Macintosh and Windows 3.0: A Developer’s Perspective

Waldemar Horwat

Abstract

The Macintosh Toolbox and Microsoft Windows 3.0/3.1 are both powerful graphical environments for writing applications. Although many programming details of these two environments are almost identical, they follow quite different philosophies and differ in fundamental aspects. Neither is clearly superior, but their weak and strong points are very different. This paper examines and compares both operating systems from a developer's perspective, highlighting their differences in philosophy.

This paper also contains an appendix that defines object-oriented programming and abstraction and shows one instance in which they conflict. This may be one of the operating system/application interface conflict areas in the near future.

1. Introduction

The two main graphical environments for personal computers today are Microsoft Windows 3 and the Macintosh operating system. This paper compares these two environments from a software developer’s perspective. I’ve had opportunities to develop major applications in both environments and will include some of my experiences as examples; however, the comments are intended to be general and should apply to other application developers.

The scopes compared will be the general paradigms used by the two systems (section 2), important technical considerations (section 3), and development environments (section 4). Although features will be used as examples, this paper is not a feature comparison of the Macintosh and Microsoft Windows. Most of the features are duplicated in both systems, and there are other sources of information for readers interested in specific features—refer to [1] and [3]. Also, no attempt will be made to compare the two environments from the point of view of general users.

One can avoid some of the issues presented below by programming in a machine-independent environment such as XVT or Cognits. However, these environments do not completely insulate the program from the operating system on which it is running. Moreover, one may not wish to write a program for the lowest common denominator system; in fact, as the operating systems are getting more powerful and provide more services, it is becoming increasingly more difficult to design abstract interfaces that let a program run well under several environments. Thus, the issues in this paper are still relevant to developers using cross-platform environments.

History

The Macintosh was first introduced in 1984, although it borrowed heavily from Apple’s Lisa design, which was released in 1983. Microsoft Windows was first released in November 1985[1] but did not gain wide acceptance until version 3.0 in May 1990. Both of the systems have fairly long (in software terms) histories and are rooted in software technologies of the early 1980’s. The histories of the two environments are apparent from the programming interfaces and often cause grief, as will be shown later.

In the remainder of this paper, “Windows” will indicate either Microsoft Windows 3.0 or 3.1.

2. Paradigms

In this section, the philosophies and guiding principles behind the Macintosh operating system and Windows are compared on a general level—the languages which influenced the systems’ original designs, the environments’ attitudes towards object-oriented programming, and their emphases in simplifying certain tasks at the expense of others. This section closes with

[1] It was announced in November 1983—a few months before the Macintosh introduction!

a discussion of the levels of entropy in these two environments.

Languages

Possibly owing to their origins in the early 1980's, both the Macintosh operating system and Windows have Pascal-based interfaces to their system calls—Pascal calling conventions are used. The main C compilers for both environments use extra keywords such as `pascal` when declaring the interfaces to the system functions. Of course, one can write programs in other languages, but the interfaces to the operating system calls are limited to what one can express in Pascal—no variable numbers of arguments, and no C++ features.

Several other places, such as string handling, reflect the language origin of the original operating system design. Most Macintosh system calls use Pascal strings; Windows was less Pascal-dominated and its strings tend to be null-terminated or have explicit lengths. Equivalent functions that use C null-terminated strings are available as libraries on the Macintosh, but they incur extra overhead. Furthermore, they can cause problems in that the strings returned by the system (such as filenames) can contain nulls, which most standard C routines have trouble with. I am not stating that Pascal or C-style strings are better (there are also other alternatives), but just that the choice of initial language affects the "style" of operating system interfaces.

Object Orientation

Syntax

As a result of the Pascal/C origins of the environments, neither environment provides direct support for object-oriented programming such as using C++ classes. One can encapsulate system calls in object-oriented shells, but this leads to one of the following situations:

• Using one of the standard libraries like MacApp or Borland's C++ object library. This provides standard object-oriented interfaces, but the program size usually increases significantly and performance may suffer. The object-oriented libraries tend to do a lot more work and make more system calls than necessary by doing things like constantly focusing on the drawing port. Also, the libraries may be somewhat behind the state of the art when the operating system interfaces evolve.

• Writing a custom object-oriented library to encapsulate just the managers one uses. This can lead to good performance, both in terms of code size and speed. Unfortunately, everyone's libraries will be different, making it harder to read or reuse other programmers' code.

Using MacApp with C++ does expose one weakness which is due to its origins with Object Pascal—it does not support C++'s more advanced features such as constructors, destructors, multiple inheritance, or type casting. Thus, one does not entirely gain the advantages of fully encapsulated objects.

Structure

Aside from the language *syntax* (which is important), one can also ask whether the *structure* of the operating system is object-oriented. By this I mean whether system structures are presented as abstract, self-contained entities that can only be manipulated by functions and from which one can derive subclasses (see appendix A).

In this respect, Windows has a more object-oriented structure than the Macintosh operating system. Windows communicates with application windows and controls by registering an application's entry point and then sending it messages as appropriate. If the window or control does not wish to do anything special with a message, it can pass the message to a default handler specified by Windows. Thus, each window and control is effectively overriding a default. Windows also allows defining controls by overriding the behaviors of pre-defined controls, but this is difficult and has to be done by trial-and-error for the reasons stated in appendix A—when changing one area of a window by overriding the draw message, one cannot be sure that the window doesn't make assumptions about the size or shape of the area being changed. Fortunately, the default window and control functions don't seem to be doing that, but they could without violating any rules in the documentation.

The Macintosh is more procedural, in that the operating system relies on the program to dispatch events and has no direct access to a program's windows and controls. This is a disadvantage; for example, when a modal dialog is shown, the program's event loop is not executing until the dialog is dismissed, so none of the program's other windows can be updated. If part of another window belonging to the program becomes invalidated, a partial deadlock will occur. The workaround is quite messy [2].

While Windows' object design is better for most applications, it is not superior in all cases. One trouble spot in Windows' implementation of messages is quitting a program (or, worse, quitting Windows). The program gets one quit message and is then expected to instantly decide whether it wants to quit, and, if so, do it cleanly without receiving any more messages. This is fine if the main program's event dispatcher gets the quit message—it can just exit the program. On the other hand, if a dialog event loop gets the program's quit message, then it is difficult to cleanly unwind all of the pending calls on the stack precisely because the program is not procedural; the dialog's only choice is to return to the Windows dialog handler, but when that handler returns to the place in the program where the dialog was invoked, the program won't know that it is supposed to quit.

Philosophies

As with object orientation, the philosophies of the two environments differ significantly. The Macintosh environment is built from the bottom up—the various managers provide general, flexible interfaces, only parts of which are necessary to support the next higher level of managers; low-level managers are emphasized. On the other hand, Windows is constructed in more of a top-down manner—the higher-level managers are emphasized, and they provide larger building blocks than the Macintosh equivalents, but they are often inflexible.

A couple of examples should illustrate this. Defining a little, scrollable and resizable window for editing text is quite easy in Windows—much easier than working with TextEdit on a Macintosh and hooking it up to properly respond to various events. If you want a window for editing text in Windows, then everything is simple. But suppose you would like a scrollable and resizable window containing read-only text such that the program can either scroll to a specific position or determine which portion of text the window is showing? It turns out that such a simple change is impossible in Windows! Although defining a standard text window is easy, making trivial requests such as determining what text is in the visible portion of the window is impossible[2].

Another example is using Windows' Multiple Document Interface, which is a library for nesting windows inside other windows[3]. As long as what you want is standard, closable, resizable, zoomable subwindows, everything is easy. On the other hand, suppose that your application's documents should be nonresizable. Although Windows generally supports nonresizable windows well, the Multiple Document Interface insists that every document be resizable, have close and zoom boxes, etc. I was able to write enough message filters to prevent the windows from being resized and closed (by killing appropriate messages), and to zoom them back if they are maximized[4], but there is no way to remove the zoom boxes from the window's title bar! I could define my own window class whose windows are designed to look just like the standard ones but with no zoom boxes, but that is likely to cause compatibility problems—in future releases of the system software the standard window appearance could change, or windows could acquire more features which my windows wouldn't have.

The Macintosh doesn't suffer from the building blocks' lack of flexibility as much. On the Macintosh it is often hard to get something like a text-editing window to work, but once it works, one can change it at will. Thus, Windows lets one write "standard" programs quickly, while the Macintosh lets one write more flexible programs, but slower.

Diversity

The characters of the Macintosh and Windows are also affected by the diversity of the platforms on which they run. In 1984 programming the Macintosh was simple—there was only one version of the computer and its ROM, and additional hardware and software options were limited. Now, however, writing a Macintosh program for wide distribution is like wrestling an octopus—there are more than a dozen different computers to worry about, four different processors with and without floating point coprocessors and memory management units, several versions of the operating system, and plenty of other hardware and software options.

Writing a general program on a Macintosh requires the use of `gestalt` or `sysEnvirons`[5]. The idea is that programs should check for the existence of computer features before using them. While this is a good idea and much better than the situation before `gestalt` was introduced, in practice it suffers for the following reasons:

• It's difficult to write a program that can work with or without many combinations of system features. Worrying about which combinations of managers are present can make the programmer paranoid and obscure the structure of the program.

---

[2] I cannot state with certainty that this is impossible, only that Microsoft developer technical support spent several days trying to do it and couldn't figure out any way.

[3] The Windows user interface guidelines differ from the Macintosh, in that if an application has several documents open, they should appear as subwindows within one large window belonging to the application.

[4] Just killing the maximize message turns out to be unsafe. This is yet another example of the hazard of overriding from Appendix A.

[5] Assuming that they themselves are available; one has to check for the existence of these calls first! (Some of the libraries do that automatically.)

• It's difficult to test programs that use `gestalt` extensively. When new computers are introduced that include new subsets of features, many latent bugs in programs become apparent[6].

The first problem leads to "least common denominator programming." Developers write programs for the lowest platform for which the program will be distributed and then either cannot take advantage of or must duplicate the operating system features of higher-end platforms.

Apple has been exacerbating this problem by appending new operating system functionalities to the old ones instead of supplanting the old ones. A classic example is the introduction of the desktop database calls in System 7. These calls, while quite useful, work only on hard disks! They are worthless to many developers who wish to take advantage of them, because developers must still use the old, illegal hacks to access the desktop information on floppy disks. By introducing the new calls, Apple increased the complexity of the system without solving the developers' problem. This is an important reason why revisions of the Macintosh operating system tend to break applications.

The Windows world is currently in a better situation because there are fewer versions of Windows to worry about (nobody is coding for versions earlier than 3.0 anymore, and it's safe to assume version 3.1 in future programs since there is little reason for users not to upgrade). Also, compatibility rules are enforced better in Windows—see section 4 of this paper. Thus, while Macintosh programmers must use `gestalt` to case for various features of the system software to make sure that their programs run with Systems 6 and 7 (many Macintoshes cannot run System 7 well), Windows programmers can blithely assume version 3.1 and just post a little dialog and quit the program if this version is not present.

Of course, Windows suffers greatly from least common denominator thinking in the choice of memory models and processors—the requirement that it work on 8086's made memory management very awkward (more on this in section 3). Also, Windows inherits from the chaos in the PC clone arena—there are countless video drivers and printing models, and no reasonable sound capabilities on most PC's. While Windows tries to insulate the program from these differences, they do show through, and some drivers do not support some operations[7]. Moreover, with the introduction of Windows NT, Windows could acquire the same dichotomy that currently exists between System 6 and 7 on the Macintosh.

[6]This may be an opportunity for a developer tool for testing applications—a random Macintosh system generator.

[7]Windows provides calls to let programs determine which drawing primitives are allowed on which devices; this is analogous to a fine-grain `gestalt`.

Summary

Although Windows and the Macintosh have procedural designs, both are structured partially object-oriented—Windows' structure is better—and both have object-oriented libraries available. The major philosophical difference between the two systems is that the Macintosh provides building blocks which make it possible to create almost anything, while Windows' facilities are easy to use for common tasks but make some specialized tasks needlessly difficult.

Both systems have sizable histories, originating around 1983. The Macintosh has more of a compatibility burden because old versions continue to be in use, while Windows has only one current version, 3.1; 3.0 is being phased out rapidly.

3. Technical Amenities

While the last section described general differences, this section delves into more detail on a few aspects of programming for the Macintosh or Windows. Differences in graphics, hardware, and memory models will be examined.

Graphics

One of the Macintosh's main strengths is its tight integration of graphics to virtually all applications. Almost all applications use QuickDraw in some way for user interface. Unfortunately, this tight integration, combined with Apple's original objective of having only one version of the Macintosh, made upgrading Macintosh graphics calls difficult. As a result, even today color is clearly an add-on in the Macintosh programming interfaces. Old black-and-white applications continue to work on today's color machines, but any application that wants to use color must test for the existence of several managers (such as whether Color QuickDraw or 32-bit QuickDraw are present) and do different things accordingly. This is a burden and discourages use of color in applications where color drawing is not a primary function.

When one becomes familiar enough with it to cross the potential barrier, Color QuickDraw can be a powerful tool —it is fast, has a good color model, and allows customization of its color matching scheme. On the other hand, it is difficult to learn and many operations are error-prone. Take a look at TechNote 120 to see how much code is necessary to draw into an off-screen bitmap (the situation got better with the introduction of GWorlds in 32-bit QuickDraw). Also, try to figure out which addresses are 24-bit and which ones are 32-bit; even the video card manufacturers often get that one wrong! Finally, the Macintosh drawing architecture does not fare well at resolutions substantially different from 72 dpi—the screen is assumed to have a resolution close to 72dpi, and printing higher-resolution graphics requires working around QuickDraw.

The Windows drawing model is more tightly integrated and easier to use. There is no dichotomy between black-and-white and color calls, and graphics are resolution (and even aspect ratio)-independent. Using off-screen bitmaps for tasks like saving parts of the screen is easier than on the Macintosh, and, as long as one remembers the rules for deallocating objects, figuring out the proper sequence of calls to perform even complex operations is easy.

Windows had to have a flexible drawing model due to the variety of display and printing methods available for the PC compatibles. On the downside, there are several kinds of pixelmaps that Windows has to support. Windows also provides device-independent pixelmaps, but, at least on the systems I've tried, drawing them is very slow, even on a fast 80386 machine! Printing is also problematic, because printers may not support the primitives that one would like to use to draw the printed image.

Hardware

Ease of programming for the Macintosh and Windows environments is affected by the hardware on which they run. Part of the effect is through diversity—in general, the more diverse an environment is, the harder it is to write applications for it. The Macintosh has an advantage here, in that Apple has control over all of the hardware production, and, as a result, all Macintoshes have SCSI, serial ports, fairly good sound output capabilities (except for bugs in some sound drivers) and sound input on new Macs, screens with square pixels, standardized video interfaces and pixmap formats, and standardized methods of writing drivers (unless, of course, you would like to write a driver that also works with A/UX). The operating system is behind in a few areas—few applications are able to take advantage of multiple video pages on most video cards, and writing graphics accelerators that work is more of an art than a science.

Unless the operating system is deficient or the operations are time-critical, applications rarely have to interact with hardware directly, except for one obvious area: the processor itself. Macintoshes support the 68000, 68020, 68030, and 68040, while PCs use the 8086, 80286, 80386, and 80486. The differences among these processors are noticeable in the Motorola and great in the Intel family. Macintosh programs tend to have trouble with the caches on the newer 680x0's when using self-modifying code[8]. PC programs are more upwards-compatible[9] (except for those that deal with the memory management units, which keep on changing; few programs use the MMUs, though). On the other hand, the 8086 and 80286 processors have only 16-bit registers, so there is a substantial performance and complexity penalty for writing programs compatible with those two processors[10].

For programmers working with assembly language, the processor architectures themselves are significant. Computer architects, including the chief Intel designers, unanimously agree that the 80386 architecture is a bad one—complex and haphazard instruction encoding, small and unorthogonal register file, lots of bizarre modes[11], etc. The 68020 also has a few too many addressing modes and some unnecessary instructions, but it has more registers and is much more orthogonal. It's clear that the Intel processors pay a performance price for their architecture—20 to 200% slower than equivalent RISCs[12], depending on whom you ask, but the bigger concern is probably the headaches that programming 80x86's causes. If it were simple, why are almost all Windows programs still 16-bit? What has

[8]Microsoft programs are some of the biggest offenders here.

[9]Intel had to add logic to support self-modifying code on the newer processors to avoid breaking too many existing programs.

[10]The 68000 is also a 16-bit processor but supports almost all 32-bit instructions, so, except for a few poor design decisions such as the 32K TextEdit limit, the Macintosh never had the 16-bit problems associated with the PCs.

[11]Here is an example of what can happen: Windows was occasionally failing inside a drawing routine for no apparent reason. After a day of searching for bugs in my code, I disassembled enough of the operating system code to discover that it was overwriting its own stack with a string operation. It turns out that a while earlier my program was performing decrementing string operations. Windows got confused when the string operation direction bit in the status register was set (however, there is nothing I could find in the documentation that would imply that that bit has to be cleared when calling operating system routines). Using a status register bit to change the behavior of instructions in this way is an example of a silly 80x86 design decision. Several more of these bits will become apparent when the 80386 is used in 32-bit mode.

[12]Intel claims that this figure is 20%. If this is true, then RISCs may never become popular in personal computers, except for whichever ones Apple adopts; PC customers will prefer compatibility to a 20% speed improvement.

been happening in the seven years since the 80386 was introduced?

Segments

Anyone who has programmed Windows applications in assembly language or even a high-level language such as C or C++ is familiar with segments and their high-level language offspring: near, far, and several other brands of pointers[13]. Manipulating these pointers correctly is a time-consuming job, and doing anything which requires data structures over 64K requires care and ingenuity. System calls expect their particular brands of pointers and will fail if passed the wrong kinds of pointers or if a data structure unexpectedly straddles a 64K boundary; many calls will only work on structures smaller than 64K. Even if one uses far pointers throughout, one can get into trouble when using data in large arrays that may have elements that straddle 64K boundaries. 64K segments and near and far pointers make programmers' lives miserable (except for those who have become experts in writing fast code despite these obstacles and can thus command higher salaries), and I am glad that they will be phased out in future versions of Windows.

There is *one* positive aspect of using segments that I should mention here: they simplify testing and debugging programs. Any reads or writes off either edge of a global block of memory under Windows generate processor faults, which make indexing errors and wild references very easy to spot. The program's data structures are protected from each other on a fine scale. I found many of my bugs this way; no Macintosh equivalent is available[14]. Nevertheless, this advantage will disappear when segments are eliminated in Windows NT—although the 80386 will always have segments for compatibility, all of a program's data will be put into one large (up to 4 gigabytes) segment.

Unfortunately, this protection also causes unexpected trouble: doing a mere `p++` when `p` points to the last element of an array can cause a crash[15]. You don't even have to dereference the pointer. Really! Even some Windows system calls crash due to this bug.

[13] A near pointer is a 16-bit offset to the program's current data segment (the program also has code and stack segments, which may or may not be equivalent depending on which of the many memory models you are using). A far pointer is a 16-bit segment number and a 16-bit offset within that segment; under Windows 3.1, only 16-bit segments can be used easily (there is a library to overcome this limitation on 80386s, but using it requires writing glue routines for all system calls and re-engineering the appliation program), so segments are limited to 64K.

[14] A few Macintosh programs check the consistency of the heap, but they won't catch out-of-bounds reads or writes of one block that fall into another.

[15] The crash is a clear violation of ANSI C; one often increments a pointer past all of the elements of an array, and it is always legal to do so.

Memory Models

In addition to segmenting, other aspects of memory management affect Macintosh and Windows programmability. The Macintosh memory management grew out of the 128K Macintosh and is weak in virtual memory and sharing memory among applications. Virtual memory uses a number of clever tricks (hacks) to work but is not perfect. It uses half-solutions such as keeping the entire system heap in memory to avoid dealing with individual memory residency problems. In some cases, especially when dealing with hardware interrupt handlers and device drivers, determining which blocks of memory to lock down becomes difficult.

Macintosh applications are notoriously incestuous —they walk through many data structures in the shared system heap and globals, and sometimes follow pointers into each others' heaps. INITs do even more bizarre things. All of this makes providing real memory protection while retaining compatibility difficult, which will continue to make the Macintosh liable to crashes.

Another remnant of the Macintosh's origins is the Finder size specification for applications. Applications only get the amount of memory that the user allocates to them and must use Multifinder memory for other purposes. This dichotomy of memory usages complicates writing programs; in most cases programmers don't want to be bothered with details like these[16].

Windows has a more modern memory model, with applications sharing a single global heap (but still having separate local heaps). This dichotomy works well in practice, except when the local heap overflows (it's limited to less than 64K); also, one has to be very careful not to mix handles to blocks in the two heaps.

There was one terrible aspect of programming for Windows, which, fortunately, was eliminated in Windows 3.1. Windows was often forced to relocate code by changing every pointer to that code. This meant that in unexpected places Windows was walking through the stack of an application and changing return addresses (the stack frames always had to be in one of two uniform formats, depending on whether a near or far procedure was called). Furthermore, Windows would move the

[16] There are a few exceptions, where specifying memory usage limits for applications is useful. For example, unless restricted somehow, systems like Lisp will attempt to grab every byte of available memory, displacing other applications.

program's data segment at unexpected times; thus, no far pointers were allowed to the program's local or global variables or data in the local heap! Merely calling `foo(&c)`, where `c` is a local variable and `foo` expects a far pointer, could cause a crash. If one really did want to call `foo` on far pointers, one had to write two versions of `foo`; if `foo` was a Windows system call, one sometimes had to allocate a block in the global heap, copy `c` there, and then call `foo`. Moreover, since the program's data segment did not move very often, these bugs were hard to find.

All of these complications only occurred in Windows' real mode, which was required for the 8086 and optional for the 80286 and above. Since Windows is able to relocate segments without changing pointers to them on the 80286 and above (this is the essence of protected mode), none of the above problems occur. Real mode was dropped in Windows 3.1, probably because of the headaches it was causing for programmers.

Summary

Both the Macintosh and Windows environments suffer for historical reasons. The Macintosh is burdened by its graphics and memory models, which were adequate when the Macintosh was introduced but now must be extended in tricky ways to provide new functionality while retaining compatibility. The Windows environment is burdened by the Intel 80x86 architecture, which shows through even into high-level languages like C or C++. Segmenting, the way it is implemented on a 80x86[17], is a major cause of software frustration, but it does have one redeeming feature of simplifying debugging and testing.

The above list by no means exhausts the important areas where the Macintosh and Windows environments differ. There are many others, such as file management, networking, document interchange, and help systems. Graphics, processor architecture, and memory management are just the most pervasive ones.

4. Development

So far we have concentrated on the contents of the programs themselves. Next we will take a quick look at the usability of the development tools for writing programs. The issues are setting up the development systems, referencing documentation, using languages on the two environments, solving programming problems, and debugging and testing the programs.

Setup

When it comes to setting up a usable development environment, the Macintosh was a clear winner, at least from my experiences. It's fairly easy to set up a Macintosh computer and the software for it straight out of the box; the only trouble areas are potential compatibility problems with newer models of the Macintosh such as the Quadras (gurus who use forty-six INITs may also run into compatibility problems).

It took me over two weeks just to install Windows 3.0 and get a development system running on a PC clone, and I am not a novice. All sorts of things went wrong; the simplest were extraneous commands in the AUTOEXEC.BAT and CONFIG.SYS files. After installing it, I found that Windows would hang when it tried to run a DOS program. What was causing this? Obviously, it was the video driver, right? I had followed the Windows installation instructions for my main video card exactly according to the manufacturer's specifications, but I found that I could get Windows to work if I installed the driver another way. The manufacturer's technical support just referred me to Microsoft when asked about this.

The most complicated problem was using a second video card (Microsoft's CodeView required one under Windows 3.0). The video card worked under other applications but not Microsoft's CodeView. After several days of frustration and calls to technical support lines (which yielded nothing), I discovered that changing some of the DIP switches on the video card suddenly made CodeView work. None of this was documented anywhere.

Not everyone will be able to set up a Macintosh this smoothly or have as much trouble with PC compatibles. Nevertheless, the concept of error checking during installation on PC compatibles is much less advanced than on the Macintosh. You can never be sure that all of the switches, options, and configuration files are set up right, even after you have been using the system for a while. The Macintosh has considerably fewer options and settings, and, in general, when they are set incorrectly, the system will fail in more obvious ways.

[17] There are other ways to implement segmenting that yield excellent results and make programming even easier than on linear address-space processors.

Documentation

Form

Both the Macintosh and Windows have voluminous documentation. The primary Macintosh documentation is Inside Macintosh, volumes I-VI, and the Technical Notes, while the primary Windows documentation is Microsoft's Software Development Toolkit. The Macintosh documentation is written in book form. Although on-line Inside Macintosh is available, it is still mainly a computer-readable book; I found the HyperCard implementation of Inside Macintosh to be awkward and slow and lack the formatting of the printed form. Moreover, since Inside Macintosh volumes IV, V, and VI and the Technical Notes are delta documents, one has to check in a number of places to determine whether the description that one read in Inside Macintosh I-III is still valid. This deficiency can be remedied only partially by publishing new books as long as developers want to develop for pre-System 7 Macintoshes.

On the other hand, Windows documentation is primarily in the form of a hypertext help file/application that has good formatting and links between concepts (Microsoft had the Word engine available to them to do this); two of the reference books are just linearized printouts of the hypertext (instead of the hypertext being created from the books). This format makes browsing through links a pleasure. As discussed in section 2, Windows does not currently have a significant problem of multiple versions.

Content

Whereas the Windows on-line documentation is fun to use, I found that it is poorly organized. Graphics primitives are spread throughout several contexts (groupings), some of which are nonintuitive—why are FillRect and FrameRect in the group of Painting Functions in the Window Manager Interface Functions, while FillRgn and FrameRgn are in the group of Region Functions in the Graphics Device Interface Functions (The Rectangle Functions in the Graphics Device Interface Functions don't include FillRect and FrameRect)?

The organization of Windows on-line documentation makes it hard to search for specific functionality in some cases (in most cases it's pretty simple). Whereas on the Macintosh I can find the appropriate chapter of Inside Macintosh and read the overview or index, this is harder to do with Windows on-line documentation. It took me five months to discover that it's possible for a program to obtain its own pathname, which I did entirely by accident; if there were overviews of the various managers, I would have found the right method right away[18].

The Windows documentation is also less complete; it omits discussion of issues such as what happens if a null handle is disposed[19]; it's not clear whether this is legal or not, and sample programs give conflicting evidence. Inside Macintosh is more complete in this respect.

Languages

There are plenty of good programming language systems available for the Macintosh and Windows. The main systems for the Macintosh are Apple's MPW and the Think languages, while the main ones for Windows are the Microsoft and Borland languages. Since I was writing applications in C++, I used MPW and Borland C++.

Both MPW and Borland are powerful programming environments and provide good compilers and assemblers. The performance of both systems and their compiled code is adequate. Although Borland C++ is easier to use for beginners due to its integrated environment (like the Think languages on the Macintosh), it is much more complex than MPW when used by advanced programmers. There are numerous options, and it's quite easy to compile and link a program with inconsistent ones without getting any errors or warnings. I had to play with the options for a while before I got my compiled Windows program to run.

The documentation styles differ for the two systems. MPW documentation tends to be geared towards advanced programmers (although that has been improving lately), while Borland documentation is a mix of tutorials and reference manuals. The MPW reference manuals are very detailed and precise; every feature I wanted to know anything about was documented. On the other hand, while Borland's tutorials are well-done, the reference manuals leave a lot to be desired. I found it hard to find features I was looking for, and many were not documented. Examples include Borland's C/C++ extensions like near and far classes, some assembler index expressions (addition is not commutative here; I had to discover some of the

[18] There are some books which help with providing overviews of various managers, such as the tutorial in Microsoft's Software Development Toolkit and [4], but they tend to shy away from more advanced topics.

[19] Most calls work on null handles (I looked at the code). I avoid calling them on null handles to avoid future compatibility problems and also because Windows' Discipline complains about using null handles; however, most examples in books and documentation will make calls on null handles under some error conditions.

rules by trial and error), and structure operations (how do I get the offset to an element within a structure?).

Both systems provide adequate additional programming tools such as resource editors, compilers, and profilers.

Programming

When programming a system for the first time, one usually relies on examples and reading other people's code. Examples of simple functionality are plentiful on both Windows and the Macintosh. Examples of more advanced applications are harder to get; MacApp and the other object libraries are some of the best sources.

Another valuable source of examples is the system code itself. One often finds that one has to implement something that is already done in the system, but maybe a bit differently or maybe the system routine cannot be called directly. The best way to learn how to accomplish such a task is to step through or disassemble the relevant system routine. This ensures that one doesn't miss some important aspects of compatibility with future or localized systems. Also, looking at things like a window definition procedure may be essential to making sure that one's custom windows look the same way as the system ones on a variety of screen resolutions and other configurations. Finally, disassembling code is sometimes necessary to discover what is happening when the system is not behaving as expected[20].

Walking through and disassembling system code is simple on the Macintosh due to the presence of several powerful assembly-level debuggers. Also, Macintosh system calls are all routed through the A000 trap bottleneck, which makes it easy to see every call the system is making to itself while performing the task being examined. On the other hand, stepping through and disassembling Windows is much harder, and there is no easy way to identify system entry and exit points.

Debugging and Testing

The application debugging tools on both the Macintosh and Windows are adequate, if a little cumbersome to use, at both the source and assembly level. Both environments provide various tools for stressing programs such as allocating most of memory under Windows or doing a heap scramble on the Macintosh. There are also visualization tools such as Windows message browsers or A000 trap recorders on the Macintosh.

Both environments provide utilities to check whether programs are well-behaved. The Macintosh has a Discipline, which is often useful but is sometimes over-eager or over-lenient. Windows 3.1 has a built-in discipline which will report suspect system calls by sending messages out of the computer's serial port or onto another screen. This is a very useful facility for catching common programming errors such as disposing graphics structures in the wrong order or not disposing them at all (Windows' discipline will report these when the program terminates). Building this discipline into the system was a wise investment by Microsoft—it will reduce the number of crashes encountered by users and simplify Microsoft's job in evolving the system in the future.

5. Conclusion

The Macintosh and Windows are sophisticated graphical operating systems for personal computers. Both are rapidly evolving and have large markets for third-party software. They provide many of the similar features, but differ in the following areas:

• The Macintosh operating system emphasizes providing flexible services for a wide variety of tasks. Many specialized applications are possible, but simple tasks may require a large initial investment of work. On the other hand, Windows emphasizes common tasks and lacks some functionality for specialized ones. Documentation and development environments mirror this difference. I do not exactly understand why this distinction is so pervasive, but it seems to define the fundamental "attitudes" of the two environments.

• Both systems are procedural, but Windows has a more modern, object-oriented structure for passing messages and events.

• Apple has more control over the hardware substrate, resulting in more uniform system configurations and features. On the other hand, Apple has more versions of the operating system, while there is only one target version of Windows. Thus, programmers for both environments are likely to face the problems of programming for multiple configurations.

• Programming the 68000-family chips is much easier than programming the 8086 family. The 8086-family chips provide numerous hard-to-use modes, and, probably not coincidentally, most Windows programs are still 16-bit.

• System setup is much harder on PC compatibles, and Windows does not solve this problem; in fact, it

[20] MacDTS old-timers will testify to that; I've sent numerous bug reports showing the exact location of the error.

introduces numerous .INI files and other potential areas for users to make mistakes.

These are the main technical distinctions that I found in my experiences; yours will differ slightly. Of course, technical considerations may be entirely secondary in choosing the system for which one should develop programs. Windows' installed base is growing much more rapidly than the Macintosh and provides a larger market for applications. Also, even if someone were to definitively establish that, say, developing for Windows is harder than for the Macintosh (or *vice versa*), would this mean that programmers would prefer to work on Windows or the Mac? Some might want to write more ambitious programs; others might want a greater challenge.

The operating system scene is changing rapidly, and we will see many new developments over the next few years. There will be new operating systems and new processors, providing plenty of opportunities for innovation. I hope that the diversity of versions in the industry will be brought under control and that new systems will be designed wisely, paying attention to the principles of abstraction; otherwise, the operating systems will become even more difficult to maintain and entropy will rise to unacceptable levels.

A. Object-Oriented Programming and Abstraction

This appendix defines object orientation and abstraction in the context of programming languages and discusses their interaction. They are related, often confused, concepts. Object orientation and abstraction are at odds in one area; it is important to be aware of their limitations.

An object-oriented language is one that supports:

1. Defining data types (*classes*). Each instance of a class is called an *object* (or an *instance*) and has associated *methods* and *instance variables*.

2. Defining functions (methods) that operate on those objects.

3. Defining classes by *inheriting* from existing classes and overriding some methods.

Many experts claim that using object-oriented programming offers significant time savings in programming, and, in most cases, this is correct. However, object-oriented programming has a dark side to it, and, furthermore, much of the gain attributed to object-oriented programming may simply be due to the use of abstraction.

Abstraction is one of the key ideas in programming (and science in general). In programming, it means restricting the use of data structures to calling functions that operate on them and accessing public instance variables; all other operations such as calling private functions on a data structure or accessing its other components directly are forbidden. This has the following benefits:

- The user of an abstract data structure does not have to know how it's implemented. He need only examine the interface to use it.

- The implementor of an abstract data structure can change its implementation, and, as long as he keeps the same interface, he will not disrupt the users.

We use this concept all the time. A car is an abstraction, with the functioning of the steering wheel and pedals being the interface—one does not need to know how a car works to use it, and the car manufacturer can change the car's engine design without having to re-educate the users.

Features that aid in defining abstractions are often built into the newer programming languages, but one can define abstractions even in older languages such as C or Pascal by following conventions. Keep in mind, though, that merely using a good language will not guarantee that one's program has good abstractions—it's easy to take short cuts and make all instance variables public or define inappropriate interfaces. Defining abstractions is an art to be learned and practiced.

What does abstraction have to do with object-oriented programming? Well, characteristics 1 and 2 of object-oriented programming are precisely what is needed to define abstractions. On the other hand, characteristic 3 is controversial and can sometimes undo all of the benefits gained from 1 and 2. The problem arises when one begins to override methods when defining subclasses. If the overridden method replaces the original method even when called by other methods from the superclass (in C++ terminology it is a `virtual` method), then other methods in the superclass can suddenly change meaning. For instance, if class `ArrowDrawingA` overrides the `drawLine` method so that it draws lines with arrows, then `drawRectangle` may or may not suddenly start drawing rectangles with arrows, depending on whether it calls `drawLine` or some other internal method to draw its lines. Which one happens *cannot be deduced* from the interface to the `DrawingA` class; thus, by using virtual overriding, one breaks the abstraction barrier and *relies on the class's implementation* instead of just the interface. This is why most object libraries such as MacApp are not useful without the source code.

Shallower, non-`virtual` overriding such as that in `DrawingB` is always safe. `Virtual` overriding relies on information not present in class specifications[21] and should only be done with caution—don't declare every method virtual by default as some books recommend! `Virtual` methods have some good uses such as defining callbacks, but in those cases the exact circumstances under which `virtual` methods are called and not called should be documented along with each class declaration.

```
class DrawingA
   {
   virtual void drawLine(Point start,
                                     Point end);
   void drawRectangle(Point corner1,
                                 Point corner2);
   ...
   }

class ArrowDrawingA:DrawingA
   {
   void drawLine(Point start, Point end);
   ...
   }


class DrawingB
   {
   void drawLine(Point start, Point end);
   void drawRectangle(Point corner1,
                                 Point corner2);
   ...
   }

class ArrowDrawingB:DrawingB
   {
   void drawLine(Point start, Point end);
   ...
   }
```

21Some time ago this problem became apparent to me in another manner—I was looking for a strong typing system for an object-oriented language that did not require type-casting. It turns out that even *designing* such a language is exceedingly difficult (much less writing a compiler for it), and almost all of the difficulties stem from `virtual` overriding. The same difficulties that break abstractions also make programming languages difficult to design and understand, for computers as well as humans.

Bibliography

[1] Apple Computer, Inc. *Inside Macintosh*, Volumes I–VI. Addison-Wesley, 1985-1991.

[2] Apple Computer, Inc. *Macintosh Technical Note 304: Pending Update Perils*. August 1991.

[3] Microsoft, Inc. *Microsoft Windows Software Development Kit*, 1991.

[4] Charles Petzold. *Programming Windows*, Second Edition. Microsoft Press, 1990.