare corner cases.

One possibility would have been to convert the entire application to deal with asynchronous interfaces. This, however, would have been poor choice unless the previously synchronous image loading backends would have been converted to asynchronous also. The reason is that different behaviour would have introduced platform specific bugs. Converting the sychronous backends to asychronous would have meant introducing threads into the program. The authors generally consider threading to be harmful [20]. Specific to this case, we probably would have run across different platforms exhibiting different threading behaviour.

A better solution to the problem came from an isolation technique [14] used, amongst other locations, in the popular OpenSSH networking daemon. In MD Symbian startup we create a thread whose only function is to handle texture loading. Communication between the application execution context and texture thread happens through a synchronization primitive. The application first triggers the texture loading and then sleeps on top of the synchronization primitive. When texture loading is complete, the texture thread triggers the application to continue executing. After replacing the nested scheduler with this scheme, all inexplicable crashes disappeared. We propose that all who want to emulate sychronous interfaces on Symbian use this method.

### 2.5 Stdio problems with locales

The stdio call families of `printf` and `scanf`() have a problem with the thousands separator and decimal separator on Symbian. It seems that modifying the application's private locale does not affect the separators at all and using the system-wide locale it is only possible to change the characters, not totally remove them (more important for the thousands separator). This leads to a situation where it is not possible to reliably read and write floating point numbers from using an externally provided source, such as a config file.

Third party options were not available due to licensing or problems with globals (see Chapter 2.2), so we crafted our own implementations called `fgetfloat`() and `fputfloat`(), which read and write, respectively, a float using the given stdio stream. These are suboptimal, because they disrupt code flow.

In retrospect, the right choice would have been to drop floating points from files all together.

## 3 Problems & solutions, tools

### 3.1 The build process

Currently, using the native build systems for each platform, we have different build systems for:

- UNIX desktops: Linux, Mac OS X, NetBSD (make & GNU'ish toolchain)
- Windows (MS Developer Studio, Visual C++)
- Windows CE, PocketPC 2002 (MS Developer Studio, Visual C++)
- Windows CE, PocketPC 2003 (MS Developer Studio, Visual C++)
- Symbian Series60 V1 (makmake, Visual C++/gcc)
- Symbian Series60 V2 (makmake, Visual C++/gcc)

This means that adding a source file to the project or for example adding a project-wide C preprocessor definition requires modifying seven different files. The MS Developer Studio projects are not even meant for hand-editing, so touching them from outside the actual IDE is dubious practice.

As the number of platforms increases, the maintenance overhead grows soon beyond acceptable limits. If various platforms require a lot of manual editing to keep up, they will likely end up being out-of-sync with the main development environment. At one point after a project has grown onto multiple platforms, an attempt to unify the build procedures for all platforms should be made.

We will attempt to make this unification for mLOMA in the future. One possibility is to autogenerate the Symbian makmake project files from the UNIX Makefiles and use GNU Make in the Windows builds. The latter is accomplished by a well-known scheme of having a MS Developer Studio project file, which just contains the instructions to run GNU Make for building the project and leaves the details of the build process up to the Makefile.

Another possibility for accomplishing the same effect would be to autogenerate the project files. The UNIX Makefiles could easily be used to act as the autogeneration facility, since they are written in a clean fashion separating input data (e.g. source file names) from rules (e.g. how to product an executable). It should be fairly simple to autogenerate the .mmp files for Symbian builds and there is evidence that autogenerating MS Developer Studio project files is possible [8], even if not directly available.

In a sense, the build system can be equated with program source code and the concept discussed in Chapter 1.2. A portable program will also have a portable and flexible build system.

### 3.2 Local language support

The mLOMA client application needs to support various different languages, as it is aimed primarily for tourists, who benefit from local-language support. This means that our software cannot include hardcoded messages to the user in the middle of code, but rather the code must contain identifiers, which can be translated on the fly. While is it well-known how to accomplish this on any given platform, for example Linux [2], the problem is finding something usable on all platforms; for example even UNIX vendors cannot agree amongst themselves on should they use `catgets`() or `gettext`().

**Message database** The problem here is not so much abstracting the programming interface as it is abstracting the message database. If we were to use the native i18n services of each platform, it would require us to input the translated messages into several different databases. This would, first of all, mean learning the tools of the various message catalogs. Second, and worse, this would most likely mean that some of the catalogues would be out-of-sync with others, as development takes place on different platforms.

Since we only need to do simple key-to-text translation, a self-authored component was created for translation purposes. This was done by writing a script in *awk* for translating the input text into lookup tables which could be used from within the code. A selection of the input text format is presented in Table 1. This table is translated by the script into code usable at runtime. For all except Symbian, this means creating tables of C strings and for Symbian this means creating resource files and appropriate descriptor tables.

**Table 1.** Selected example translations from master_ui.txt

```
!fi
FORM_ROUTE_FASTEST Nopein reitti
MENU_HELP Ohjeet
!en
FORM_ROUTE_FASTEST Fastest route
MENU_HELP Help
!et
FORM_ROUTE_FASTEST Kiireim tee
MENU_HELP Abi
```

After the translation tables have been built, they are compiled into the client software and can be accessed through a call to a function a bit misleadingly named `localize`() [3], for example the call `localize(UISTR_MENU_HELP)` would produce the string "Ohjeet", "Help", or "Abi" depending on if the selected language was Finnish, English or Estonian, respectively.

[3] After all, the call only gives a translation of the string. It does not, for example, convert monetary units, dates or thousands separators to local conventions.

**Runtime interface and language selection** Deciding which translation to use runtime is equally, if not more, difficult than deciding how to do the translation. In a perfect world it would be possible to do this while holding on to two guidelines: changing the language should be similar on all mLOMA platforms and the method for changing the language should be in alignment with the platform's native way of doing runtime language selection.

POSIX does not specify anything about local language support in the locale interface, so we cannot use the `setlocale`() interface for querying the language: `LC_MESSAGES` would be close, but not being a part of POSIX it is not defined by Windows. Environment variables (`getenv("LANG")`), are not supported by Windows CE. Symbian has its own framework.

Currently all platforms use specific implementations: UNIX and Windows use `getenv`(), Windows CE uses a compile-time selector and Symbian uses its own resource file framework, which allows the application to select the correct locale at application startup. The future is undecided, although all things considered, a configuration file entry might be the simplest choice even though it means going against established platform conventions.

### 3.3 Memory management

Our memory resources are different from modern GUI applications. We have to assume an extremely small amount of available memory, around 5MB in the minimum configuration. In addition, there is no secondary memory on the Symbian and Windows CE platforms, so we need to control memory management ourselves.

Our scheme for dealing with the memory limit is simple: we have a wrapper around malloc, `memory_malloc`() [4], which checks if memory allocation fails, frees all memory available to be free'd and tries to allocate the same amount of memory again. Only if this second allocation fails, the wrapper will return a failure to the caller and the caller must deal with the situation.

For parts of allocated memory it is easy to tell if it is currently in use or not. A lot of memory usage comes from the geometric model and associated textures used to render the 3D scene. This static information is easy to reload if it is later required. In a sense, this type of operation can be compared with a practice used in some operating systems, where the read-only text segment is not paged out to secondary memory. To perform a memory sweep in case of a shortage, we simply walk the list of textures and meshes and free ones which are currently not in the field of view.

**Tracking allocated memory** Symbian is designed for low-memory environments with long-running applications and tries to encourage proper memory

[4] For the diversity of platforms we have, it is much simpler to have a completely different symbol name for the memory allocation function than it is to try insert a wrapper using the same name as the platform malloc and still try to call the platform malloc from within the wrapper.

**Table 2.** CPP tricks for memory allocator interface

memory.h:

```
#ifdef MEMORY_DEBUG
void *memory_malloc(size_t, unsigned /*magic*/,
                    const char *, const char *, int);
#define memory_malloc(a,b)                          \
        memory_malloc(a,b,MEMORY_DEBUG_MAGIC,   \
        __FUNCTION__,__FILE__,__LINE__)
#else
void *memory_malloc(size_t);
#endif /* MEMORY_DEBUG */
```

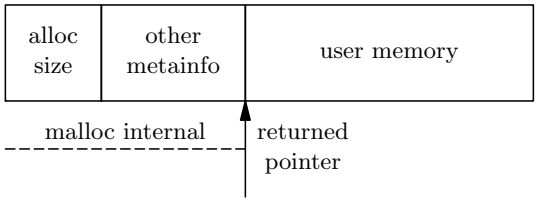
memory.c

```
#ifdef MEMORY_DEBUG
#undef memory_malloc
void *omamemory_malloc(size_t);
#else
#define omamemory_malloc memory_malloc
#endif /* MEMORY_DEBUG */
```

management habits to avoid memory leaks. This exhibits itself by the debug builds panicking at exit if any allocated (non-freed) memory remains. Most UNIX and Windows programs do not free their memory upon exit, as keeping track of all memory allocations requires extra work and in any case the operating system will unmap the pages of an exiting process.

While we could simply not care about the issue, as Symbian release builds do not complain, playing along with the platform memory management functionality seems like a correct option. This mandates us to do memory tracking if we wish to avoid two related problems: the Symbian debug builds panicking upon exit and standard desktop programming practices contributing such errors. While a tool such a Valgrind [10]would work perfectly for this, normal development cycles are not usually done within it and since we already feature our own `malloc`(), coupling tracking with it is the right choice.

**Fig. 2.** Memory meta information reserved by our `malloc`()



Some `malloc()` implementations register the amount of memory reserved in extra space right before the pointer returned to the caller [19], also illustrated in Figure 2. Our idea is to use this same space to achieve an $O(1)$ lookup for memory allocation chunk describing metadata. Using the information contained in the chunks of metadata, the application prints out diagnostic messages when exiting:

```
non-free'd chunk at 0x8a16a1c, size 0x24
main/mother.c:mother_init(), line 55
```

This indicates that memory reserved from the module mother.c, in the function `mother_init()`, on line 55 in the module was not freed before exit. Upon seeing this message, it is much easier to figure out what is going wrong than from having the program crash on the Symbian platform with the following error message:

```
Program closed: MLOMA ALLOC: 132df248 0
```

By using certain C preprocessor tricks illustrated in Table 2, memory allocation works, without any modifications to calling code, for the memory wrappers compiled with or without `MEMORY_DEBUG` and the calling code compiled with or without `MEMORY_DEBUG`. The tracking layer is implemented directly as `memory_malloc` and it calls the backend called `omamemory_malloc`. If the memory module is compiled without `MEMORY_DEBUG`, the call to the tracking layer is simply skipped by renaming the `omamemory_malloc` symbol. In the opposite case, a caller compiled without `MEMORY_DEBUG` will not pass the correct `MEMORY_DEBUG_MAGIC` signalling that the rest of the arguments are garbage and should not be examined.

**Table 3.** Compiled (gcc 3.3.3, NetBSD/i386) total size of memory free'ing subroutines

| optimization flags | resulting code size (bytes) |
|---|---|
| -O0 | 2485 |
| -O2 | 1770 |
| -Os | 1534 |

We could of course use the metainformation to free all memory, but it was decided against that. First of all, the code size (Table 3) for the freeing code is insignificant when compared with the allocation overhead, at least two pointers per allocation. Second, and more important, an automatic solution would not be in alignment with the original reason for freeing all memory.

### 3.4 Networking

The networking code used in the client is divided into four different layers.

1. platform-provided networking interface

2. platform-specific implementation backing our networking abstraction layer
3. abstraction layer for platform networking interface
4. protocol unit serialization and deserialization layer

**Platform networking interfaces** The underlying implementations and their limitations must be understood before abstracting them can be attempted. Our platforms are divided into two categories: the Berkeley-influenced [15] platforms such as Linux, Windows and Mac OS X in one category and Symbian in the other.

Symbian uses active objects to provide an asynchronous interface to normal socket operations. The major difference to the normal Berkeley-style interface is the fact that Symbian sockets do not support synchronous operation at all.

**Platform-specific implementations** The differences within the Berkeley category are subtle enough so that grouping them under a single implementation is feasible and painless.

The relevant differences we encountered between the UNIX implementations and the Windows implementations can be seen from Table 4. All of these problems could be circumvented by simple cpp macros and a typedef.

**Table 4.** UNIX and Windows socket differences

| | UNIX | Windows |
|---|---|---|
| initialization | *none* | `WSAStartup`() |
| error query | myerr = `errno` | myerr = `WSAGetLastError`() |
| errno values | `EINPROGRESS` / `EAGAIN` | `WSAEWOULDBLOCK` / `WSAEWOULDBLOCK` |
| ioctl call | `ioctl`() | `ioctlsocket`() |
| shutdown() arguments | `SHUT_RDWR` | `SD_BOTH` |
| sockaddr length type | `socklen_t` | *none* |

The Symbian implementation is completely disjoint from the Berkeley-family implementation. It uses its own data structures, descriptor buffers and active objects to interface with the Symbian platform networking interface. Conversion from descriptor buffers to buffers in machine-independent code (`char *`) and vice versa is currently inefficiently done using memory copy.

**Abstraction layer** As noted above, the only major difference between the two families of platform network interfaces is Symbian's inability to do synchronous operation. This is not a hindrance at all, since being a single-threaded application, asynchronous network operation is the only choice if we do not want to block the entire UI in case of e.g. network congestion.

For managing connections, we need two different interface functions: one for initiating a connection and one for disconnecting. The asynchronous nature of the TCP connection is handled internally. In case the connection to the server is not successful, the situation is no different from the user perspective as a

**Table 5.** Machine Independent Networking Interface

```
int             network_init(struct network *net);
void            network_exit(struct network *net);

int             network_enqueue(struct network *net, uint8_t *data,
                                size_t datalen, int message_type);
struct netbuf * network_dequeue(struct network *net);
void            network_buf_done(struct netbuf *buf);

int             network_connect(struct network *net,
                                const char *address,
                                unsigned short port);
void            network_disconnect(struct network *net);
```

failed login and it will be treated as such: the network functionality will be unavailable to the user.

Network send and receive functions in a two-level fashion. Sending data onto the network first puts the data onto a network buffer list. This is done synchronously from the application point-of-view. We cannot directly always attempt to send data onto the network, since the network might be congested, the socket buffer therefore full, and sending would either block or fail, depending on if we were operating in blocking or non-blocking mode [15]. After data has entered the network buffer list, it is periodically drained onto the network using the GLUT timer functionality. Receiving data happens conversely: the network buffer queue is periodically filled by a function called from a GLUT timer handler and the application can read complete protocol data units off it synchronously.

To reduce the strain on memory allocation for the clients, this layer is not completely protocol-agnostic, but knows also about the application protocol framing mechanism we use, so that it can allocate memory chunks of the correct size for incoming transmissions.

**Protocol serialization layer** To avoid subtle but difficultly trackable incompatibility issues between the various client platforms and the server, the from- and to-wire routines are autogenerated from an XML representation.

The interface used to access the protocol unit contents is simply struct member access provided by the C language. A single PDU is always represented by a single structure and the structure representation is auto generated from the XML information. After all fields have been filled, the autogenerated `serialize`() routine is called to produce a byte stream representation of the contents of the structure. Conversely, `deserialize`() is called for a byte stream received from the network to fill out a struct representation of the same byte stream.

## 4 Conclusions and future work

Writing a portable mobile application for UNIX, Windows 98/2000/XP and Windows CE is simple when compared to the situation with Symbian. Symbian is a different type of system and many normal programming idioms were found to be unsuitable for Symbian. However, including Symbian produces a symbiotic relationship between the platforms: the requirements of Symbian keeps questionable programming practices down to a minimum while tools available on other platforms aid development on Symbian.

The scheduling model used by Symbian causes major problems: most platform functionality is a schedulable service, which in turn causes its interface to be asynchronous. For software with prior design elements based on synchronous interfaces, we showed an acceptable method for emulating synchronous interfaces on Symbian. Another major set of differences are memory limitations, both the lack of a read/write data segment on Symbian as well as the small amounts of main memory and lack of secondary memory on PDA/mobile devices.

When attempting to write software with code portability to multiple platforms, it is most important to understand the limitations and characteristics of each platform and make design decisions based upon that understanding. If platform expertise is not available at the beginning of the project, resources for some necessary development iteration to get the interfaces right should be allocated. The main goal is to make, as far as reasonably possible, all components either shared or behave similarly on all platforms. This will not only unify the user experience across various platforms, but, more importantly, reduce development, maintenance and testing effort.

Future work with the project includes unifying the user interface and program menu code: currently Symbian uses its native components while other platforms use OpenGL. In addition, unifying the build system to support a single project file across all our platforms needs work.

## References

[1] Matti Dahlbom. Image loading and color reduction. 2003. URL `http://www.newlc.com/Image-loading-and-color-reduction.html`.

[2] Pancrazio de Mauro. Internationalizing messages in linux programs. *Linux Journal*, 1999(March 1999).

[3] Richard Harrison. *Symbian OS C++ for Mobile Phones*. Wiley, 2003.

[4] Martin Husemann. Fighting the lemmings. In *EuroBSDCon*, pages 45–53, 2004.

[5] Steve Johnson and Dennis Ritchie. Portability of C programs and the UNIX system. *The Bell System Technical Journal*, 57(6):2021–2048, June–August 1978.

[6] Mark J. Kilgard. *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*. 1996.

[7] David G. Korn. Porting UNIX to Windows NT. In *USENIX Annual Technical Conference*, pages 43–57, 1997.
[8] Paul Kunz. Building with automake.
[9] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, 1993.
[10] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
[11] Geoffrey J. Noer. Cygwin32: A free Win32 porting layer for UNIX applications. In *2nd USENIX Windows NT Symposium*, 1998.
[12] Nokia Corporation. Symbian OS: Active objects and the active scheduler. 2004.
[13] Antti Nurminen and Ville Helin. Technical challenges in mobile real-time 3D city maps with dynamic content. In *IAESTED Software Engineering*, 2005.
[14] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, pages 231–241, 2003.
[15] W. Richard Stevens. *UNIX Network Programming*, volume 1. 1998.
[16] Symbian. Symbian developer library, 2003. URL `http://www.symbian.com/developer/techlib/v70sdocs/doc\_source/reference/cpp/AsynchronousServices/CActiveSchedulerClass.html`.
[17] Martin Tasker. *Professional Symbian programming*. Wrox Press, 2000.
[18] Paul Todd. Finding initialized or uninitialized static data in a dll. 2004.
[19] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.
[20] Robbert van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *ACM SIGOPS European Workshop*, pages 82–87, 1998.
[21] Peter van Sebille. EMame: a MAME port to EPOC Release 5 and Symbian platform v 6.0. 2001.