The Cost of User-Friendly Programming: MacImage as Example

Ferren MacIntyre
Kenneth W. Estep[1]
John McN. Sieburth
Graduate School of Oceanography,
University of Rhode Island, Narragansett, RI 02881-1197

Abstract

The Xerox Star and Apple Macintosh computers introduced the concept of the "user interface" to personal-computer software. The user interface is an intuitive, easy-to-use environment through which the user directs operation of the computer. It simplifies program learning and use, but at the cost of increased code and a limitation of programming options. As an example of this cost, we describe MacImage for the Macintosh, an image-analysis application. The actual code needed to implement a basic Macintosh interface is given for a simpler application called Simple Tones. The user interface occupies 47% and 60%, respectively, of the Forth code for MacImage and Simple Tones, but results in software that can be used almost immediately, without consulting a manual.

A second, less exploited, but ultimately more important aspect of user friendliness is that both the programmer and user should recognize that the user will need to optimize code for his application, and the program must be designed with this eventuality in mind.

The motivation for the work described here was user dissatisfaction with the software accompanying a commercial product, the Artek 810V™ Image Analysis System, shown in Fig.1. While the $35,000 hardware has much to recommend it, the software, as delivered, was slow, incomplete, and user hostile, in the sense that operations by the user, other than those anticipated by the programmer, resulted in program abort with loss of data. Since the logic of the menu structure never became apparent to the user, such procedural errors were annoyingly frequent.

In cooperation with the manufacturer, we undertook to produce software written from the user's point of view. We had a free hand to refine the software until all annoyances were removed, but there were over-riding constraints on our work, the most important being that we were limited to the data generated by the Artek firmware, shown in Fig. 2A, (although we anticipated at that time that Artek would change their proprietary code to produce the data shown in Fig. 2B, as discussed below).

The range of images that can be analyzed by a given machine is probably greater than any single program can encompass. MacImage is a silhouette analyzer, providing shape and size information for objects of sufficient contrast to be discriminated from the background, provided only that there are not more than 3 perforations in a silhouette which produce gaps in a given raster line (this is a limitation in the Artek 810 firmware). We use MacImage to analyze objects

[1] Present address: Havforskningsinstituttet, Nordnesparken 2, N-5024 Bergen, Norway

ranging from the microscopic latex calibration spheres of Fig. 7, to the complex 3-mm fish larvae of Fig. 8 (Estep, et al. 1986).

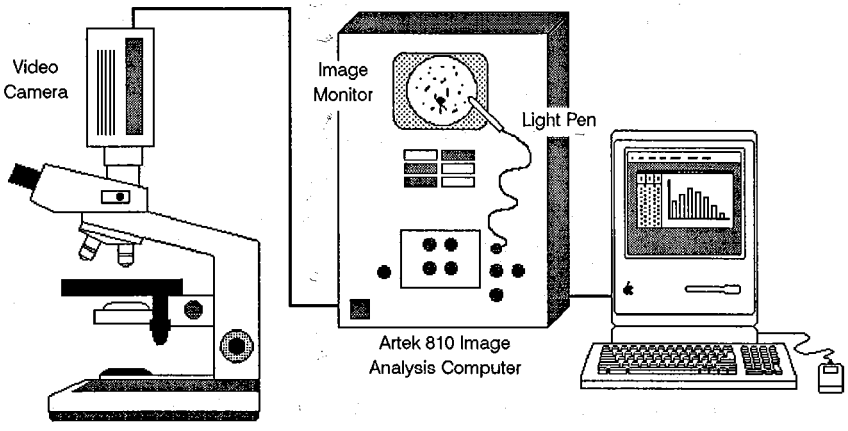

Figure.1 The Artek system with a Macintosh front end. The image monitor displays a binary, digitized image derived from a standard television signal from the video camera. Hardware controls govern such parameters as aperture size and thresholding level on the Artek. Communication consists of control signals to the Artek and raw data to the Macintosh, over an RS-232 link.

Approaching image analysis, we expected it to be highly mathematical. To our surprise, the most striking feature of the MacImage code is shown in Fig. 3, which analyzes the distribution of Forth code blocks by function. Arithmetic blocks, which actually perform numerical operations upon the data, represent 9% of the code, with the rest devoted to overhead, I/O, and interface. It might be thought that the Artek was doing arithmetic for us, thus making our user-interface-code percentage unrealistically high. However, for our second image-analysis system (Forth and Macintosh based), in which we do the arithmetic ourselves, mostly in a slaved IBM clone, we found the ratio of image-processing-code to user-interface-code (by either lines or memory locations) to have changed only from the 1:11 found for MacImage to 1:8, thus supporting the folk belief that 10% of the code does 90% of the work.

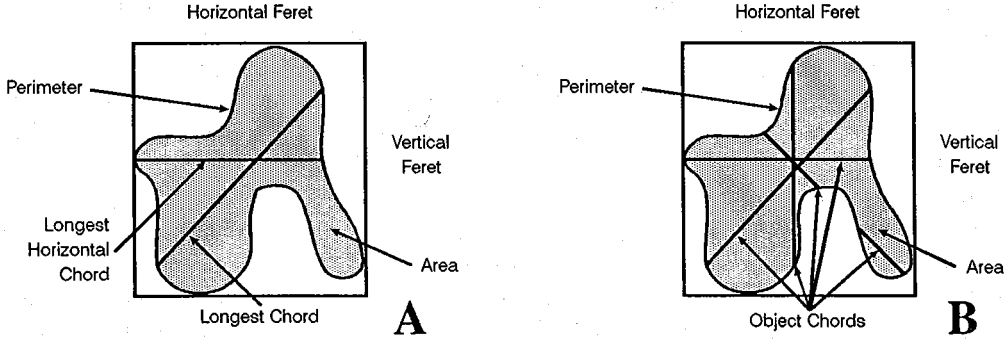

Figure 2. Arbitrary silhouette representing typical object of image analysis. Basic measurements from the Artek 810 image analysis computer (A) as they are currently calculated and (B) as suggested by the authors. "Feret" (with a silent "t") refers to the diameter measured between parallel tangents, as first used by the French cement chemist, L.R. Feret (1931).

Functional Distribution of MacImage Code

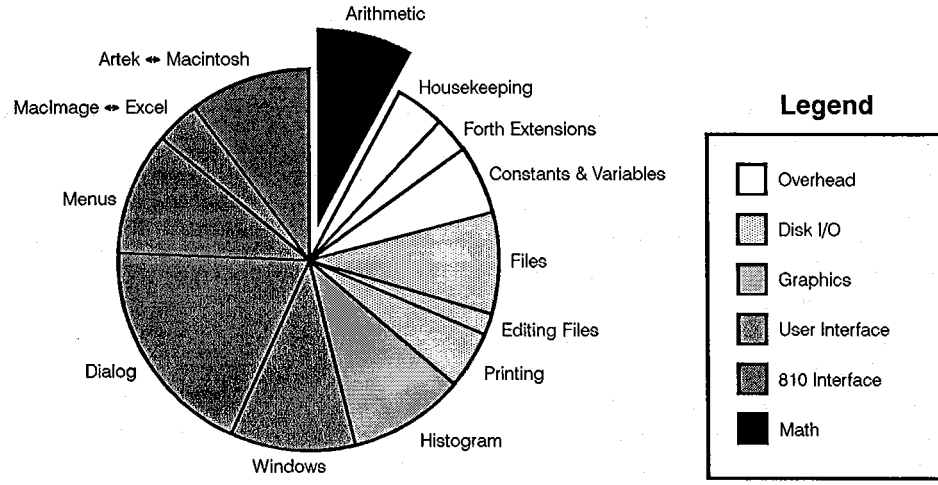

Figure 3. Functional distribution of the MacImage code. Note the predominance of the code necessary for the user interface, and the small amount of arithmetic code.

Almost all operations are in 32-bit integer arithmetic, but we still found it necessary to load 25 blocks of software floating-point code to calculate one crucial parameter, the number of objects per unit volume of sample. This number has three components, each of which is comfortably within the range of scaled integers, but which together are not. It is composed of the number of objects per microscope field (0.1 to10,000), times the number of fields per sample (1 to10,000), divided by the sample volume (0.01 to10,000), with a combined dynamic range of 10^{15}. When we considered devising an automatic scaling algorithm to cope with the possible inputs, we found that we were re-inventing floating point. (So ingrained seems resistance to floating point in parts of the Forth community that we were repeatedly told that we should have been able to scale the numbers. Still, none of the advocates of scaling has published a demonstration that this can be done for numbers which may vary by many orders of magnitude, or shown how the results would differ from floating point. In our hands, at least, scaling works when, and only when, the range is a priori knowledge. With the spread of hardware floating-point processors, we anticipate that even Forth users will come to accept what scientists and pocket-calculator users have long understood: floating point is useful.)

The small proportion of "Forth extensions" is a tribute to the completeness of Creative Solutions' (Rockville, MD) MacForth Level III™. The added words are mostly analogs of words from MMSFORTH™ (Miller, 1979-88). Words such as **$>NUMBER**, to convert strings to numbers, appeared in a MacForth update while we were writing code, and proved invaluable in dealing with communications between our component machines.

The components of user friendliness

The largest category of programming code, 47%, is the one we call the "user interface", devoted to making the program work properly in the Macintosh environment. Such user friendliness as we have achieved results from the interaction of several components, chief of which appear those shown in Table 1:

Category	*Examples*
Good programming	"Bombproofing", logical design
Macintosh features	MacDraw, Excel, user interface
Machine technology	Processor speed, bus width, memory size

Table 1. Components of user friendliness

It is difficult to apportion credit for user friendliness among these components, but certainly good programming is a *sine qua non* without which nothing useful can be accomplished. For instance, some of the speed increase seen in Fig. 4 can be attributed to the Macintosh's 68000 CPU, but more comes from attention to optimizing data transfer between the Artek and MacImage, and from calculating the right numbers at the right time and storing them efficiently. Some comes from making use of the menu structure of the Macintosh, but more from a pragmatic layout of menu items.

User friendliness has a special meaning in Macintosh software. One of the things that makes the Macintosh a unique computer is the extensive ROM routines that were developed by Apple Computer to standardize the operation of diverse software packages. All Macintosh programs include icons and standard operating procedures. Underlying these standard features are the programs in ROM, for example, there is a "Menu Manager" that creates and controls standard "pull-down" menus for program commands and control, a "Window Manager" that creates windows that operate in the standard Macintosh fashion, a "Dialog Manager" that creates and controls "dialog boxes" (a special window type), and a "File Manager" that produces standard dialog boxes for opening, closing and saving files.

Use of these ROM routines by all Macintosh software creates a set of expectations on the part of the user about how the machine will perform in response to a given action from the keyboard or mouse, and Macintosh magazines, such as *MacUser*, include a category "Follows Mac Interface" in their review sections, along with "Ease of User", "Performance", "Documen-

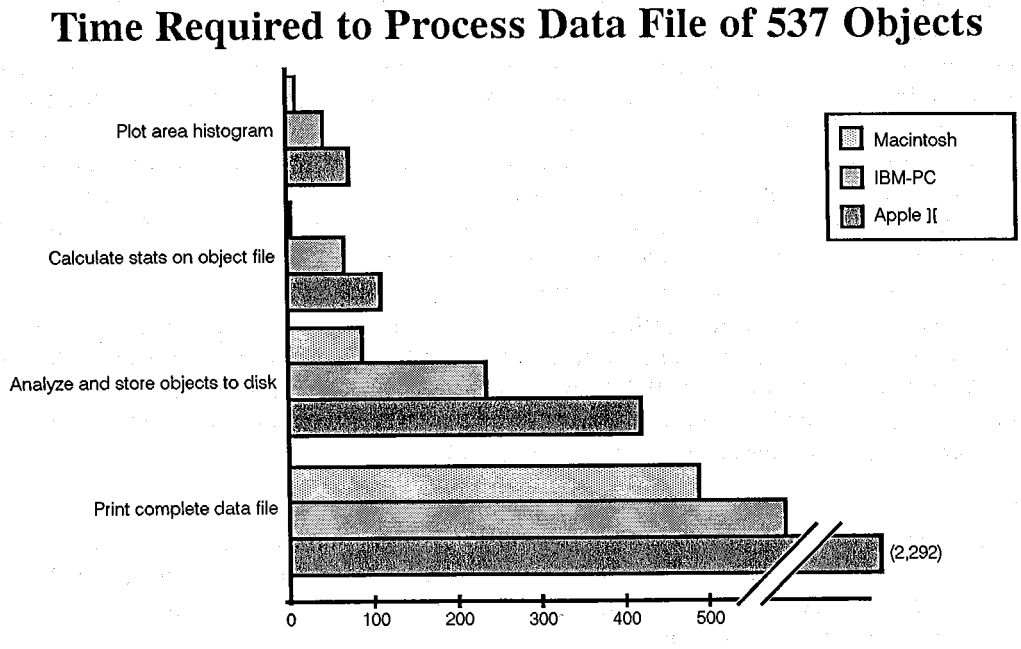

Figure 4. Program performance comparison of the MacImage program to programs for control of the Artek 810 image analyzer written for the Apple II and IBM-PC.

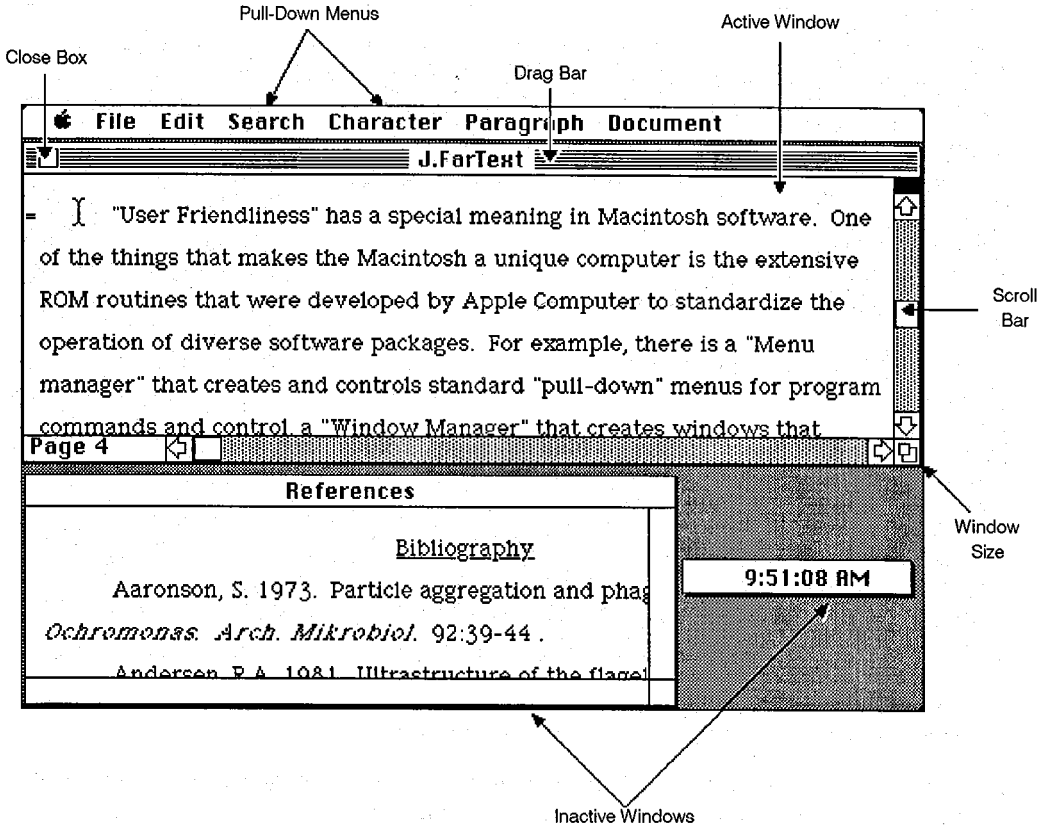

Figure 5. A screen image from the Microsoft Word word processing program showing standard Macintosh user interface components.

tation", etc. As one example of these user expectations, we have included a screen image from Microsoft Word, the word processor used to write this manuscript (Fig. 5). There are 3 windows present in this screen image, two are word-processing documents and the other is the alarm clock "desk accessory". The scroll bars, window-size box, close box and drag bar are standard features that all Macintosh programs include. Thus when one learns how to use these screen areas in one program, one expects them to work exactly the same in other programs.

For users of the Macintosh, these standardized features make work much easier, and in fact, most new software packages can be learned by trial and error. For the programmer however, the ROM user-interface routines can be a two-edged sword. Much effort must be expended in interacting with these routines, and some programmers complain about having to "do it Apple's way." The result of this effort, however, is an easily learned program, and at a cost of less code than would be required to create the same effects from scratch.

Macintosh-specific user-interface code

Because the 207 blocks of MacImage code are too long to include here, we have written a program called "Simple Tones" that illustrates the cost of a Macintosh user interface and also provides a useful model for anyone new to Macintosh programming at this level. The underlying language, MacForth K2.4, is described as Forth-79 compatible (our preferred version), with many additions to deal with the Macintosh, such as `TONE` to run the speaker, and words to generate windows, controls, and menus, exemplified by `W.BOUNDS` , `C.TYPE` , and `DRAW.MENU.BAR` respectively. Simple Tones generates a low and high tone through the speaker in the Macintosh. In

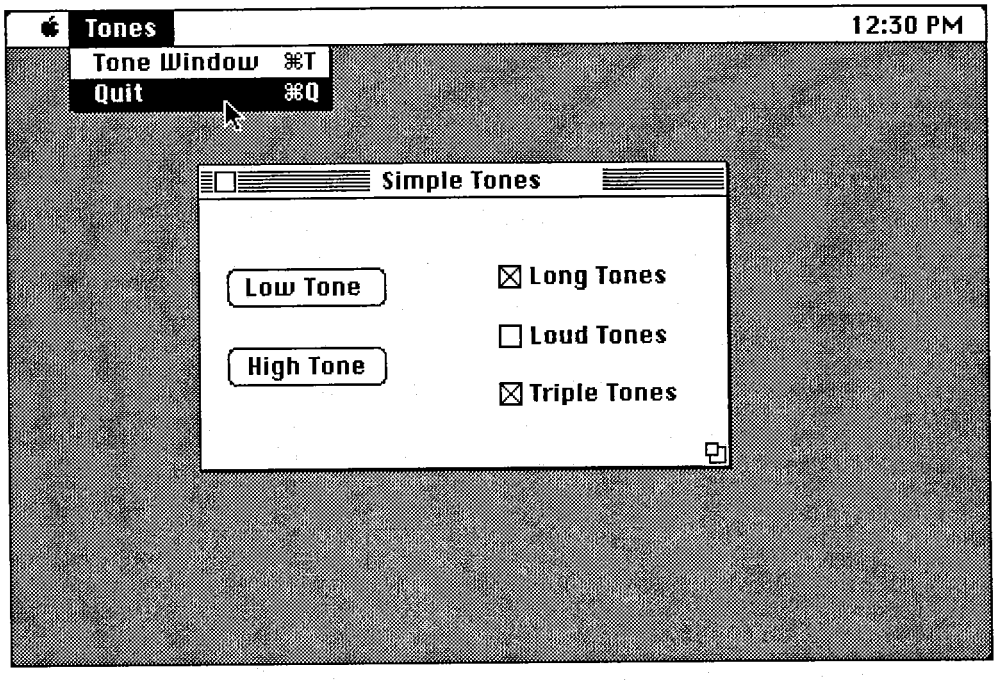

Figure 6. A screen image of the Simple Tones program, showing its single window with control push buttons and check boxes. A single menu is pulled down, with "Quit" selected.

addition, it allows control over the volume and duration of the tones generated, and also allows three tones to be generated consecutively. Blocks 1 and 2 of the program shown in Listing 1 contain the words necessary to store tone parameters and to generate the tones.

Using the words in these two blocks, tones can be generated from the Forth command window. The program could not be used, however, without consulting the source code, or an accompanying manual.

Figure 6 illustrates the appearance of the Macintosh screen after Simple Tones has been modified to include a Macintosh-type user interface. Two "push buttons" on the left side of the single window generate the tones when the mouse is used to position the cursor over them, and the mouse button is pressed. Control over the duration, volume and triple tone function is achieved by the the "check boxes" on the right side of the screen. The Simple Tones window has a close box in the upper left and a size box in the lower right, allowing the window to be resized, moved around the screen, and closed with the mouse. Finally, a menu has been added that allows the tone window to be redisplayed if it is closed, and to allow an exit from the program. Blocks 3-8 in Listing 2 control all functions related to the controls, window and menu. The **CASE** statement in Block 6 is a version of Eaker's, executing the words between **OF** and **ENDOF** for a match between the control value and the word preceeding **@** ...The "|" and unarrowed "--" symbols appearing in stack comments are not logical ORs and em dashes, but MacForth conventions for separating descriptive comments, and "before" and "after" stack pictures, respectively.

The code responsible for the user interface of Simple Tones represents more than 60% of the complete program. The result of this additional expenditure of code is a program that can be run immediately by anyone familiar with the rudiments of the Macintosh interface.

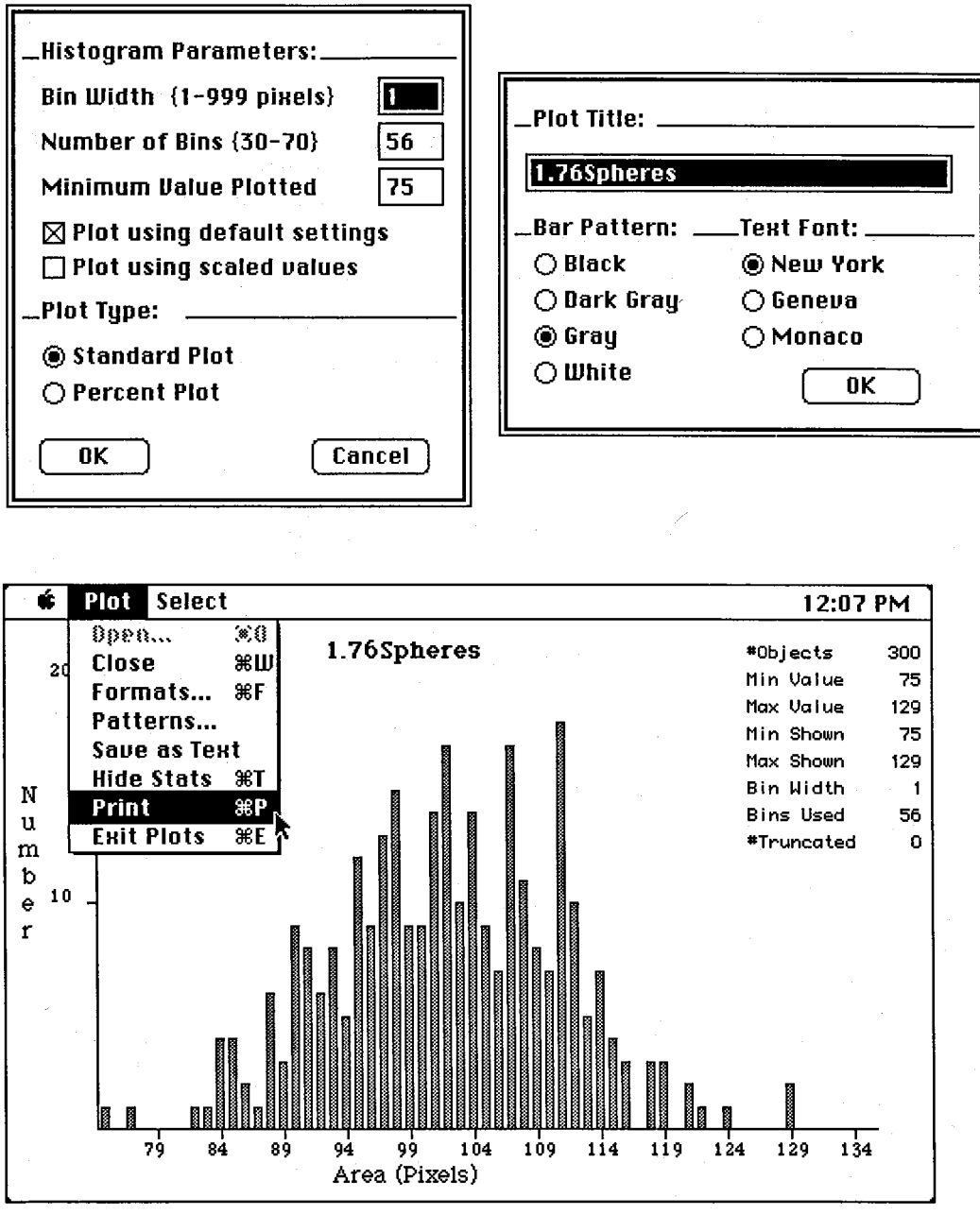

Figure 7. Components of the MacImage user interface. Two dialog boxes that control appearance of the histogram appear at the top of the figure. The bottom of the figure is a screen image of the histogram portion of the MacImage program with the "Plot" menu pulled down.

User interface in MacImage

Figure 7 displays several features from the user interface of the MacImage program. The histogram at the bottom is a calibration run on 1.76-μm latex spheres at maximum resolution. Overlying a portion of the histogram is a pull-down menu with "Print" selected, this being the only operation necessary to produce hard copy. The "Open..." option in the menu appears in gray rather than black type. This graying out of menu items, another aspect of the Macintosh

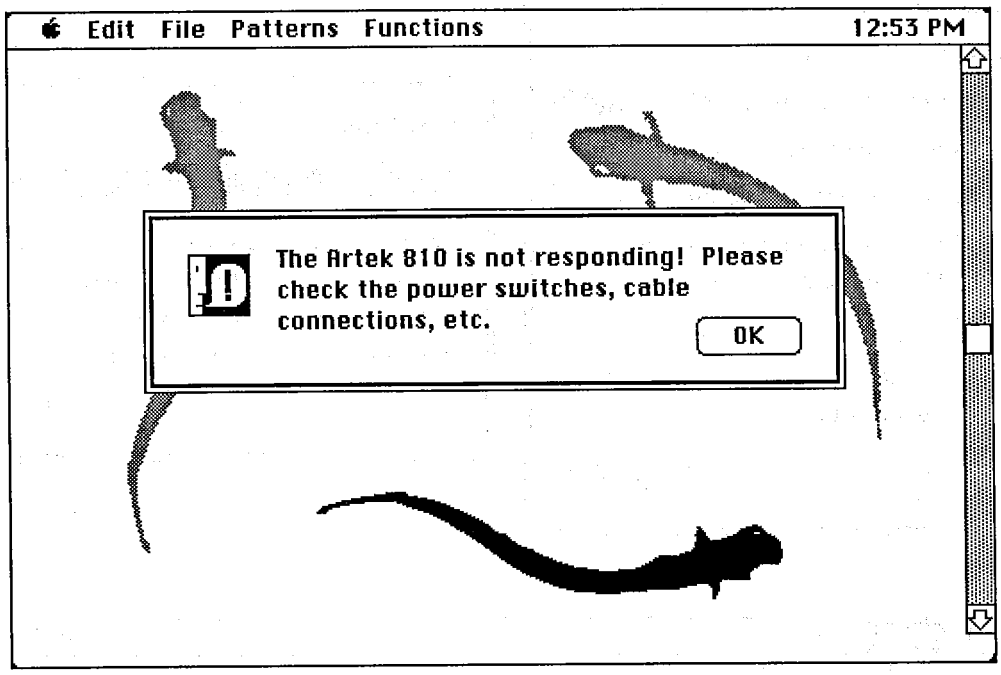

Figure 8. A screen image of the Image Transfer section of the program, illustrating another aspect of the user interface, a dialog box that informs the user that the Artek is not responding to commands from the Macintosh.

user interface, indicates program commands that cannot be selected from this point in the program. Selecting "Formats..." or "Patterns..." from the menu causes dialog boxes to be displayed on the screen. Dialog boxes are a special type of window that disables all events, such as mouse downs in the menu bar, until the dialog box has been responded to. If "Patterns..." is selected, the upper right dialog box appears, allowing one to specify title, shading, and fonts. By Macintosh convention, the black areas are active: information from the keyboard will appear in "plot title" at this point. If "Formats..." is selected, the dialog box at the upper left of the figure is displayed on the screen.

Histogram programs supplied by Artek for the Apple II and IBM PC prompted the user for "bin width" and "number of bins" before defining its terms or displaying any information about the data set, so that one was working blind. They displayed no numerical information along the axes, and had no provision for dealing with the inevitable artifacts introduced by re-binning of digitized data. Finally, when one redid an existing histogram, the old values selected for these parameters were lost. We addressed each of these problems, and consider the information and control shown to be the minimum necessary for constructive use. MacImage automatically does its best to produce an acceptable histogram; only if this attempt fails does the user intervene with new parameters. When a new histogram is attempted, the user has reference to the values from the last histogram in the "Histogram Parameters" section of the dialog box in Fig. 7. If the user strays too far from usable parameters, an "X" in the "Plot using default settings" checkbox returns the parameter boxes to the default values.

Figure 8 displays the results of an attempt to transmit stored data back to the Artek, with the Artek unconnected. In the commercial program, such an attempt resulted in an abort: here, another type of dialog window appears with the characteristic Macintosh error-message display.

As we saw in the previous section, much of the programming of the "user-interface" blocks is devoted to the mundane tasks of defining screen areas which will respond to the presence of

the cursor and a mouse click, and to storing the resulting keyboard entries. It is neither exciting nor challenging, merely time-consuming and necessary if one wants to produce a usable program. As we have shown in the Simple Tones program, and the more involved MacImage program, meaningful user interfaces can be constructed only at the cost of considerable effort. If they are a part of a coherent and standardized environment, however, they result in programs that are much easier to learn and use. The appearance of numerous Macintosh-like environments, such as the Atari 1040ST, Amiga, and the GEM desktop for the IBM-PC demonstrates the usefulness and popularity of the user interface concept.

User-modifiable Code

The second cost of user-friendliness is a willingness on the part of programmer and user to plan for modifications by the user. Except in the most routine applications, there will always be situations unanticipated by the programmer in which the user can, by making small changes, optimize the operation of a machine. This philosophy is somewhat counter to current practice, which appears to favor sealed-package programs that attempt to be all things to all users. This approach perhaps arose partly from the natural desire of vendors to profit from proprietary programming, and we have no objection to this as long as it does not preclude user improvement. A logical place for proprietary code is in highly optimized drivers and similar low-level programming. The other source of concealed programming may be the historical difficulty of writing and implementing user-added code with grace and efficiency, which made sense only so long as efficient code had to be written in assembler, and easily written code in one of those FORTRAN- or BASIC-like languages which Lewis (1979) described as having the same relation to computer language "as sign language has to English" Fortunately, such artificial constraints are obsolete, and it is increasingly possible to design for user optimization of programs.

Unfortunately for this philosophy, we were not making the decisions for Artek, who chose to keep the code we had written for them proprietary, and therefore unmodifiable by the user. For our second, state-of-the-art image-analysis system, whose marketing strategy is under our own control, we have a policy of open source code, thereby giving the user complete access to everything we know about the internals of the machine, and complete ability to modify or extend its capabilities.

A case in point is the choice of parameters in Fig. 2. Artek's set (2A) was chosen by a mathematician, who tested his algorithms on smooth convex shapes very much larger than a pixel. Yet because of digitization errors (and an apparent error in a square-root algorithm) we found perimeter untrustworthy for our measurements, and the chord selection in 2A was demonstrably less useful than 2B for our needs. But the 810 was not designed to permit such radical changes, and we were left with discarding perimeter and working our way around the inferior choice of chords.

In our research environment, we create one-of-a-kind instruments, whose operational and data-reduction code can be rewritten and tested in a matter of minutes. When it proves erroneous, or hostile, or simply clumsy, we fix it. The "secrets" lie in putting firmware in electrically-erasable programmable read-only memory (EEPROM) with provisions for immediate *in situ* re-programming, providing a SCSI port (Small-Computer Standard Interface), as on the Macintosh Plus, and—most important, in writing in a language which is efficient, flexible, and simple—a need exactly filled by Forth. Rockwell's Forth-speaking R65F11/12 is often an adequate basis on which to build such systems.

So important is this "reprogrammable" level of user friendliness that we will no longer purchase a commercial instrument unless it offers an easy way to rewrite its program for our particular needs.

As a concrete example of making the user responsible for understanding what his machine is doing, we deliberately omitted some key functions from the core MacImage program. Artek hoped that we would provide software equivalents of buttons labelled "Length", "Width", and "Volume", which would extract these functions from silhouettes of arbitrary shape. But we have watched too many graduate students present computer-processed data as factual simply because they had been processed by a standard library package, while the student had no concept of the accuracy, significance, or appropriateness of the processing. We deliberately chose not to encourage such blind faith in software, and instead offered instead an easy way to move sample data from MacImage to the spreadsheet Excel™, where the user may process them as desired.

We thus unsubtly compel user attention because there are many ways of converting from two-dimensional data to three, and all of them require assumptions. The assumptions behind the original Artek program produced demonstrably erroneous results for our objects; the assumptions behind similar operations in other image analyzers seemed equally implausible to us, so we wrote our own to work on our samples. Implementing such assumptions is a matter of a minute's coding in Excel, requiring nothing more than typing in one equation and manipulating a few of the standard Macintosh user-interface features described above. (This is aided by Excel's relatively complete set of mathematical functions.) We felt surface area to be an important parameter of marine microorganisms (since, to a first approximation, all chemical reactions occur at surfaces), so we added an ability to estimate the surface area. We have published our chosen data-reduction equations (Estep et al.), but we do not force them upon the unwary.

Attributions and Acknowledgments

Estep wrote the bulk of the MacImage program as part of his PhD thesis for and supported by Sieburth's marine microbiological laboratory under NSF grant OCE-85-11365, consulting MacIntyre as necessary on Forth and the numerical treatment of data. Artek provided a 10-Mb "Hyperdrive" hard-disk for Estep's Macintosh and a Macintosh with a Hyperdrive and ImageWriter printer for program development. Robert Goldstein, Artek's programmer, provided information on the inner workings of the Artek 810.

We acknowledge superb telephone support from Creative Solutions, and the assistance of numerous MacForth programmers on Compuserve, including Dave Sibley, Ward MacFarland, and Dave and Don Colburn, without which we might never have gotten the program working.

References

Estep, K.W., F. MacIntyre, E. Hjörleifsson and J.McN. Sieburth (1986). "MacImage: A Portable, Moderate-cost Image Analysis System for the Rapid and Accurate Mensuration of Marine Organisms." *Mar. Ecol. Prog. Ser.*

Feret, L.R. (1931). "Particle size of pulverulent materials." *New Intern. Assoc. Testing Materials,* Sept. 9 pp,

Lewis, T.G., (1979) "Some Laws of Personal Computing." *BYTE* 4:186-191.

Miller, A.R. (1979-88) *MMSFORTH Users' Manual.* (Natick, MA 01760-2099).

John Sieburth is professor of microbiology at GSO. Ferren McIntyre left his Research Professorship at GSO to found A/S Pixelwerks, Ltd. with Kenneth Estep after a grant proposal was rejected because, "the author has no experience with image analysis." To his chagrin, he finds that he is still living off of the same granting agencies, but at least other people have to write their proposals.

Listing 1. The working code for Simple Tones.

```
Block 1:
( Simple tones - sound generation words )              ( Ø51Ø86 KWE)

 16Ø  CONSTANT LOUD            \ Constant for loud tone
  3Ø  CONSTANT SOFT            \ Constant for soft tone
   3  CONSTANT SHORT           \ Constant for short tone
  1Ø  CONSTANT LONG            \ Constant for long tone
 CREATE  ~VOLUME    SOFT  ,    \ Variable for tone volume
 CREATE  ~DURATION SHORT  ,    \ Variable for tone duration
 CREATE  ~3TONES?   Ø  ,       \ Flag for triple tone generation
 \ Sound generation words
    : DELAY  1ØØØØ Ø DO LOOP ;

    : LTONE  ~DURATION @ ~VOLUME @ 3ØØØ TONE ;
    : HTONE  ~DURATION @ ~VOLUME @ 5ØØØ TONE ;

    : 3LTONE LTONE DELAY LTONE DELAY LTONE ;
    : 3HTONE HTONE DELAY HTONE DELAY HTONE ;
                                                          ->
Block 2:
( Simple tones - sound generation words, con't)        ( Ø51Ø86 KWE)

: CHANGE.VOLUME       ( - | toggle tone volume between loud and soft)
     ~VOLUME  @  LOUD = IF SOFT  ELSE LOUD THEN ~VOLUME   ! ;
: CHANGE.DURATION     ( - | toggle tone length between loud and soft)
     ~DURATION @ LONG = IF SHORT ELSE LONG THEN ~DURATION ! ;
: CHANGE.TRIPLE.TONE ( - | toggle triple tone function)
     ~3TONES? DUP @ NOT SWAP !  ;
: DO.LTONE            ( - | generate the low tone)
     ~3TONES? @ IF 3LTONE ELSE LTONE THEN ;
: DO.HTONE            ( - | generate the high tone)
     ~3TONES? @ IF 3HTONE ELSE HTONE THEN  ;
                                                          ->
```

Listing 2. The interface code for Simple Tones

```
Block 3:
( Simple tones - Window definition)                          ( Ø51Ø86 KWE)
NEW.WINDOW TONE.WINDOW
  1ØØ 1ØØ 24Ø 375        TONE.WINDOW W.BOUNDS      \ Set window size
  " Simple Tones"        TONE.WINDOW W.TITLE       \ Set window title
SIZE.BOX CLOSE.BOX +     TONE.WINDOW W.ATTRIBUTES
                              \ Add a close box and a window size box
 TONE.WINDOW  ADD.WINDOW      \ Add window pointer to window list
                                                                  ->

Block 4:
( Simple tones - Tone window push buttons)                   ( Ø51Ø86 KWE)
NEW.CONTROL LTONE.BUT  \ low tone push button control definition
    A.PUSH.BUTTON    LTONE.BUT C.TYPE          \ control type
    " Low Tone "     LTONE.BUT C.TITLE         \ control title
    14  36           LTONE.BUT C.POSITION      \ control location
    TONE.WINDOW      LTONE.BUT APPEND.CONTROL \ window pointer
NEW.CONTROL HTONE.BUT \ high tone push button control definition
    A.PUSH.BUTTON    HTONE.BUT C.TYPE
    " High Tone"     HTONE.BUT C.TITLE
    14 77            HTONE.BUT C.POSITION
    TONE.WINDOW      HTONE.BUT APPEND.CONTROL                  ->

Block 5:
( Simple Tones - Tone window check boxes)                    ( Ø51Ø86 KWE)
NEW.CONTROL  LOUD.TONE   \ Tone volume check box definition
    A.CHECK.BOX         LOUD.TONE  C.TYPE
    " Loud Tones"       LOUD.TONE  C.TITLE
    154 62              LOUD.TONE  C.POSITION
    TONE.WINDOW         LOUD.TONE  APPEND.CONTROL
NEW.CONTROL  LONG.TONE   \ Tone duration check box definition
    A.CHECK.BOX         LONG.TONE  C.TYPE
    " Long Tones"       LONG.TONE  C.TITLE
    154 31              LONG.TONE  C.POSITION
    TONE.WINDOW         LONG.TONE  APPEND.CONTROL
NEW.CONTROL  TRIPLE.TONE \ Triple tone check box definition
    A.CHECK.BOX         TRIPLE.TONE  C.TYPE
    " Triple Tones"     TRIPLE.TONE  C.TITLE
    154 92              TRIPLE.TONE  C.POSITION
    TONE.WINDOW         TRIPLE.TONE  APPEND.CONTROL            ->

Block 6:
( Simple Tones - control handling)                           ( Ø51Ø86 KWE)
: DO.TONECTRLS   ( ctrl value – | do function based on ctrl value)
        CASE     LTONE.BUT    @  OF DO.LTONE            ENDOF
                     HTONE.BUT    @  OF DO.HTONE            ENDOF
                     TRIPLE.TONE  @  OF CHANGE.TRIPLE.TONE  ENDOF
                     LOUD.TONE    @  OF CHANGE.VOLUME       ENDOF
                     LONG.TONE    @  OF CHANGE.DURATION     ENDOF
        ENDCASE ;
: CTRL.SELECTED
     LAST.CONTROL  DUP TOGGLE.CONTROL  DO.TONECTRLS  ;
                                                                  ->
```

```
Block 7:
( Simple Tones - control handling, con't)                ( Ø51Ø86 KWE)

: DO.TONE.CONTROLS     ( - | main execution word for controls)
      IN.CONTROL?                \ Is mouse down in control region?
                IF FOLLOW.MOUSE     \ Was mouse up in control region?
                        IF CTRL.SELECTED THEN
            THEN ;

: TONE.PROGRAM    ( flag - | program for tone.window)
    IF BEGIN DO.EVENTS MOUSE.DOWN =  \ check for mouse button pushes
           IF DO.TONE.CONTROLS THEN AGAIN THEN  ;

TONE.WINDOW ON.ACTIVATE TONE.PROGRAM
 \ Assigns TONE.PROGRAM as program to be run when
 \ TONE.WINDOW is activated
                                                          ->

Block 8:
( Tone menu definition)                                  ( Ø51Ø86 KWE)

99 CONSTANT TON.MENU                    \ Constant for tone menu

: TONE.MENU  ( -- )
   TON.MENU DELETE.MENU Ø                \ Delete menu if it exists
   " Tones"  TON.MENU NEW.MENU           \ Menu title
   " Tone Window/T;Quit/Q"               \ Menu item definition
   TON.MENU APPEND.ITEMS     DRAW.MENU.BAR   \ Add items to menu list
   TON.MENU MENU.SELECTION:  Ø HILITE.MENU   \ Make menu
    CASE 1 OF    TONE.WINDOW CHOSEN  ENDOF   \ Make tone.window active
         2 OF    BYE                 ENDOF   \ Exit forth
    ENDCASE  ;
TONE.MENU                                 \ Add the menu to the screen
```