An Input-Output Model for Interactive Systems

Mary Shaw

Computer Science Department
and
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, Pa 15213 USA

Abstract: Interactive user interfaces depend critically on underlying computing system facilities for input and output. However, most computing systems still have input-output facilities designed for batch processing. These facilities are not adequate for interfaces that rely on graphical output, interactive input, or software constructed with modern methodologies. This paper details the deficiencies of batch-style input-output for modern interactive systems, presents a new model for input-output that overcomes these deficiencies, and suggests software organizations to take advantage of the new model.

1. Introduction

Input and output are perhaps the most systematically neglected features of programming languages. They are usually ad hoc, and they are usually poorly integrated with the other facilities of their hosts -- the languages in which they are embedded. Input and output are generally supported only for the primitive scalar types of the host languages, although they are occasionally supported (though usually in an inflexible way) for nonscalars such as records and arrays.

The situation was bad enough before the introduction of abstract data types and interactive graphic displays, but these additional complications have overburdened the classical ad hoc input and output mechanisms beyond their design limitations. It is now time to develop a sound model for input and output that will address the problems introduced by modern programming technology. Such a model will help to put classical input and output on a solid footing; it will also provide a basis for abstract data types and interactive systems.

Interactive input and output are fundamentally different from conventional implementations of input and output in two ways:

 ACM 0-89791-180-6/86/0400 - 0261 $00.75

- The output device serves as a continuous sensor or observer of the application software and provides current information about the state of the computation, whereas conventional input and output provides information to the human user only when the application software chooses to report.
- Input is an interactive process requiring feedback (sometimes from the application software that will receive the input), whereas input is conventionally treated as a simple parsing task. Moreover, interactive input is often under control of the human user rather than the program, yielding an event-driven system rather than a program-driven one.

The model developed here explains both conventional input-output and the newer, event-driven view of interaction.

This paper begins by suggesting an informal model of input and output that explains classical -- that is, batch-oriented -- input and output. It then describes the problems introduced by modern programming technology and extends the model to deal with them. The paper closes with some remarks on experience with software organizations to support the model.

2. Classical Input and Output

Input and output facilities in programming languages are responsible for receiving input data from the external world and for returning output to the external world. In order to do this, these mechanisms must be able to convert input data to the internal representations used by the program and then convert the internal representations back to output form. Thus, the crucial issue for input and output is *change of representation* for given data values.

We are concerned here with input and output between programs and humans rather than with communication among programs or between programs and mass storage. We are interested primarily in imperative (Pascal-like) languages, but most of what we say applies to other classes of languages as well. The model relies heavily on the use of abstract data types -- either with or without direct language support. It has also been influenced by the object-oriented style of program organization.

The programming languages of the 1960's and 1970's provided specialized input and output facilities for

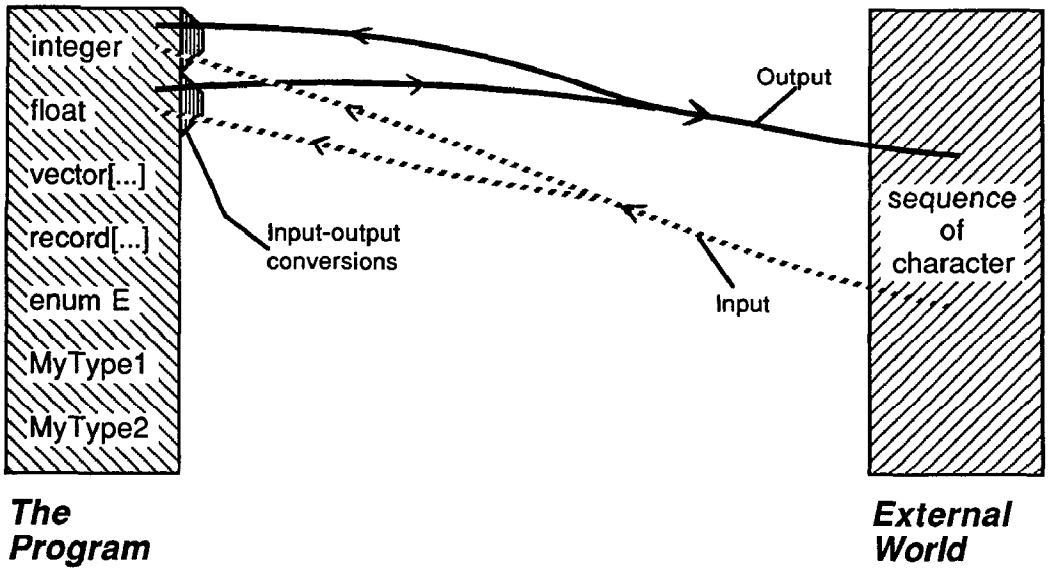

Figure 1: Simple input-output requires conversion between data types of the program and specialized types suitable for direct transmission to physical input and output devices.

processing individual lines of text or streams of characters. The roots of these mechanisms lie in batch processing, and interactive computing using conventional video terminals (CRTs) still largely continues this tradition.

This section develops a model that treats input and output as a problem of converting between the data types of the program and some specialized types suitable for direct transmission to the available input and output devices. This suffices to explain classical character-and line-oriented input and output. Section 3 extends the model to abstract data types and interactive systems.

2.1. Minimal Input and Output

At the very least, input requires conversion of data between some specialized string type particularly well-suited for the input-output device (a *device-relative* type) and the built-in types of the program such as integers, floating point numbers, booleans, strings, etc. Likewise, output requires conversion between between the program's built-in types and a device-relative string type well-suited to the output device (often the same as the string type used for input). We will let P_i denote any of the primitive data types of the programming language for which input and output are supported.

The simplest possible view of input and output is thus that some simple string type, say *seq-of-char*, is built into the input-output system. This special type is not in general available for direct use by programmers. For most primitive types P_i the minimal input-output system provides built-in operations with signatures

Out: P_i --> *seq-of-char*
In: *seq-of-char* --> P_i

The functional specifications of these operations simply require correct conversion between the two representations. This specification is essentially syntactic, so we do not provide formal pre- and post-conditions. Figure 1 depicts the problem of simple input-output.

In addition to the direct conversion operation, input and output operations usually have side effects on the input or output stream, such as extending an output file or truncating an input file. With allowances for the variability of these side effects, the model presented here applies both to stream and to line-by-line input and output.

Default use of the input-output facilities of many languages provides roughly this simple level of functionality. In Basic, for example, the commands

```
INPUT x
PRINT x
```

encode the operations

```
x = In(InputStream])
Append(OutputStream, Out(x))
```

If the input stream contains "12.35", the effect of these commands is to set the value of variable x to 12.35 and produce the output "12.35".

2.2. Formatted Input and Output

The simple input-output scheme of section 2.1 is too restrictive for practical use, and virtually all languages support some means of influencing the details of formatting. The formatting information is encoded in a special format notation. We will let F_{in} and F_{out} denote input and output formats, respectively. The encoded formats are passed to the conversion routines along with the variables. Thus the system provides operations with the functionality

Out: $P_i \times F_{out}$ --> *seq-of-char*
In: *seq-of-char* $\times F_{in}$ --> P_i

Figure 2 depicts the problem of simple formatted input-output.

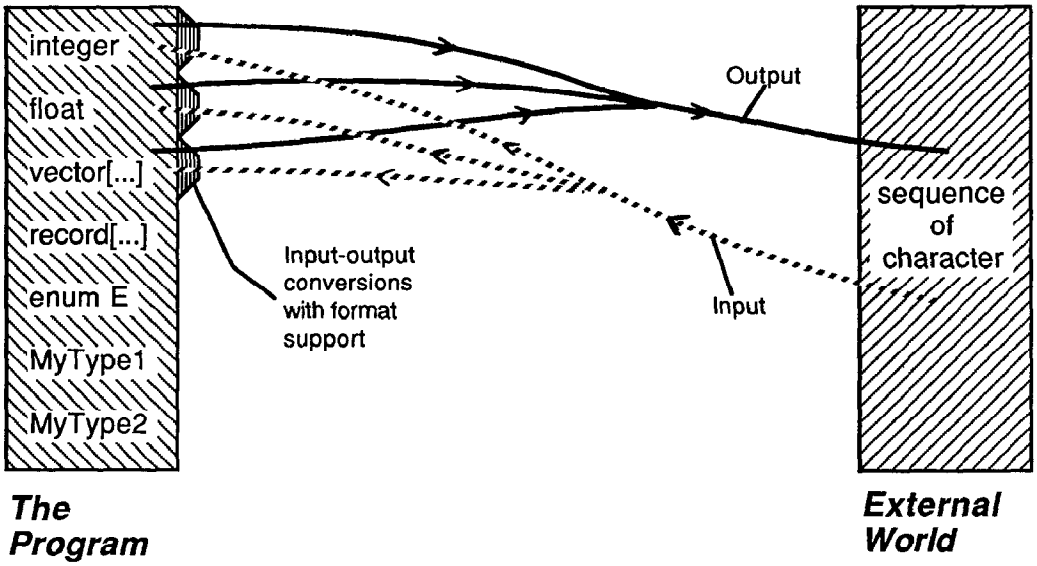

Figure 2: Simple formatted input-output consults format information for each value as it is converted.

A variety of formalisms for formatting have been used. The details are not important here, but the essential requirement for a format notation is that it provide enough information to select from among the conversion alternatives provided by the system. These examples from Fortran, C, and Pascal suggest the possibilities:

Fortran	Pascal	C
99 FORMAT(F5.2) READ (5,99) X WRITE (6,99) X	read(x); print(x:5:2);	scanf("%f",x); printf("%5.2f",x);

If the input stream contains "12.35", all these programs read that input and yield the output "12.35". However, these commands have different effects when the input contains either more or fewer than five characters.

The format languages for input and output, though deceptively similar, are actually not identical. For simple cases like field width or desired precision, the same expression often has reasonable interpretations for both input and output. However, the output processor can make strong assumptions about the internal representation and the possible output forms, whereas the input processor must be prepared to handle any of a number of input strings, including erroneous formations. As a result, the rules for allowable input strings in any particular system may be rigid or fairly flexible.

Format notations span a wide range of expressiveness and power. In addition to specifying the rules for converting individual scalars, they may specify the way individual scalar values are arranged in the input or laid out on the output medium. Special syntax is often used to combine several individual input or output operations into a single invocation. As a result, most input-output systems do not provide precisely the operations described here. These operations do, however, explain the underlying functionality after the syntactic sugar is converted to calls on the basic operations. With the extensions of section 2.3, they explain most existing formatted input-output systems.

The specifications of the *In* and *Out* operations must be refined to account for the format parameters, but the operations are still fundamentally syntactic: they require the conversion between the two representations to adhere to the format restrictions and to be correct within the accuracy specified by the format. Thus detailed specifications of the operations *In* and *Out* depend on the semantics of the particular format language being used. For purposes of this paper, the choice of format language is not significant, provided it can express at least the usual range of variability.

The operations presented here provide a model that explains the most significant characteristics of formatted input-output systems. Since the syntactic idiosyncrasies tend to obscure the essential character of input-output as a process of changing representations, they are ignored here.

2.3. Realistic Input and Output

Even with the addition of formatting, this formulation of input-output is still too restrictive, especially for output. The actual output of a system is often influenced by information about the state or history of the input and output transactions. For example, if a table of numbers is being printed, the position of individual numbers on a line and the number of spaces produced between individual numeric values must be adjusted for column alignment. For a second example, the system may provide page numbers and column headers whenever a page fills up or may align columns on a page. Thus input and output for many systems require state information that persists beyond individual invocations of the conversion functions. Since this state is persistent, operations must be provided to initialize and manipulate it. These operations are commonly implicit in the format language.

In order to explain actual input and output systems, we introduce an explicit state, *IOState*, to capture the state of the input-output facility that persists beyond in-

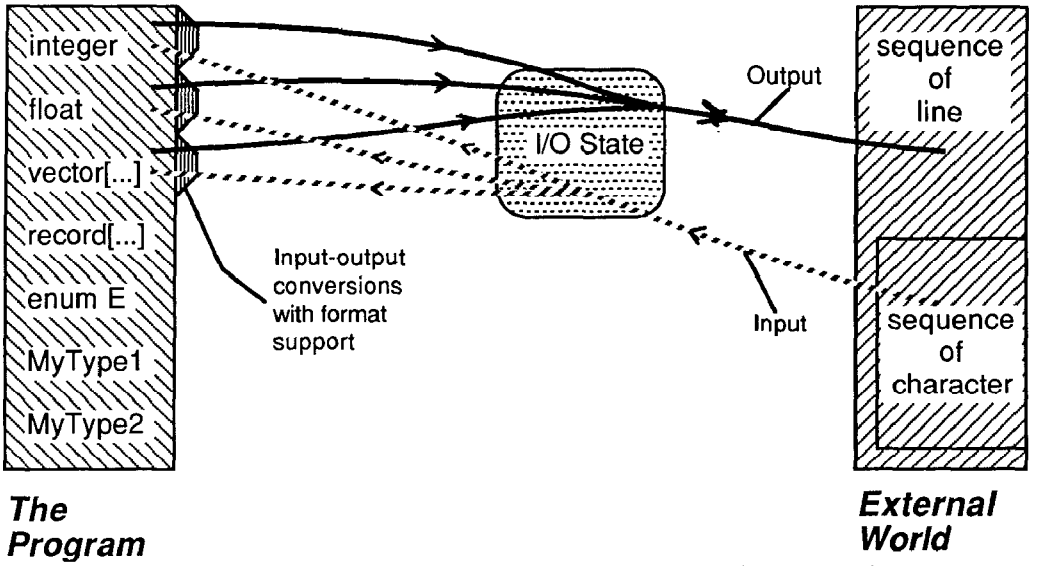

Figure 3: Realistic formatted input-output requires history or state information in addition to formats for individual values.

dividual primitive operations. This state influences the interpretation of both input and output operations, it can be manipulated either explicitly (by *SetUp* operations) or implicitly (as a side effect of *In* and *Out* operations. We also select a richer type than *seq-of-char* for the device-relative object type of the *Out* operation in order to support full-page layouts. For the case of classical character-oriented output the output can be segmented into lines (with vertical alignment controlled by *IOState*), so we will use *seq-of-line* to represent the richer device-relative output type. Real systems therefore provide input and output mechanisms to support operations with signatures more like

$$Out: \; P_i \times F_{out} \times IOState \rightarrow seq\text{-}of\text{-}line \times IOState$$

$$In: \; seq\text{-}of\text{-}char \times F_{in} \times IOState \rightarrow P_i \times IOState$$

$$SetUp: \; IOState \times \ldots \rightarrow IOState$$

The *SetUp* operation is only partially specified here; this definition is suggestive of a class of operations that take parameters appropriate to the state change being performed. The scope controlled by the input-output state need not be restricted to single phrases or lines. In RPG, for example, a page restore and new column headings can be generated automatically each time an output page is filled. Figure 3 depicts the problem of realistic formatted input-output.

The role of the state is demonstrated by the Fortran format and statement

```
100 FORMAT ('1 New page header',(/10(F10.2, 2X)))
200 WRITE (6,100) V
```

These statements print all the elements of vector *V*, beginning on a new page with page heading, printing ten elements per line, until the vector is exhausted. The bookkeeping required to keep track of information such as the current position in the vector, the current column, and the current line depends on maintaining a correct representation of the current state of the output transaction while it is in progress. With respect to our model of input and output, these Fortran statements form a composite operation that could be decomposed into *Out* format operations for the individual vector elements and blank spaces together with *SetUp* operations to handle the overall layout.

3. Abstract Data Types and Interactive Interfaces

The interface between people and computers is a critical factor in effective computer use. This interface has become increasingly important with the advent of interactive systems with high-performance graphic displays. I will now turn to the problems of input and output for such systems.

As noted above, early interactive systems used input and output mechanisms designed for batch processing, simply treating the teletype as both an input and an output device. When video terminals replaced teletypes, we retained the same strategy (but we lost the ability to look back on the paper listing for results that had been produced a few dozen lines previously). The record of an interactive session that takes place in this mode, with line-by-line input and output scrolled on a display, is called a *typescript*. Typescripts are often discarded as they are displayed, but some systems provide a mechanism for logging them in a file.

In a typescript-oriented system, the output is a program history -- a stream of snapshots of subsets of the program state. The subsets of the program state and the timing of the snapshots are controlled by the program that is executing. A system that provides only this form of output neglects the needs of the human user, who is often better served by a single, continuously updated, view of the program state. It ignores the advantages of providing the human user with control over the particular view to be displayed. In addition, it fails to exploit the human's ability to use (and the terminal's ability to provide) cues based on keeping current values of particular information in fixed positions.

In contrast, a two-dimensional interface presents a variety of information simultaneously and updates it dynamically; a given piece of information can be kept up-to-date in a fixed position. Work on text editors [Card 83, Meyrowitz 82] and interaction techniques [Embley 81, Hirsch 81] support the intution that two-dimensional displays are better than one-dimensional typescripts. Thus, the availability of inexpensive high-performance displays provides an opportunity for *qualitiative* improvements to interactive interfaces. Sophisticated display interfaces are currently difficult and expensive to develop, and a number of research projects are addressing various aspects of the problem [Baecker 79, de Jong 80, Green 81, Guttag 80, Hayes 82, Kernighan 81, Mallgren 82, Myers 80, Reisner 81, Rowe 83, Teitelman 84, VanWyk 82, Wallis 80]. The present work contributes a model for input and output that is complementary to those methodological explorations.

Because the role of the human user is much more significant in interactive systems than in batch systems, the significance of the timing of input and output events is more critical in interactive systems. Sensitivity to event timing occurs in several ways.

- The feedback provided by the system must be properly synchronized with input from the user. For example, if a system prompt is provided, it must actually be sent to the output device before user input is supplied. Further, feedback is usually provided to the user as each input phrase is developed.
- Any system that allows the user to provide input values by pointing to values currently displayed on the screen must be extremely careful to update the display whenever the stored values change. Otherwise the user may provide incorrect input by pointing to a displayed image that does not match the stored value that will be inserted in the input stream.
- Interactive systems must cope with input provided asynchronously by the user: processing of the command that means "terminate this process immediately" should not wait until the current process terminates on its own.

These timing issues force the implementor of the input-output system to rethink such issues as the use of buffers, the significance of system delays, and the use of interrupts.

In addition, advances in software and hardware technology have introduced new problems. These include system provisions for

- *Abstract data types (user-defined types)*: Appropriate input and output conversions for user-defined types should be selected as part of the design of the abstract type. This requires type-specific knowledge, but conventional input-output systems (and the model presented in Section 2) do not provide a way to supply that information to the input-output system. Similar considerations often apply to primitive nonscalar types such as arrays and records.
- *Two-dimensional output*: High-performance displays provide a graphic capability that can enhance the quality of interfaces. This requires the ability to construct two-dimensional images that represent data values, but conventional input-output systems do not support graphics.
- *Interactive input*: Interactive sytems must provide feedback to the user while he or she supplies input actions. In addition, input operations may refer to information that is currently displayed on the terminal. This requires active involvement of the input-output system in feedback and the interpretation of pointing actions, but this notion is entirely foreign to conventional input-output systems.

We will consider each of these problems in turn, then extend the informal model of section 2 to show how to deal with them. The purpose of this section is to present the problems introduced by modern programming systems. The next section proposes a model for input and output in such systems.

3.1. User-Defined and Nonscalar Types

During the 1970's, work on abstract data types showed how to extend the set of primitive types provided by a language to include types defined by the user [Liskov 77, Shaw 81]. One of the objectives of this work was for the programming language and support system to provide the same system support for types defined by the user as it provided for primitive types. Although the work was generally successful, it did not succeed in providing genuine type extensibility for input and output.

Full support for input-output in the presence of type extensibility (or even system-supported nonscalar types such as records) requires solutions to two problems

- Providing the type-definer with control over conversion between internal representations and the device-relative types used for input and output.
- Providing the type-definer with access to system input and output facilities, including the *IOState*, the facilities for constructing output renderings by composing renderings of components, and the facilities for parsing input streams.

Thus, in order to make the input-output facilities extensible to new types, it is necessary to provide a way for user-defined conversions to be added to the set of conversions known to the input-output system. Some systems provide default conversions from internal representations to strings, but since these have no knowledge of the type abstractions involved, they can do no better than dumping the internal structure of the type. However, this is not sufficient. Consider, for example, a set represented as a vector (used for the set elements) and an integer (used for the current number of set elements). A common representation stores current set elements in the lowest-indexed array elements. A default output conversion could do no better than converting the integer and all the array elements to strings and inserting a few standard delimiters. A type-specific conversion, however, would convert only the active elements, separate them with commas, and add set braces at the front and back. The integer would not be converted for output; it would be used only to determine how many array elements to convert.

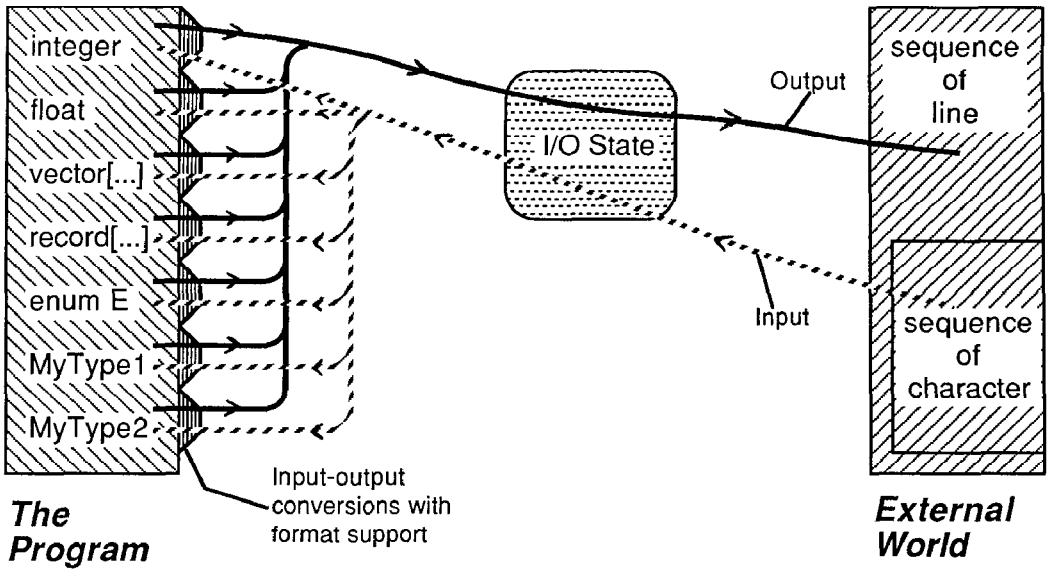

Figure 4: Input-output for user-defined and nonscalar types requires the formatting capabilities to be provided for all types, not just for primitive scalar types.

Since user-defined types are often composite structures whose elements are of other types, reasonable implementations of input and output must also be able to call on the input and output conversions of the other types as subroutines. In order for this to work, the device-relative types used for input and output conversion must be available to programmers as actual data types. (For this reason, type-specific routines that do individualized conversion but interact directly with the input or output devices also fail to solve the whole problem.) This facility is not typically provided by conventional programming languages. There are some exceptions: the functions *sscanf* and *sprintf* in C and so-called "core-to-core" input-output in certain other systems allow the programmer to obtain the results of built-in input-output operations. However, even these systems do not allow user-defined types to be added to the built-in input-output conversion tools.

For user-defined types, we will let T_i denote any user-defined type. Only the designer of the user-defined type T_i has enough information to know how to convert between T_i and the standard special type used for input or output. In addition, proper formatting for nonscalars varies a lot from application to application. Therefore we need a richer input-output facility, one that supports user-defined types T_i wherever it supports primitive types P_i. The required functionality is achieved by adding T_i to the set of types required by the signatures of the *In* and *Out* operations:

$$Out: \{P_i, T_i\} \times F_{out} \times IOState \dashrightarrow seq\text{-}of\text{-}line \times IOState$$

$$In: seq\text{-}of\text{-}char \times F_{in} \times IOState \dashrightarrow \{P_i, T_i\} \times IOState$$

$$SetUp: IOState \times \ldots \dashrightarrow IOState$$

Although it is straightforward to extend the signatures of the specification, it is harder to implement the change. The type-specific conversion routines will be provided by the user program, and it is not practical to recompile the runtime system each time the set of types under consideration is changed. Therefore, in addition to allowing the programmer to define the operations themselves, the system must provide a mechanism for registering these definitions with the input-output control mechanism so that they can be invoked as necessary. Figure 4 depicts the facilities required for input and output of user-defined and nonscalar types.

3.2. Two-Dimensional Output

The introduction of high-resolution, high-bandwidth displays has made a qualitative difference in the kinds of user interfaces that can be defined. The use of graphics literally adds a new dimension to the class of images that can depict data values, and the ability to maintain particular data values at fixed locations on the display allows the user to take advantage of physical layout to avoid searching a typescript for information. Three new problems arise as a result of the new technology:

- Conversion of internal representation to output form that includes graphics as well as text
- Continuous maintenance of the image
- Allocation of space on the display

The power of two-dimensional interfaces is best illustrated with examples. Figure 5 shows the display during an interactive session using the Xerox Cedar programming environment [Teitelman 84]. In this display, the bottom row of small images denote processes that are alive but not of current interest. Three processes are active, each within one window. The three windows are arranged so as to use all the display without allowing any window to obscure any of the others. At the top of each window and at the top of the screen, menus provide quick access to common commands.

Figure 6 shows the Microsoft Chart application running on an Apple Macintosh. The program constructs plots from given data. One window is used for the data for each plot, and several plots can be combined in a single

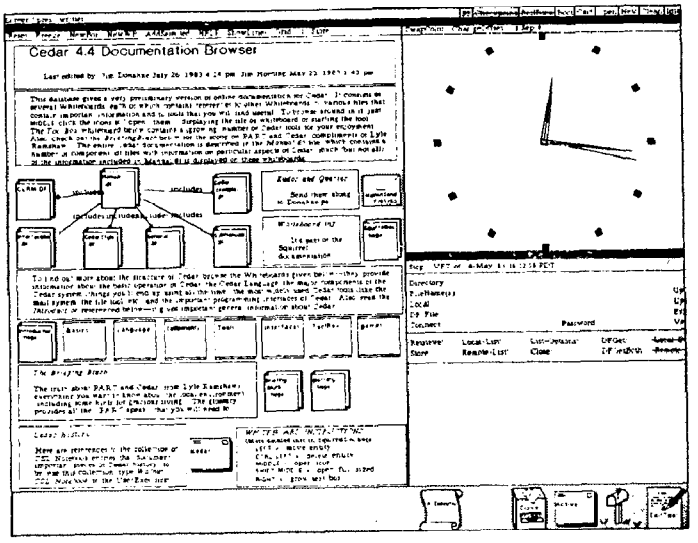

Figure 5: Display layout in Cedar Documentation Browser.

graph. This example differs from the previous one in several ways. First, the windows are allowed to overlap; simple user actions select the window that is active, or intuitively "on top," at any time. Second, all the windows shown here belong to the same process. In particular, changes to either of the data windows are reflected in the window that contains the graph.

Decisions such as the number of windows to associate with a process and the policy for sharing display space among several windows are included in the design decisions make when interfaces are created.

Clearly, the sequence of characters is not an adequate internal representation for images directed to a graphics output device. Therefore, a new internal type for constructing the output image is required. A number of choices are possible, depending on characteristics of the graphics device, but they must support images that include at least character strings, lines, and colored (pattern-filled) areas. For the sake of discussion, we'll call the type *Image*; it might be represented as a display list of strings, lines, and shapes.

User-defined types require the ability to create partial images and compose them to form the complete image to be displayed, just as they did for character-sequence output. Less obviously, this facility can be used to good advantage by primitive types of the programming language. Since two-dimensional images are richer than character sequences, the internal state required to compose them will also be richer.

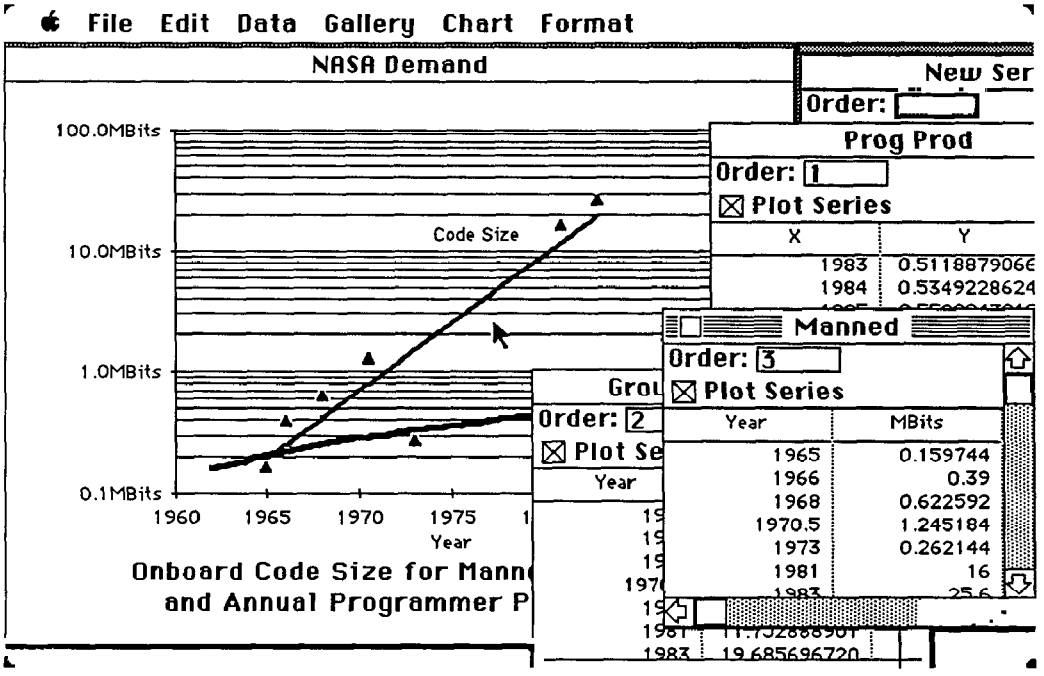

Figure 6: Display layout in Macintosh Chart application.

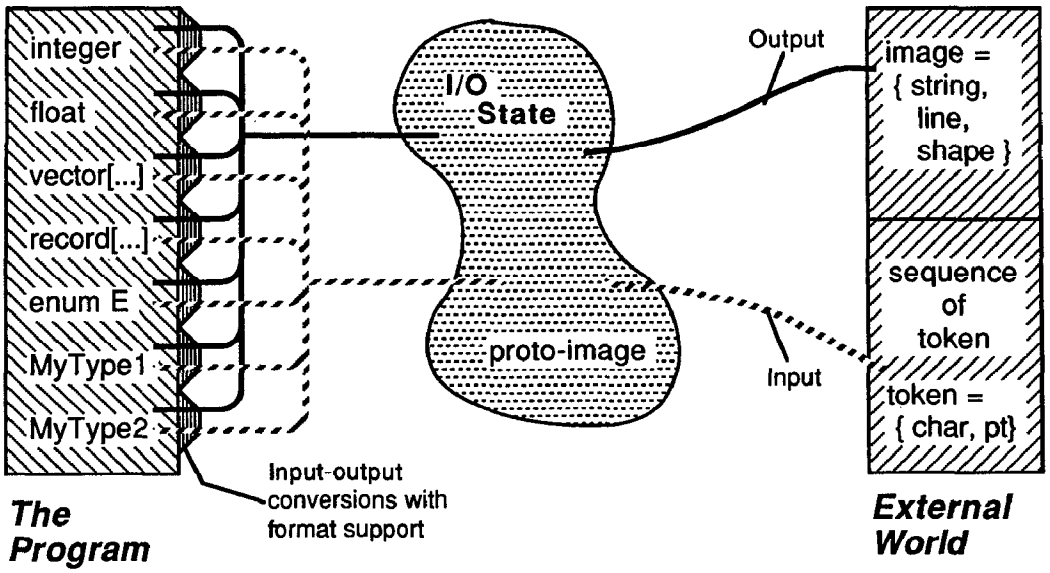

Figure 7: Two-dimensional output requires a significantly larger intermediate state and a richer target type.

Graphics displays provide the ability to improve on the typescript mode of output as well as on the image generated for individual values. It is useful to regard the entire display as a single image that is built up from images of individual values. This decision allows the same mechanisms to be used for constructing images of individual values and for composing the entire display.

When the problem is viewed this way, the generation of a partial image is similar to the generation of a single scalar value to print in a typescript. The partial image can be produced without binding all display decisions, such as color and position on the display. Decisions about the proper location of each image can then be represented in the internal state of the input-output system and used to bind any remaining decisions about the display when the partial images are composed into the complete display.

A partial image can be generated with limited reference to the internal state of the input-output system; all the information required can be encoded in the output format. We will separate this task of producing partial images from the task of composing several partial images into a displayable one. To describe this technique for avoiding premature binding, we'll call the partial images produced by type-specific routines *ProtoImages*. Such images might lack information required for actual display, such as exact screen position. Further format information, F_{comp}, is then required to guide the composition of partial images into complete ones.

This results in a separation of concerns that assigns steps of the conversion task to software elements with appropriate knowledge

- Type-specific routines convert from internal representations to *ProtoImage* form; this conversion is influenced by format information F_{out}
- Composition of individual *ProtoImages* into a complete display *Image* is performed in a module devoted to interface management. This is the logical locus of information about overall layout, which may be encoded as format information F_{comp}.
- Actual display of a complete *Image* is the responsibility of the implementation of type *Image*.

In object-oriented terms, each of these steps would be a method of the appropriate type.

The *IOState* contains the information to control formatting and layout. A set of functions which we call generically *QueryStyle* must be provided to extract pertinent format information (e.g., F_{in} from the *IOState*).

We now need functionality like

Out: $\{P_i, T_j\} \times F_{out}$ --> *ProtoImage*
Compose: *ProtoImage* x F_{comp} x *IOState* --> *Image* x *IOState*
In: *seq-of-char* x F_{in} x *IOState* --> $\{P_i, T_j\}$ x *IOState*
SetUp: *IOState* x . . . --> *IOState*
QueryStyle: *IOState* --> $\{F_{in}, F_{out}, F_{comp}\}$

where we allow format information to be supplied to the composition stage as well as to the input and output conversions for particular types. Figure 7 depicts the facilities required for two-dimensional output.

These definitions provide the functionality for creating full-screen images that take advantage of graphics capability. We can now address the second problem of two-dimensional output, the continuous maintenance of the image. An interactive interface of this class presents the appearance of showing the *current* values of selected portions of the program state. It is important for the values on display to be as nearly current as possible, but relying on the programmer to insert output statements occasionally is not likely to achieve this objective. In order to keep the displayed image up to date with the executing program, we need an automatic mechanism for redisplaying elements of the program state whenever they change. This is part of the overall problem of timing, but most current programming languages do not provide adequate support for the task.

The final major problem of two-dimensional output involves allocation of space on the display. Unlike typescripts and paper listings, which may be of indefinite length, the screen of a graphics terminal is of limited size. As a result, it must be treated as a scarce resource and it must be subject to an allocation policy. If decisions about whether to display the values of variables are made on an individual basis, the space required to display the collection will often be much larger than the display. A good interface system should provide allocation policies to mediate this contention for the screen resource. This is logically the responsibility of a module devoted to interface management. Information about allocation decisions -- both requests and actual allocations -- is logically part of the *IOState*. Many different policies can be devised; the details of the policy are much less important than the existence of some policy.

3.3. Interactive Input

The shift from batch to interactive systems involved the human user with the running program in a fundamentally new way. Rather than setting up input that predetermined the course of the computation, the interactive user could inspect intermediate results before deciding how the computation should proceed. This control is exercised by performing input and output to a terminal that allows the user to see the output and provide the input as it is needed. However, interactive input differs in important ways from batch input. Three new problems arise because of the tight involvement of the human user with the system:

- Feedback during input
- Interpretation of pointing operations and inclusion of "clicking", or non-keyboard input events
- Timing considerations

In an interactive system, the user expects to interact with the system, not simply to provide a stream of input characters. As a result, the input collection facilities must provide feedback while the user constructs an input value rather than simply waiting until a complete sequence of input tokens has been collected before starting to parse the input. This feedback may be a simple as echoing characters, processing backspaces, and supporting some sort of operation to cancel the phrase just typed instead of processing it. It may be as complex as validating partial input to allow immediate correction or providing default field values for an input record after some fields have been typed.

Despite the attractiveness of viewing input and output as inverse operations -- a view which has been very useful in older systems with one-dimensional input and output -- the symmetry does not exist in modern interactive systems. In such systems, output combines text and graphics into two-dimensional images. The input, however, is still essentially one-dimensional; although it may contain pointing information in addition to alphabetic characters, it is still a sequence of discrete tokens. As a consequence, formats for input (F_{in}) and for output (F_{out}) must be treated separately. (Note also that the failure of the inverse-operation view increases the difficulty of making unix-style combinations of programs that are individually indifferent to whether they are communicating with humans or other programs.)

When a pointing device (e.g., mouse) is supported by the terminal or workstation, the input process becomes more complex. The input stream still consists of a sequence of discrete tokens, but the tokens are drawn from an alphabet that includes special characters for extra keys, pointing information (cursor location), and perhaps timing information in addition to the characters of the alphabet. This token stream is first converted to lexemes in an alphabet expected by an application process then parsed by an appropriate type interpreter. As a result, the signature of the input routine must be revised to operate on *seq-of-token* instead of *seq-of-char*:

$$In:\ seq\text{-}of\text{-}token \times F_{in} \times IOState \dashrightarrow \{P_i, T_j\} \times IOState$$

We regard the feedback provided by the user as transient output performed under control of the input format F_{in} coupled with automatic display of the input value if the input operation changes a variable that is currently being displayed. Figure 8 depicts the facilities required for interactive input.

A significant use of the extra tokens that denote pointing information is to construct input by referring to information already on display. The two most common cases are menu selection and "cut-and-paste". In both cases the *IOState* plays a crucial role in determining what information was selected. For menu selection it is only necessary to map the cursor position through the display representation to determine which option was selected. In the latter case, one or more pointing actions are used to select a value displayed on the screen. In addition to determining what was on display, the value selected must be converted to a form suitable for input manipulation. In some systems it is converted to a string and inserted in the input stream, but this is not always possible. An alternative is to insert in the input stream a token containing a pointer to the selected value. In all cases where input is created by reference to the display, it is crucial for the display to be consistent with the program state.

The design of an interactive system must also take account of input that may be provided asynchronously by the user rather than in response to direct prompts from the running system. The model developed here does not adequately explain these timing issues.

4. A Model for Interactive Input and Output

In previous sections, I argued that input and output are much more complex and demanding tasks than their usual treatment would suggest. Certainly much more is involved for interactive systems than simply inserting input and output statements at selected points in the program. This section elaborates the model developed in section 2 in order to account for input and output in interactive systems that support user-defined types. Specifically, I propose that interactive input and output be modelled along the following lines:

1. *Legitimize the internal device-relative types used for input and output conversion.* Input and output are based on types suitable for conversion to

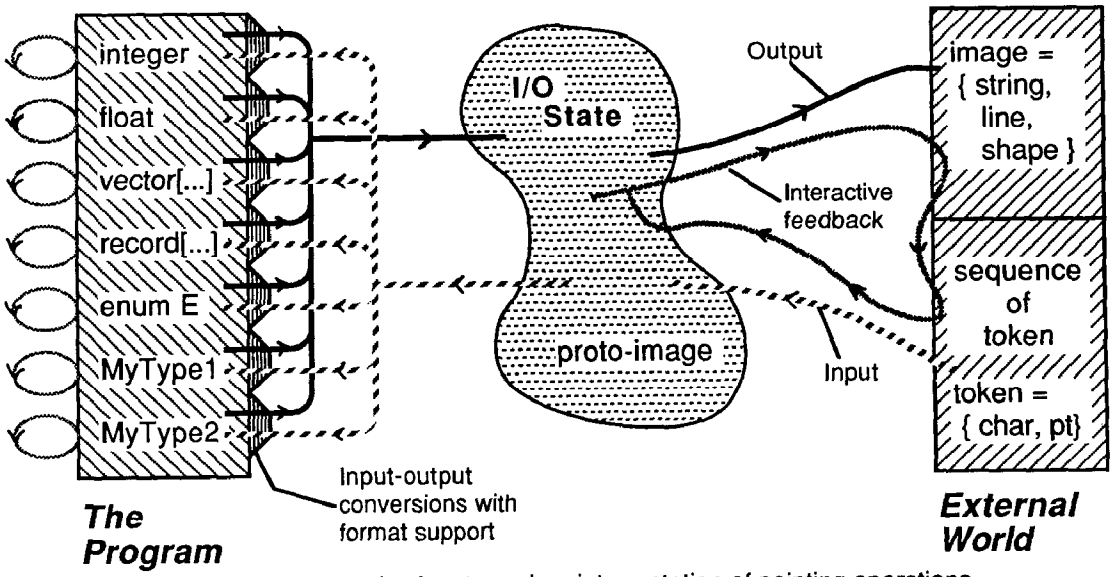

Figure 8: Interactive input requires interpretation of pointing operations and feedback during input.

physical input or output devices. These types are usually hidden from the programmer. User-defined input and output, especially for user-defined types, requires user-defined operations on these types.

2. *Create a well-defined formatting state.* Input and output are context sensitive. They depend on persistent program state and on the states of the input and output streams or devices. This state should be public and explicitly manipulatible by the program.
3. *Define types to support format descriptors that can be manipulated at run-time.* Formatting information must be provided, often from a source other than the main computation (e.g., the input state). Moreover, this information must be processed by procedures in user-defined data types, so the representation of the format information should be comprehensible.
4. *Provide a mechanism for registering input and output routines for user-defined types.* With the auxiliary structures described above, input and output can be treated as ordinary functions. Users can then add functions for new types.

The effect of these steps is to make the mechanisms used for input and output publically available and accessible to the programmer. When these steps have been taken, we obtain a model for modern input-output mechanisms that includes the following components:

- A device-relative type for data arriving directly from the input source: *seq-of-token*
- A device-relative type for formatted values ready to send to the output destination: *Image*
- Conversion operations between internal representations and device-relative types:

Out: $\{P_i, T_j\} \times F_{out}$ --> *ProtoImage*

Compose: *ProtoImage* x F_{comp} x *IOState* --> *Image* x *IOState*

In: *seq-of-token* x F_{in} x *IOState* --> $\{P_i, T_j\}$ x *IOState*

- One or more internal type(s) suitable for parsing input and composing output: *ProtoImage*
- A definition of the persistent input-output state, including display layout and a mechanism for automatically updating the display when necessary
- Operations on the persistent input-output state:

SetUp: *IOState* x . . . --> *IOState*

QueryStyle: *IOState* --> $\{F_{in}, F_{out}, F_{comp}\}$

- Format representations for input and for output, with format descriptors: F_{in}, F_{comp}, F_{out}
- A facility for adding user-defined types or user-defined conversions for existing types to the system on a program-by-program basis.

Like most models, this one can be instantiated in a variety of ways. For example, in addition to these primitive facilities, a programming language will provide composite operations to allow input or output for several values at the same time. However, the model is largely independent of particular input or output devices or or particular programming language syntax. Its chief deficiency is its failure to address timing issues.

5. Experience with the Model and Software Development

The model described here arose from work on Descartes, a system for spsecification and construction of interactive display interfaces [Shaw 83]. A major purpose of Descartes is development of a software organization and specification system powerful enough to define a broad class of interactive styles, rather than the restricted range usually provided by User Interface Management Systems or software packages for constructing intractive interfaces. The model presented above arose as part of the Descartes project.

Just as classical input-output systems fail to support interactive computing, the classical program organization also falls short. Input and output through typescripts associate input and (especially) output operations with the

running application code (the *client*). The result is a confounding of responsibilities in which one programmer must be concerned simultaneously with the computation of the application and with decisions about what information is to be displayed, and at what times.

In Descartes, we began with three principles for the design of interactive systems:

- *Strong linkage between display and program:* The display should reflect the current program state at all times.
- *Decoupling of application from interface:* The input-output interface should be separable from the main program.
- *Freedom without license:* Uniformity of interface style is an advantage, but users need guidance about style and organization. However, decisions about display and interaction style should not be gratuitously preempted.

Based on these principles, we established an explicit separation of concerns that led to a system organization with four distinct areas of responsibility:

- *The Client:* The client component is concerned with the computation of the application of interest. Using techniques based on ideas from constraint systems and continuously-evaluating functions, we were able to trigger display updates managed elsewhere whenever information of interest was modified. This is critical to the separation of interface issues from the mainstream computation.
- *Type-Specific Input-Output:* We established standard signatures for input and output routines, provided routines with these signatures to handle the standard types, and thereby allowed user-defined types to be treated uniformly with built-in types. These routines were registered with the utility support code so they could be invoked whenever they were needed.
- *Interface Control Module:* Each client has its own interactive display interface (or possibly more than one). We localized the client-specific portion of the definition of this interface in a single module (which, of course, makes heavy use of the utility support).
- *Utility Support:* These facilities do not arise without cost. We developed extensive runtime support for registering new types, triggering updates automatically, describing formats, and maintaining the I/O state of the system.

6. Notes on Extensions

The model developed here addresses both classical one-dimensional input-output and interactive input and graphical output for a single application process. This section notes some areas in which it would be worth extending the model and comments on each of those extensions.

6.1. Timing and Synchronization

In addition to the problems of input interpretation and output rendering addressed by this model, an interactive system must deal with a number of timing and synchronization problems.

Multiple Input Sources: If a single user employs multiple devices at once -- such as a keyboard, a mouse, a set of potentiometers, and special graphics devices -- to create a set of inputs, we view that collection as a single conceptual input stream. The details of how to implement this will depend on the underlying operating system. In the prototype Descartes implementations, the system automatically merged mouse input into the standard input stream. A solution for graphic devices under Tenex added explicit multiplexing and timing information to preserve the conceptual unity of the stream [Sproull 79].

Unscheduled User Input to Application Processes: In systems that are not purely event-driven, it is desirable for the human user to be able to send signals to the application software while it is executing. At a minimum, this can be simply a "stop executing cleanly" signal, though many elaborations are possible. The issue may even arise in an event-driven system if individual operations are not fast enough. Again, the solution to this problem depends on the underlying system. If an interrupt facility is not available, the application may be forced to poll. If independent processes are supported, the character of the solution may depend on whether they can share address spaces. In any case, unprocessed input that is already pending at the time the user sends the signal must be dealt with appropriately (usually by discarding it).

Consistency between Display and Program State: As discussed in Section 3.3, the creation of input by reference to values on display depends critically on consistency between the display and the program state. Rapid redisplay reduces the severity of the problem, but careful synchronization of input containing cursor references with display updates is required for absolute correctness. A solution in the Tenex context is given by Sproull [Sproull 79].

6.2. Communication between Programs and Mass Storage

Although this model was developed with human interaction in mind, there are no obvious major obstacles to extending it to input and output between programs and mass storage. The extensions will generally involve the device-relative types used for conversion. Although the mechanisms for automatic redisplay of modified program variables will be useful for some applications, explicit output requests will more often be the useful mode for sending data to mass storage.

Sequential File Access: For sequential access to a file, all that is required is a conversion to and from some suitable (e.g., binary recorded) representation and input/output operations that update the file in addition to converting the values. Conversion to such a representation is usually easy for values that do not contain pointers and hard for values that do. Input is somewhat simpler than in the interactive case because it is not necessary to find the end of the input sequence for a given value.

Logging and Typescripts: It is often useful to create a transcript of a sequence of values. This might, for example, be a log of all values taken on by a single variable or a typescript similar to the sequence of lines produced on an interactive system using one-dimensional input and output. The former could be accomplished using the representations for sequential file access suggested above and the mechanisms for automatic output generation used to update an interactive display. The automatic mechanism would be used in this case to send output to a file. Typescripts, on the other hand, are intrinsically periodic snapshots of collections of variables produced under program control. A simple typescript can be produced by converting the desired messages to strings and sending them explicitly to files. A typescript file produced as a byproduct of an interactive system that (by design) uses a typescript style of output in some window is a little trickier. One possibility is to convert each line and assign it to a string variable, then use the automatic-update mechanisms both to send the string to the typescript file and to update (and send) the display.

Acknowledgements

This model for input and output evolved over a number of years, and I am grateful for the many hours that my colleagues spent discussing the problem with me. I am especially indebted to the members of the Descartes project at Carnegie-Mellon and to the DARPA Quality Software Working Group. Comments from CHI '86 referees were very constructive and contributed to improved exposition. An earlier version of this paper was presented at the Dutch conference, *NGI-SION 1985 Informatica Symposium: uitdaging aan de informatica.*

The research presented here was supported by the National Science Foundation under Grant MCS-80-11409 and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

The author is currently Chief Scientist of the Software Engineering Institute and Associate Professor of Computer Science at Carnegie-Mellon University.

References

Baecker 79. Ronald Baecker, William Buxton and William Reeves. Towards Facilitating Graphical Interaction: Some Examples from Computer-Aided Musical Composition. 6th Man-Computer Communications Conference, National Research Council of Canada and Canadian Man-Computer Communications Society, May 1979, pp. 197-207.

Card 83. Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, N.J., 1983.

de Jong 80. S. Peter de Jong. The System for Business Automation (SBA): A Unified Application Development System. Information Processing 80, IFIP, October 1980, pp. 469-474.

Embley 81. David W. Embley and George Nagy. "Behavioral Aspects of Text Editors." *ACM Computing Surveys 13*, 1 (March 1981), 33-70.

Green 81. Mark Green. "A Methodology for the Specification of Graphical User Interface." *ACM Computer Graphics* (August 1981), 99-108.

Guttag 80. John Guttag and J.J. Horning. Formal Specification As a Design Tool. Seventh Annual Symposium on Principles of Programming Language, ACM SIGPLAN/SIGACT, January 1980, pp. 251-261.

Hayes 82. P.J. Hayes. Cooperative Command Interaction Through the Cousin System. Proceedings of the International Conference on Man/Machine System, University of Manchester Institute of Science and Technology, London, July 1982.

Hirsch 81. R.S. Hirsch. "Procedures of the Human Factors Center at San Jose." *IBM Systems Journal 20*, 2 (1981).

Kernighan 81. B.W. Kernighan. "PIC - A Language for Typesetting Graphics." *ACM SIGPLAN Notices 16*, 6 (June 1981), 92-98.

Liskov 77. Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert. "Abstraction Mechanisms in CLU." *Communications of the ACM 20*, 8 (August 1977).

Mallgren 82. William R. Mallgren. "Formal Specification of Graphic Data Types." *ACM Transactions on Programming Languages and Systems 4*, 4 (October 1982), 687-710.

Meyrowitz 82. Norman Meyrowitz and Andries van Dam. "Interactive Editing Systems, Parts I and II." *Computing Surveys 14*, 3 (September 1982), 321-415.

Myers 80. Brad A. Myers. *Displaying Data Structures for Interactive Debugging.* Ph.D. Th., MIT, June 1980.

Reisner 81. Phyllis Reisner. "Formal Grammar and Human Factors Design of an Interactive Graphics System." *IEEE Transactions on Software Engineering SE-7*, 2 (March 1981), 229-240.

Rowe 83. L.A. Rowe and K.A. Shoens. "Programming Language Constructs for Screen Definition." *IEEE Trans. on Software Engineering SE-9*, 1 (January 1983), 31-39.

Shaw 81. Mary Shaw (editor). *Alphard: Form and Content.* Springer-Verlag, 1981.

Shaw 83. Mary Shaw, Ellen Borison, Michael Horowitz, Tom Lane, David Nichols, and Randy Pausch. Descartes: A Programming-Language Approach to Interactive Display Interfaces. Proc. SIGPLAN Symp on Programming Language Issues in Software Systems, ACM, June 1983, pp. 100-111.

Sproull 79. Robert F. Sproull. Raster Graphics for Interactive Programming Environments. Tech. Rept. CSL-79-6, Xerox Palo Alto Research Center, June 1979.

Teitelman 84. Warren Teitelman. "A Tour Through Cedar." *IEEE Software* (April 1984).

VanWyk 82. C.J. Van Wyk. "A High-level Language for Specifying Pictures." *ACM Transactions on Graphics 1*, 2 (April 1982), 163-182.

Wallis 80. Peter J.L. Wallis. "External Representations of Objects of User-Defined Type." *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 137-152.