Descartes: A Programming-Language Approach to Interactive Display Interfaces

Mary Shaw, Ellen Borison, Michael Horowitz, Tom Lane, David Nichols, Randy Pausch

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract: This paper shows how the principles of programming methodology and language design can help solve the problem of specifying and creating interactive display interfaces for software systems. Abstraction techniques, such as abstract data types, can support both the specification of display interfaces and the implementation of those interfaces in a variety of styles. These abstraction techniques also guide the organization of software systems that will use display interfaces. We are developing a system that includes specifications, interface description tools, prototype organizations, and runtime support. The emphasis is on flexibility and on the separation of policy from particular instances. Preliminary results from implementations in a prototype domain indicate the feasibility of the approach.

1. Introduction

The Descartes project extends research on abstraction techniques in programming languages to a new problem domain: the design and creation of interactive program interfaces that use high-performance displays. The programming-language viewpoint helped us to separate independent issues, to understand the degree of generality and flexibility required, and to organize the program structure.

The interface between human users and computers plays a critical role in effective computer use, especially for naive users. The relative costs of human professional time and computer time have shifted to place the premium on professional time, and that professional time is currently underutilized. One study showed that information retrieval, text processing, and automated calendars have high potential for improving the use of professional time [35]. Since many professionals are not computer professionals, the *quality* of the user interface is important.

In the past, interactive interfaces have usually been one-dimensional: the user typed a sequence of independent commands and the program appended input and output text to a *typescript*, or textual record of the session. 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 and in a reserved screen position. Studies of text editors [6] and interaction techniques [9, 22] support the intuition that two-dimensional displays are better than one-dimensional typescripts. Thus, the availability of inexpensive high-performance displays provides an opportunity for *qualitative* improvements to interactive interfaces. Unfortunately, sophisticated display interfaces are currently difficult and expensive to develop.

The specific objective of this project is to simplify the task of developing interactive display interfaces by applying the techniques of abstraction, specification, and programming language design. To achieve this, we are developing concepts and models for specifying human-computer interactions in a wide variety of styles. We emphasize the use of concepts appropriate to the level of description -- for example, by describing interactions in terms of user-level notions such as "menu" and "scrolling" rather than primitive graphics notions such as "picks" and "valuators." We are also developing software tools that allow specific interfaces to be created easily. We believe it should be easier to construct and to use richly interactive two-dimensional interfaces than it currently is to use typescripts -- and these display interfaces should be so attractive that no one can stand to interact in any other way.

We are concerned with conceptual and software tools for system developers; the entire community of users will benefit if system developers can take advantage of display technology quickly and efficiently. Our primary emphasis is on high-performance displays (high resolution and high bandwidth), but personal computers and "smart" character-oriented terminals are also of concern.

Consider, for example, a program that helps students to understand finite automata by allowing them to define finite-state machines and provide input for simulated execution. Such a program supports several kinds of operations: creating and editing FSM definitions, saving and restoring these definitions as files, simulating execution on specific inputs, and providing instructions on the use of the program. A typescript interface can provide this functionality; indeed, we developed a typescript-driven FSM simulator for class use several years ago. It is easy to imagine other alternatives for the interface: One possibility is a textual display in which the machine definition, input tape, instructions, and so forth occupy fixed positions and are updated independently. Another possibility is a display on which menus are used to select operations and the FSM is defined and manipulated through a graphical display of its transition matrix. Multiple viewports could be used, for example, to browse through on-line course material, communicate with the course instructor, or work with more than one machine definition at a time.

These options and others are suggested by interfaces that now exist. Certainly, the design of the *client program* (in this example, the FSM simulator proper) should be independent of the design of the interface, though the application will naturally set the requirements for the interface. We see a modest number of general styles for these interfaces and an enormous amount of variation in specific details. However, the tools that currently support the creation of the interfaces either require the designer to work at a very low level of abstraction or else preempt the decision about general style and many of the specific details.

Our objective is to find a unifying framework for these design alternatives and develop a set of supporting tools and design techniques. These results must allow the interface designer to range freely over the design space and to instantiate a design easily for any interface device with adequate power. The remainder of this paper surveys previous work in programming methodology and graphics, discusses interface specification and implementation issues, and describes our early implementation results.

2. Background

Since Descartes draws heavily both on the methods of programming languages and on the examples and techniques of display interfaces, we will briefly review these areas and suggest a basis for their interaction.

2.1. Methodological Basis

Descartes builds on a substantial history of abstraction techniques in programming languages [42]. In the 1970's, work on abstract data types refined an intuition ("organize programs around major data structures") and some examples into a systematic theory. This theory had specific requirements for program organization [34], language support [26, 29, 43], formal specification [16], and verification [23]. Although the abstract data type provides a good paradigm for organizing many programs, it is not suitable for *all* programs. Nevertheless, the process of developing the theory of abstract data types can serve as a model. The methodological goal of this research is to improve our understanding of how useful theories about program organizations emerge from practical intuitions.

The past decade's results on abstract data types offer both guidance and stimulation. Whereas abstract data types were explored largely in the context of general-purpose programming language design, we are working with specifications, system organization, and the special abstractions needed for dynamic interactions. We believe it is now appropriate to focus on a narrower task area, trading generality for problem-specific power.

In addition to using specification techniques for abstract data types, we can build on existing formal specifications for graphics [30], viewports [17] and interactive input [37]. Interactive interfaces have several separable components, including display layout, input protocols, computational properties, and the relation of the interface to the client program. Thus recent research on writing and combining partial specifications [7, 18] is also pertinent.

In a broad sense, of course, interactive interfaces implement languages for controlling programs, so many of the design criteria for programming languages also apply to these interfaces. Interfaces are much less concerned with complex control flow than are general purpose languages, and they are much more concerned with input interpretation, output formatting, and ease of use. Concepts such as data structure, scope, extent, binding, and abstract definition apply in both cases. Although programming languages provide little guidance about input and output, we can draw on related work on output for diagrams [25, 46] and data types [32, 47].

2.2. Prior Work on Display Interfaces

A number of hand-crafted systems have explored the potential of the display medium, often for a specific application domain such as electronic mail handling [3], music synthesis [4], business automation [8], or creating documents with both text and graphics [10, 21, 41].

The chief difficulty in developing and evaluating interactive display interfaces is that they are hard to build. For example, a study of interactive business applications showed that display generation and management code typically constitutes a *majority* of the code [44]. This has naturally led to work on systems and tools which aid in constructing display interfaces.

Existing systems for developing display interfaces cover very limited domains. Typically, only one interface style is supported, such as "form-filling" in a fixed network of forms [40, 39, 20] or a "table top" of overlapping viewports on distinct processes [41, 45]. Substantial control over display appearance is offered, but user interaction protocols are predetermined.

Some principles for organizing interactive systems have been suggested; they address questions such as ways to compose definitions of independent components [7], the use of databases and assertions about data dependencies [12], global metaphors for interaction [1, 13], robustness to human error or misunderstanding [19], the organization of complex interaction scenarios [38, 49], and means for avoiding dependence on specific hardware [1, 39].

A variety of graphics support software also exists: the Core graphics standard [15] is supported with software packages [11, 33, 48]; operating systems have been adapted to support interfaces to several processes through distinct viewports [27, 45]; and specialized systems for generating interfaces of specific types have been written [2, 8, 24, 28, 49].

However, experience with high-performance displays is not yet widespread; the available tools usually preempt many decisions about the nature of the interaction; and few general, flexible tools are available to implementors who want to base user interfaces on these displays.

3. Principles for Organizing Interactive Systems

The Descartes design is driven by three principles concerning relations between the underlying application (the *client program*) and the display (as represented by a software module called a *compositor*).

- *Strong linkage between display and client program:* At all times, the display should reflect the current state of the displayed variables. In general, assignment to a displayed variable must be thought of as potentially requiring complete regeneration of the display. Naturally, optimizations are desirable.
- *Decoupling of application from interface:* The input-output interface should be separable from the client program. Software systems should be organized so that it is straightforward to replace one display with another display or with a different kind of interface. One implication of this principle is the separation of general style from specific

layout details in the display design; in turn, this separation makes it easier to support a variety of styles.

- *Separation of policy from instance:* Stylistic uniformity of interfaces is an advantage, but interface designers need guidance about style and organization. Conversely, coercing designers to a single style is too rigid; they need freedom to choose from a selection of styles. However, freedom does not imply complete license; it should be simplest to follow an established style. Hence, stylistic policy should be separable from the layout decisions for any particular interface.

4. Interface Specification Issues

Interface specifications, like specifications in general, should be written in terms appropriate to the design rather than in terms of implementation mechanisms. The specification language must therefore capture the constructs and the kinds of variability that the designer expects to use.

Current theories of abstract data types do not deal adequately with input-output even in linear text, and the problem becomes critical for interactive displays. Before the value of some variable is displayed, its internal representation must be converted to a human-intelligible literal form. Since there may be many output renderings for a given value, the conversion must also take formatting information into account. For one-dimensional output this literal is usually a character string. For two-dimensional output the possibilities also include images of various sorts; we call the resulting literals *icons*.

Even worse, interactive input requires a complex dialogue to provide feedback as the human user creates individual input values. Even in the simplest case, a "backspace" character should remove the offending character from the display instead of adding some deletion character to the tail of the string; extremely complex dialogues can arise when input involves, for example, interactive validation or selection from a collection of displayed alternatives. We call this dialogue *prosody*, in an extension of the definition of "prosody" as the rhythm, cadence, and emphasis of spoken prose.

Designing an interactive interface involves choosing the information from the client program to display, the ways the user may manipulate this information, and the static and dynamic arrangement of the information on the display. Many of the elements included in an interface will be defined in terms of other, more primitive elements. We therefore need a formal specification system that supports precise specifications of individual components of a display and also generic composition rules that allow the creation of new components from existing ones. The major conceptual problems arise from the dynamic nature of interactive computing. We address them here as specification issues.

- *Model for specifying display output:* Program values must be converted to iconic form before they can be displayed. This conversion requires knowledge about the representations of the program data types as well as knowledge about icons and formats. Since most values can be displayed in many different ways, a format notation for controlling the conversion is required. In an interactive system, format control cannot be exercised solely by the client program as it converts values to icons. The compositor and the human user may also need to influence format decisions, so an arbitration mechanism is needed to resolve conflicts.
- *Model for interpreting interactive input:* Client programs receive input that may include references to the display, timing information, and special-device input as well as ordinary characters. As noted above, processing this input may require intermediate feedback to the human user. The prosody, or protocol for this feedback, plays a role for input comparable to the role format plays for output. Like format, prosody requires models for specifying elementary protocols (e.g., "releasing red button selects an element in this viewport" or "a backspace character causes the previous character to be deleted") and for combining them into more elaborate ones (e.g., selection from a multilevel menu or inspection of a document by scrolling). Another problem of interactive input is interpreting "pointing" -- that is, determining precisely what value is denoted by an input token such as a mouse selection of a displayed icon.
- *Specification system:* An interface design has several components; each is complex enough to warrant individual attention. We must therefore be able to develop partial specifications for properties such as functionality (e.g., "selecting '#' saves the definition"); geometry (e.g., "instructions are displayed at the bottom"); formated appearance (e.g., "header is black on gray with white border"); and prosodic behavior (e.g., "red button changes viewport size"). Creating full specifications from fragments that deal with different properties involves checking compatibility and determining interactions as well as simply merging the fragments.

In all these cases, ease and uniformity of design will be improved by localizing general policy decisions about format and prosody. Following the example of the Scribe text formatting system [36], in which document layout policy and physical device characteristics are obtained from a database rather than being defined with each document, we establish *style* definitions that provide initial reasonable choices for these design decisions. A database for interface development must provide for a variety of interface styles that are internally coherent, though different style definitions may lead to rather different interfaces. When a designer selects a style, defaults are established for numerous decisions on format and prosody, so the simplest actions for the designer produce uniform, usable interfaces.

4.1. Specification of Components: Menus

A major task in the development of a system such as Descartes is identifying suitable abstractions for interactive communication with programs. These abstractions may ultimately be implemented, for example, with the elementary primitives of the graphics standards, but considerable extra structure and support is required before they are suitable for interface design. In this example we examine one familiar example of an interface component, the *menu*. We establish its role in an interface, suggest the stylistic variety of menus in various existing systems, and establish a design space that largely explains that variety.

Interactive programs often require the user to supply a value from a pre-defined set, such as a value for an enumerated type.[1] In typescript systems, the user typically supplies the value by typing a literal string. In two-dimensional interfaces with pointing devices,

[1] The case of a dynamic set is similar provided all values are known when the menu is accessed. More generally, it may be desirable to select several values from an enumeration (i.e., a value from a powerset), but for this example we consider only the simpler case.

the option of selection from a displayed *menu* allows the use of one precious resource, screen space, to reduce the load on another, the user's time and attention. Many variations on the menu theme exist in current systems [3, 4, 10, 13, 14, 32, 41]. This example describes some design alternatives that account for most of that variability and hence provide a model for specifying a large class of menus. The resulting comparison indicates that no version is clearly superior, but individual designers clearly hold a variety of strong opinions.

Three kinds of information are required to define a menu: linkage to the client program, display format, and input prosody.

- The linkage to the client program has two components: the set for which the menu is providing selection and a variable to be set by the selection operation.
- Display format requires format decisions for assembling the iconic literals of the menu's elements, including decisions about the *creation* and *presentation* properties described below. An intermediate representation for icons (type *glyph*, described in Section 5.2) provides much of the support for these decisions: for example, the specifications that establish the style for viewports should also establish a matching style for menus.
- Prosody requires a *selection* mechanism that interprets low-level user actions such as key transitions and locator positions to make the actual selection; this, too, can take advantage of existing abstractions and the intermediate representation.

The *creation* properties of a menu describe where, how, and under what circumstances the menu will be visible. The locations and sizes of the viewports to be allocated for the menu are established, possibly on the basis of other information such as the current cursor location. A menu may be "visible or "invisible"; if it is invisible it is not currently consuming display resources. Creation properties include the format rules for those aspects of the menu display that are independent of the particular icons for the alternatives (e.g., the colors of the background and the border, whether the name of the menu is displayed).

The *presentation* properties describe the mechanisms used to present the items for selection. The straightforward approach is to display all the items simultaneously, but this sometimes consumes too much display space. Therefore, the general menu schema must include a way to consider the alternatives selectively. The common options include:

- *Scrolling:* The items are formated into a two-dimensional plane, and a window into that plane is shown in the viewport of the menu. "Scroll bars" on the menu borders allow panning over the plane to bring other items into view.
- *Cycling:* A limited amount of space is used to display (typically) one item, and the user may examine the alternatives in some fixed order. This option differs from scrolling chiefly in the fixed order for exhibiting the alternatives.
- *Subdivision and hierarchy:* The menu alternatives are partitioned, and partitions may be displayed individually. If the menu is hierarchical, selection of an item in one partition may activate the menu for another partition. If the partition is not hierarchical, the partitions may be displayed independently; they may also share screen space by being "stacked up" so that all partitions are partly visible but selections can be made only from the top element of the stack.

The *selection* properties describe how the user's low-level input actions will be interpreted to indicate selection and the feedback that supports this. Common existing selection mechanisms include:

- Simply typing the literal "name" for the item as a string.
- Typing, with command completion or spelling correction.
- Software mapping of function keys to particular items.
- Screen position of a locator device (e.g., a mouse) when a button is depressed or released.

The protocol, if any, for highlighting a tentative selection (e.g., the alternative under the cursor when no buttons are pressed) is also a selection property. More generally, Descartes prosody mechanisms must cover all cases of interest.

The examples of Figure 1 show how menus in several existing systems fit into this design space. Figure 1a shows the Star system's menu [31, 41]. This statically allocated menu contains varied items, some of which have internal structure. When icons are selected, they "display their contents" in a form of submenu (e.g., a folder opens to show the files inside). The user selects an icon by pointing and clicking with the mouse. Figure 1b shows some "pop-up" menus in the Mesa debugger [32]. Debugging commands are partitioned into several submenus. When the debugging menu is requested, a set of submenus appears at the current cursor location. The mouse is used to select the "banner" of a partition to access that submenu, then again to choose an item within that submenu. Tentative selection is indicated by highlighting (in this case, with reverse video as long as the mouse button is depressed). Figure 1c shows a scrollable menu from the Toronto music system [4]. The object named in the small window on the menu is displayed in the adjacent panel. The menu can be scrolled; it can also be switched from temporary to permanent objects.

4.2. Specification of a Complete Interface

In this section we return to the example at the beginning of the paper: an FSM simulator. We discuss the development of the display layout for this application in terms of abstract composition

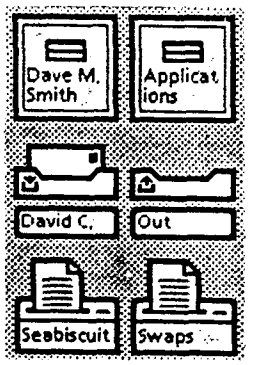

Figure 1a:
Xerox Star Menu

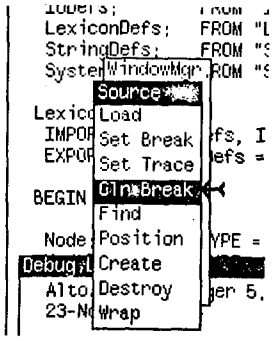

Figure 1b:
Mesa Debugger Popup Menu

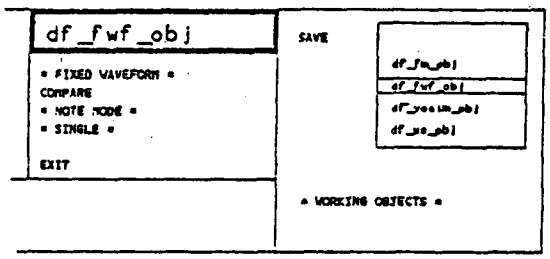

Figure 1c: Music System Editor

rules and of components such as menus. In this example, we are concerned only with format -- the display layout -- and with the level of abstraction appropriate to the task. Specifications for overall style, prosody, and detailed functionality are not addressed here; they should be separable so that the interface designer can focus on specifying the desired viewport layout, leaving details of the layout to the system and dealing separately with functionality and interaction protocols.

Since display interfaces are highly visual, the designer should be able to develop a specification interactively with the help of software tools that illustrate the current layout of the interface at the same time as the formal specification is defined. Although we have only begun to prototype these tools, we can describe how we would expect them to be used for developing a very simple interface for the FSM simulator.

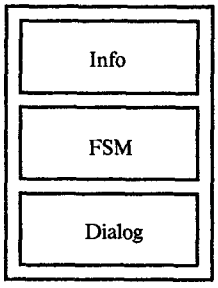

Figure 2a: Initial Layout of Design Sketch

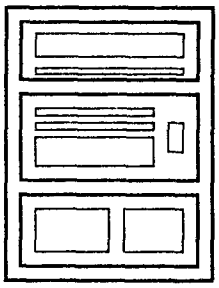

Figure 2b: Addition of More Specification Structure

The interface designer might begin by deciding that the interface will consist of a "frame" that provides fixed information about the system and contains three components: one for system identification, one for displaying the current FSM definition, and the last for handling the user interaction. The designer might represent these decisions as in Figure 2a, by designating the major components and indicating their relative positions. The development tool, in turn, provides policies for dividing the available space among the several components. At this stage, with no information about the contents of the components, it is reasonable to allocate each subcomponent equal space. As the makeup of a component is established, it becomes possible to refine the allocation on the basis of the requirements of individual components.

Each component of this general description might be refined as in Figure 2b. The "Info" component contains constant system identification. The "FSM" component contains input and output tapes, current state, and the machine definition. The "Dialog" component contains simple input and output buffers, the command menu, and a typescript of documentation. Some of the elements of this refinement are associated with character string variables; the portion of the screen allocated to each of these might now be constrained to provide for a single line of text.

Section 4.1 presented a model that leads to a generic definition of menu; one possible incarnation is used here for the command menu. The association of scrolling with a body of information is similarly generic, and the help region can appeal to a predefined schema for scrolling. In both cases, the actions required of the designer should be selecting the appropriate composition rule and associating it with the display region and the corresponding data structure in the application program. An interface constructed along these lines in the current Descartes system is illustrated in Section 6.

During the specification process, the design tool will construct a definition of the interface. The graphical form that is manipulated interactively by the designer is only one representation of that definition. The definition must also be represented in a form that becomes part of the compositor; this might be either tables or code fragments. Further, it is highly desirable to have a static, textual version of the formal specification.

As an example of a suitable form for this formal specification, we can show specifications for the example above. The purpose of this example is to illustrate the structure of a specification. Precise semantics are also essential; both the syntax and the semantics are still under development.

We will specify each element in the form:

<*name*>: <*rule*> **of** <*components*> **with** <*format decisions*>

In this template, <*name*> is an arbitrary (optional) identifier. <*Rule*> indicates that the named icon is produced either directly from program data (e.g., *PgmVar*, *Text*) or by various forms of composition (e.g., *Compose*, *Menu*, *Scroll*) of the named icons. The **of** clause lists the components of the element being defined; these components are all constructed with composition *rules*. The **with** clause provides format information; defaults for many of the attributes controlled by these clauses are pre-specified by an interface style with a *use rule* in a separate style specification. These defaults are established in the form:

for <*rule*> **use** <*format decisions*>

The FSM simulator interface described above can now be formally specified as:

```
Simulator: Compose of Info, FSM, Dialog with Align = Vert
Info: Compose of
        [PgmVar of Logo with Format = StickFigure]
        [PgmVar of Version]
    with Format = Plain, Align = Vert
FSM: Compose of
        [Compose of InTape, OutTape, Mach with Align = Vert],
        State
    with BkGrnd = Grey, Align = Horiz
InTape: Compose of
        [Text of "Input: "],
        [PgmVar of InputTape with BkGrnd = Red]
    with Format = Plain, Align = Horiz
OutTape: Compose of
        [Text of "Output: "],
        [PgmVar of OutputTape with BkGrnd = Red]
    with Format = Plain, Align = Horiz
Mach: PgmVar of FSMDef
State: Compose of
        [Text of "Current State"],
        [PgmVar of CurState with Format = Char, BkGrnd = Red]
    with Align = Vert
Dialog: Compose of Command, Help with Align = Horiz
Command: Compose of
        [PgmVar of Prompt],
        [PgmVar of Response with Prosody = FullLine],
        [Menu of [PgmVar of Options]]
    with Align = Vert
Help: Scroll of [PgmVar of HelpText]
```

All program variables named by the rule *PgmVar* must be supplied by either the compositor or the client. The types of these variables are not at issue, for they are converted to a common intermediate representation before they reach the compositor.

In this example, the composition rules are governed by the following default formatting decisions. Collections of such rules define "styles" and can be provided in a library.

for *Compose* **use** Format = Framed, Font = TimesRoman, AllocPolicy = Fair, BkGrnd = White
for *PgmVar* **use** Format = String, Prosody = NoWrite
for *Menu* **use** Create = Static, Present = All, Select = Mouse
for *Scroll* **use** Heading = "Option Meaning", BarPos = Left, BarUp = Always, Prosody = Mouse

The format language used in this example is a variant of Scribe's environment definition language for text documents [36]. The definition of *Compose* should capture the policy for dividing the available space among the components; in this case the rule reserves space for fixed-sized elements and divides the remainder evenly among variable-sized elements. The *Menu* and *Scroll* composition rules must be defined in terms of primitives and of other abstractions. Format information in the **use** or **with** clauses is inherited by subcomponents unless explicitly overriden; it may qualify the way a composition rule is used for rendering images.

The organization of this formalization is quite similar to the descriptions of interfaces now presented in user manuals (e.g., p. 8 of [3], p. 41 of [5]).

This specification does not address the actual functionality of the program or the protocols for input, field selection, menu selection, or scrolling. Although the range of possibilities for the display layout interacts with decisions about those other properties, we believe that the decisions are largely independent and hence that the specifications should be largely separable.

5. Software Organization Issues

Descartes must support implementation as well as specification of display interfaces. To that end, the organization of an application developed under Descartes reflects the principles of Section 3: strong linkage between client and display, decoupling of application and interface, and separation of policy and instance. In accordance with the first principle, Descartes provides a simple way for a *client application* to make selected variables available for interaction; the interaction is managed by a client-specific module called a *compositor*. In accordance with the second principle, Descartes is organized so that the use of these variables by the compositor is almost transparent to the client. In accordance with the third principle, the Descartes system will provide tools for the interactive graphical development of the specifications for the interface; these tools will make use of a data base with mechanisms for sharing general definitions.

The code in a system with a display interface includes the modules of the client application itself, a compositor developed specifically for the application, and some utility code shared by all Descartes interfaces. The organization of a system with a Descartes interface is illustrated in Figure 3. Note that the display utility is common to all systems; all other modules are specific to one application.

The compositor is responsible for screen layout and for mediating between the client program and the user. It binds the specific decisions about layout, format and prosody made in the interface specification to the internal data structure that represents the display. Since the shape of this internal data structure mirrors the structure of a specification, preliminary results indicate that it may be possible to generate a substantial portion of the compositor code automatically from the interface specification.

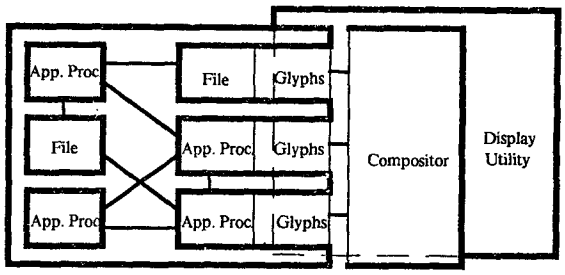

Figure 3: Organization of a system with a Descartes interface.

The shared utilities include a number of largely-independent components that maintain the state of the display and address issues of interaction:

- *Interactive extension of ordinary types:* Both primitive and user-defined types must be extended to produce iconic output and to interpret a user's reference to the displayed icon. Some types are supported by the utility package, but the input-output extensions for newly defined types must be created before the types can be incorporated directly in interfaces.
- *Intermediate representation to support icon construction:* A new data type, *glyph*, is used for intermediate representation of the information that will become a screen icon.
- *Input Handling:* Routines must be provided to translate low-level input events into values of the respective types. Some of this code is type-specific, but other problems are common to all types.
- *Format and prosody specification:* Notations (and run-time representations of those notations) for describing format and prosody are important components of a system; our current implementations are extremely simple.
- *Basic graphics support:* The virtual graphics device provides low-level primitives; it is responsible for actual generation of the display.

The remainder of this section presents the design of the Descartes implementation: Sections 5.1 to 5.4 discuss the shared utilities; Section 5.5 deals with the ways the development tools and the data base help to implement a compositor. The current status of the prototype implementation is discussed in Section 6.

5.1. Interactive Extensions of the User's Types

Two of the principles central to Descartes concern the relationships between an application and its interface. As discussed in section 3, the principle of strong linkage implies that the display should reflect the current state of those variables being displayed, and the principle of decoupling implies that it should be possible to link several different interfaces to the same application program. This section discusses how the implementation supports these objectives.

The use of a display interface should have minimal impact on the client program. In Descartes, each compositor that builds an interface for a client determines which program variables will be displayed (e.g. finite-state machine tape), what the display attributes will be (e.g. color, font, position), and how necessary values will be obtained (e.g. typing, selection). To maintain the strong linkage principle, a compositor associates display components corresponding variables so that every assignment to a displayed variable automatically triggers an appropriate update of the display.

We view this consistency requirement as an invariant constraint on program execution that must be maintained by the system. Descartes supports a class of such invariants between arbitrary program variables. The most interesting use of these invariants is to establish the relation that a display component reflects the current value of its associated variable. This can be achieved by triggering a display update whenever the value of the variable changes.[2] No other restrictions are placed on an application program.

Each data type whose values may be displayed must also provide a set of extensions to support interactive two-dimensional input and output. The compositor uses these extended operations to establish the display invariants connecting variables and display components, to construct images, and to interpret input events. Many of these routines require no special knowledge about the data type and can therefore be generic. Naturally, the routines that construct intermediate representations of icons (i.e., glyphs) and interpret references to the resulting icons require substantial type-specific knowledge and must be constructed individually. These routines may use other utility support such as image construction and ASCII string collection operations. By assigning the responsibility of these routines to the individual types, we can achieve a clean decomposition of the Descartes system.

5.2. Glyphs: Intermediate Representation of Icons

Display images have a rich, composite structure. They may be composed of subimages that have specific relationships both with program data (e.g., displaying a value) and among themselves (e.g., one image is immediately below another). It is not only necessary to build, compose and manipulate these images, but it is also necessary to propagate information about formatting and interpretation decisions around the composite structure.

The data structure chosen to represent the information necessary to exhibit a program's data on a screen is called a *glyph*. The glyph structure is designed to represent structural relations among display components, format and prosody policies for these components, and dynamic visual attributes of the display. In the following paragraphs, we describe the glyph data structure, its important properties, and the operations provided to manipulate it.

In order to achieve the desired flexibility for the specification of display characteristics, a glyph is structured as an inheritance tree.[3] The display attributes (e.g., color, font, position) at any given node in the tree may be left underspecified. Complete values for these attributes are calculated by inheriting values from the node's ancestors and combining them with the partial (or nonexistent) value given at the node. Thus, the compositor may achieve a uniformity of style by specifying that various display characteristics are to be inherited by descendent components.

Stylistic uniformity for interfaces also involves making consistent dynamic changes in the display attributes when the program state changes. These attribute changes are achieved by storing in each glyph node several sets of display attribute values and some indication of which set is currently active. By resetting which value set is active, visual cues may be given to indicate the current application program state. For instance, when a screen button is "pushed" indicating that some command is to be executed, the background color of the region may change. By allowing a compositor to choose how the different regions of an interface react to changes in program state and how display attribute values are inherited, style and policy can be separated from the particular instances of display attribute values.

A value for a display attribute may be specified in several ways with regard to how it interacts with the inheritance mechanism. The important decision determines whether the value is *absolute* (no need for inheritance), *relative* (must be combined in some way with the value from the node's parent), or *absent* (must either be replaced totally with the value of the parent or left completely unspecified). In addition, each attribute must provide the rules by which relative values are combined. The following specification of the function *PropertyOf* defines the basic inheritance mechanism for determining the value of an attribute of a node in the glyph tree:

```
function PropertyOf(G: Glyph, P: Property) returns
  if HasProperty(G, P) then
    if IsRelative(GetProperty(G, P)) then
      if HasParent(G) then
        Combine(PropertyOf(Parent(G), P), GetProperty(G, P))
      else
        Combine(DefaultValue(P), GetProperty(G, P))
    else
      GetProperty(G, P)
  else
    if HasParent(G) then
      PropertyOf(Parent(G), P)
    else
      DefaultValue(P)
```

Of the operations used in the definition, *HasProperty* and *IsRelative* indicate whether a glyph's value for a given property is absent, relative, or absolute; *Combine* describes how to merge relative values of a given type; and *GetProperty* produces the local value associated with a given property for a glyph node. The actual problem is somewhat more complex than the one presented here, and the actual mechanism is correspondingly more compex.

Much of the function of the glyph structure deals with displaying the current state of a program's variables. In particular, we are interested in maintaining the invariant that the screen reflects the current value of displayed variables. Each program variable being displayed must therefore be associated with a glyph node. Since assignment to a variable may trigger an update of the screen, the set of glyphs that display the variable must be explicitly associated with the variable. Thus, every glyph node either has no associated variable or is a member of the set of nodes connected to a variable.

When a new type of component becomes available -- especially one that involves extensive implementation, such as a document preparation system -- it should be easy to incorporate instances of that component as elements of interfaces. The best mechanism for this extension is still an open problem.

The glyph structure also allows a compositor to arrange the screen and interpret users' references to displayed information. An important decision is the policy used to determine what gets displayed when two components share the same screen position. This display policy affects the algorithm used to map between a user-generated position input and a particular node in the glyph structure (see Section 5.3).

So far, the glyph data structure has proven adequate in representing the program data to be shown on a screen, in flexibly describ-

---

[2] In the absence of user-defined assignment, we have had to resort to procedure calls in place of assignment to achieve this.

[3] Although strict hierarchies are usually too restrictive, display interfaces seem to be organized such that every screen region is entirely contained by some other screen region. Therefore, we currently assume that this will be the case for Descartes interfaces.

ing the display attributes of exhibited information, and in simple composition and input tasks. We expect that glyphs will also prove useful in the composition of more complex icons and in the definition of sophisticated input interpretation.

5.3. Input handling and control organization

The other side of constructing two-dimensional interfaces involves furnishing the means to define the complex dialogues that provide input to the application and feedback to the user. Issues include where control over the input process resides, how user input is interpreted, and how the interaction between input and echoing is defined. We present the issues below and describe the mechanisms in the current Descartes system that address them.

A variety of program organizations can be used to coordinate the client application with the interactive user. At one end of the spectrum is the *control-driven* organization used in a data-entry program. The program asks for the values it needs in a particular order and the user is constrained to supply each value as it is requested. At the other end of the scale are *data-driven* programs which wait for any user input and respond to it, as in a screen-oriented text editor. Data-driven programs may be characterized by tables of ⟨*input*, *action*⟩ pairs, whereas control-driven programs are more easily described by the code that implements them.

Most programs, however, are a mixture of the two types. For example, the text editor may ask for a parameter to a command and disallow other input while collecting that parameter. Thus, it is important that systems provide a description mechanism that allows easy blending of the two styles when needed.

Descartes supports a variety of control organizations: the main thread of control can be distributed in various ways between the application proper (for the control-driven style) and the compositor (for the data-driven style). At present, there are no description tools for the data-driven organization.

The second issue, interpretation of user input, can be approached by organizing the input facilities as a set of transducers. Each transducer takes inputs at one level of complexity and produces (usually fewer) outputs at a higher level of complexity. The granularity of these transducers makes it easy to provide libraries of abstractions such as scroll bars and menus.

A low-level transducer might provide a "lexical analysis" facility for user input events. Many interactive programs have conventions that cause a small number of low-level events to be interpreted as a single higher-level event. For example, a mouse button typically provides an indication when it is released as well as when it is depressed. Programs often use this to indicate tentative selection of an option. However, the client using this facility would like to receive only an "option has been picked" indication.

The current facilities for these transducers are quite primitive. We provide a transducer that supports the usual backspace and line deletion operations on string input. The null transducer gives the compositor direct access to the sequence of keystrokes and button clicks from the user. We also provide a transducer for each data type (as part of the type extension) that translates a sequence of low level events into a value of that type. Input formats provide some control over the behaviour of these transducers.

A third problem is that of integrating the input facilities with the output facilities. The input routines must echo the user's input to provide feedback to the user. To do this, they will need access to the same output facilites that Descartes clients use. Also, the glyph tree contains information useful for input event routing among the various transducers.

In the current system, this integration is limited. For instance, when a type transducer is invoked, it is given a screen region associated with a glyph node in which to echo user input. Also, the glyph tree may be used to interpret pointing at a program variable. We do not fully understand how this interaction between input and output will evolve in future versions of Descartes.

We are slowly beginning to understand the issues concerning program control and input interpretation. Our goal is to achieve the same level of understanding that we currently have about the issues dealing with image construction and display.

5.4. Graphics support

A small set of graphics routines form the interface to the physical display device. They allow the drawing of graphical primitives at the level of line, character string, filled polygon, and so on, with a variety of attributes such as color, line style, etc. Input events are provided at the lowest ("rawest") level under the assumption that higher-level transducers will translate them into more abstract tokens or lexemes.

5.5. Achieving "style"

The compositor realizes the specification of the application's interactive user interface by maintaining details of display layout, icon format, and input prosody. It does this on the basis of particular definitions of the *rules* and *format decisions* used in the specification. The vehicle for this administration its the intermediate data type *glyph*.

A set of these definitions constitutes a "style." A style serves two purposes: By providing the definitions of composition rules, a style determines what role a *component* plays in the interface. The definition of *compose* determines how its components are placed in a display; the definitions of *Menu* and *PgmVar* must also say something about the kind of interaction allowed. By providing the definitions of format and prosody decisions, a style determines how a *component* is to fulfill its role. Applied to the root of the *glyph* tree, a style may be inherited by the entire display, enforcing a consistent appearance and consistent prosody on the entire structure. Thus changing the style of the root may effect a major change in the appearance and prosody of the entire display.

Following the model of Scribe [36], we intend to provide a library of styles. The library will consist of a data base together with a mechanism that allows general definitions to be shared. In most cases, we expect that the designer of an application will be able to select a style from this library without modification. In some cases, however, this will not be possible; we intend to provide a means for the designer of an application to modify or extend the library. In particular, we must establish guidelines for adding the format and prosody rules in a notation or representation that can be interpreted by user-defined types. It should be possible for the designer to add new rules, to add new meanings to the rules, or provide new sets of options for formatting and prosody.

At present, the compositor is a hand-written module, typically containing the series of calls to create individual *glyphs*, initialize their state vectors, register their variables and compose them into *glyph* trees. Style is embedded in these calls. We believe the code for most compositors will be highly patterned; if not amenable to automatic generation, our experience suggests that the code can be forged by rote.

6. Prototype Implementation

We are exploring Descartes design and implementation problems by building software support for small prototype domains. There have been two such domains to date. The first was extremely primitive; it was primarily useful for exploring program organization and sharpening our notions about separability of definitions. The second is a bit more ambitious; it incorporates elementary graphics as well as text, and it allows us to work with more realistic examples. This section discusses the two domains and the state of implementation in April 1983.

6.1. First Prototype

For the initial investigation, we selected an extremely restricted problem domain. The purpose of the restrictions was to allow us to concentrate on system structure instead of implementation diversity; we tried to avoid taking unfair advantage of the restricted character of the domain.

For our initial prototype domain we chose 24x80 character "smart" terminals with cursor addressing and a keypad for cursor control; screen configurations based on regular rectangular composition operators with layouts normally bound at definition time; primitive elements including string, integer, float, enumeration and date (a user-defined type); icon formatting including hierarchical inclusion, normal and highlighted depiction; and several varieties of prosody. The software is written in SubAda.[4] It runs on VAXes under Unix with "smart" terminals and on PERQs under POS, where it does not take advantage of the bitmap display, but it does use the Canvas graphics support [2] in anticipation of a larger problem domain.

Four small clients run on VAXes and PERQs. They are organized as described in Section 5: display updating is handled through property lists associated with program variables and interaction is handled in a separate compositor module. The basic utility support has a common interface for VAXes and PERQs. It provides extensions for data types, types for intermediate representations, and a high-level virtual terminal in a form somewhat different from that described in Section 5. The VAX implementation of this support manipulates the terminal directly and the PERQ implementation uses Canvas [2].

Finally, a prototype of a PERQ-based interactive development tool for 24x80 character displays has been constructed. It does not support editing or formal specifications, but it does generate code for setting up complex hierarchical interfaces.

Although this domain is quite restricted, we were able to get substantial experience with certain aspects of the program organization, including modularization and control alternatives, data structures, and prosodic options. This served as the basis for the design described in Section 5 and for the second prototype.

6.2. Second Prototype

The second prototype closely reflects the design presented in section 5. We are concentrating on the design of the type glyph, on the representation of interactive variables, and on the mechanisms that support specifying and building composite display structures. The type glyph is central to the prototype; we have simplified the association between interactive variables and glyphs and have experimented with the construction of composite components such as menus. To date, output format and input prosody descriptions remain very primitive; we have incorporated enough to test the apparatus, but we have yet to tackle the full problem.

This prototype runs on PERQs; it is implemented in PERQ Pascal. Since the experience with the first prototype provided experience with device-independence, we have in this case concentrated on the use of the PERQ bitmap display rather than on independence from the display capabilities, though we continue to avoid gratuitous dependence on the device.

We have implemented the data type glyph, a generic interactive type support procedure, with specific support for a few common types, and an interface (to Canvas) that supports the graphic concepts imbedded in glyph. As an example of the kinds of interfaces we are able to support, Figure 4 shows an interface constructed for the FSM simulator that we discussed in Sections 1 and 4.2. The simulator was converted directly from the original typescript interface and the conversion is not complete (some operations that are not yet available are stricken out on the display). Importantly, the client application has not, at this point, been changed from the typescript version, though to complete the conversion, minor changes will be necessary to undo some heavy dependency on typescripts that was built into the client.

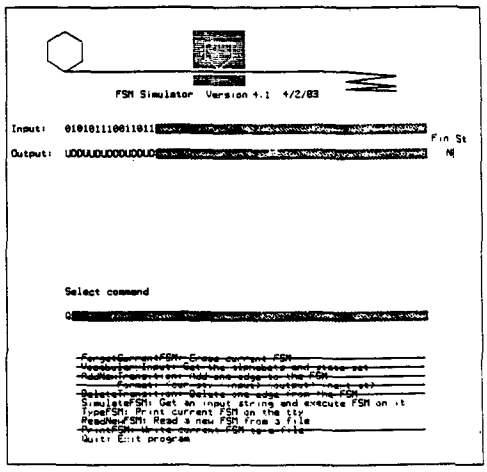

Figure 4: Sample Descartes Interface for FSM Simulator

7. Impact of Programming Language Ideas ...

The Descartes project is investigating certain issues of programming methodology in the context of design for interactive display interfaces. Our emphasis has been on the ideas about abstraction, specification, and separation of concerns that have arisen in programming language design. We summarize by noting the direct effect some of these ideas have had on the Descartes design.

7.1. ... on Interface Design

Programming languages are often designed in order to codify and regularize some aspect of programming methodology. A good design does so without unduly restricting the programmer's design alternatives. Similarly, we are codifying some strategies for

[4] SubAda is an Ada-like extension of Pascal; it would be an Ada subset if Ada had subsets.

designing and implementing display interfaces. Our objective is a unifying framework, not merely software tools. Support for the resulting methodology will include canonical program organizations, specification techniques, development tools, and runtime software utilities.

It does not seem appropriate for us to design a programming language *per se*, but many of the strategies of language design are useful. We must, for example, specify dynamic processes of input and output -- and of their interaction, as when an input is transmitted by selecting some portion of an image on the screen. Indeed, the most important distinctions between abstract data types and display interfaces arise from the introduction of two-dimensional, dynamic interaction.

Programming language design often emphasizes "separation of concerns." Such separations occur in Descartes in several ways. The most obvious is the independence of the specifications of output format and of input prosody. Our software organization encourages the separability of the display-control module from the client program. Moreover, we recognize several useful strategies or general styles for organizing a display interface, so it is not adequate to provide support for a single strategy. We therefore maintain a strong separation between general policy decisions and specific instances of interfaces; this is comparable to the separation between a document and a document style definition in the Scribe document formatting system [36].

Much of the recent emphasis in programming methodology has been on specification techniques and on the separation of abstract properties from particular realizations. On the other hand, the emphasis in the area of user interfaces has been on construction of particular classes of interfaces and evaluation of the results. Descartes shows the influence of the programming-language ideas in the view of an interface as a restricted language and its emphasis on abstract specification of interface components.

Any user interface implements a language. Although the interaction is highly dynamic, its syntax obeys certain rules. Similarly, the various syntactic units have semantics that correspond either to operations by the client program or to manipulation of the interface itself. It is not surprising, then, that language design principles such as simplicity, regularity, and well-definedness apply to display interfaces as well as to general-purpose programming languages.

In Descartes, we capture the syntactic rules in definitions of prosody. The composition rules carry much of the burden of the semantics of the interface proper; naturally the client program supplies the substantial semantics for the application. By collecting consistent sets of definitions in "styles," we support the development of simple, uniform interfaces without forcing all designers to adopt a single style. The use of generic composition rules helps to assure well-defined interfaces by providing design tools that cope with many low-level details. The language influence is also clear in the way various sets of concerns have been separated: policy from instance, definition from use, functionality from interface design, and interface code from client program code.

The influence of the abstract data type is also clear. We rely heavily on the linguistic technique of identifying abstract constructs appropriate to the problem domain and defining them in terms of primitives and other composite definitions. For example, we combine the concept of a menu with that of a scrollable region of information (both non-primitive constructs in their own rights) to form a menu with a scrollable set of selections. The design of type glyph itself and the uniform extensions of all types for interaction also show the influence of abstract data types.

7.2. ... on System Organization

The Descartes software shows the influence of abstract data types, both in its philosophy and in the implementation of particular system elements. Several of the specific techniques of abstract data types are used heavily, including a standard program organization, interface specifications, and invariant assertions. In addition, several system components are designed as abstract types.

Our emphasis is not on building subroutine libraries but on a prototypical program organization that can serve as a template for producing systems. We are exploring a particular modularization in which icon generation is handled by types in the client program and the combination of those icons (or, more precisely, of their intermediate representations as *glyphs*) into larger images is handled by a stylized module, the compositor. Similarly, prosodic interaction is handled largely by the compositor, but type-specific interpretation of the result is the responsibility of the client's types. By "stylized modules" we mean that we expect most compositors to be variations on the same organizational theme, just as abstract data type implementations are variations on the same basic theme.

We are also concerned about specification of interfaces and with verification that implementations satisfy the specifications. Preliminary results suggest that we may be able to simplify verification by taking advantage of the restricted domain and generating large segments of the interface modules directly from the specifications. In addition, we are exploring graphical techniques for developing the specifications.

Another language concept, the use of invariants, is central to the system organization. The relation of the displayed image to the program variables is defined by invariant assertions. Descartes uses property lists associated with displayed variables to maintain these invariants automatically instead of forcing the programmer to be explicitly concerned with display updating.

Acknowledgements

The ideas reported here have emerged from many fruitful discussions with our colleagues, especially those at Carnegie-Mellon, Xerox PARC, and meetings of the DARPA Quality Software Working Group. In particular, constructive suggestions from Bill Wulf, Phil Wadler, Allen Newell, Cynthia Hibbard, Phil Hayes, Marc Donner, Richard Cohn, and Jon Bentley helped improve this manuscript. We have also been influenced by numerous interactive display interfaces for which we have been unable to obtain citable documentation. This research was supported by the National Science Foundation under Grant MCS80-11409.

References

1. Ed Anson. "The Device Model of Interaction." *ACM Computer Graphics 16*, 3 (July 1982), 107-114.

2. J. Eugene Ball. Canvas: the Spice graphics package. Tech. Rept. Spice Document S108, Carnegie-Mellon University, Department of Computer Science, August, 1981.

3. Douglas K. Brotz. *Laurel Manual.* XEROX PARC, 1981.

4. W. Buxton, S. Patel, W. Reeves and R. Baecker. "OBJED" and the Design of Timbral Resources. Proceedings of International Conference on Computers and Music, 1980, pp. 1-12.

5. William Buxton. *Music Software User's Manual.* Computer Systems Research Group - University of Toronto, Toronto, Ontario, Canada, 1981. C.S.R.G. Technical Note 22.

6. Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, N.J., 1982.

7. G. Curry, L. Baer, D. Lipkie, B. Lee. "Traits: An Approach to Multiple-Inheritance Subclassing." *ACM SIGOA Newsletter 3*, 1&2 (June 1982), 1-9. Proceedings of Conference on Office Information Systems

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

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

10. Steven Feiner, Sandor Nagy and Andries van Dam. An Integrated System for Creating and Presenting Complex Computer-Based Documents. Computer Graphics, ACM, August, 1981, pp. 181-189.

11. J.D. Foley and A. VanDam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

12. Michael T. Garrett, and James D. Foley. "Graphics Programming Using a Database System with Dependency Declarations." *ACM Transactions on Graphics 1*, 2 (April 1982), 109-128.

13. Adele Goldberg and David Robson. A Metaphor for User Interface Design. Proceedings of 12th Hawaii International Conference on System Sciences, Conference on System Sciences, 1979, pp. 148-157.

14. Gordon C. Graham. "Display Station's User Interface Is Designed For Increased Productivity." *Hewlett-Packard Journal 32*, 3 (March 1981), 8-12.

15. Graphics Standards Committee. "Status Report of the Graphics Standards Committee." *ACM Computer Graphics 13*, 3 (August 1979).

16. John V. Guttag, Ellis Horowitz and David R. Musser. The Design of Data Type Specifications. In *Current Trends in Programming Methodology*, Prentice-Hall, 1978, pp. 60-79.

17. 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.

18. J.V. Guttag and J.J. Horning. An Introduction to the Larch Shared Language. MIT Laboratory for Computer Science, 1983.

19. 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.

20. *VIEW/3000 Reference Manual.* 32209-90001 edition, Hewlett-Packard Co., 1979.

21. Peter Hibbard. Document Preparation Facilities For Spice. Tech. Rept. Spice Document S143, Carnegie-Mellon University Comp Sci Dept., November, 1982.

22. R. S. Hirsch. "Procedures of the Human Factors Center at San Jose." *IBM Systems Journal 20*, 2 (1981), 123-171.

23. C.A.R. Hoare. "Proof of Correctness of Data Representations." *Acta Informatica 1*, 4 (1972).

24. David J. Kasik. "A User Interface Management System." *ACM Computer Graphics 16*, 3 (July 1982), 99-106.

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

26. B.W. Lampson, J.J. Horning, R.L. London, J.G.Mitchell and G.J. Popek. "Report on the Programming Language Euclid." *ACM SIGPLAN Notices 12*, 2 (February 1977).

27. Keith A. Lantz and Richard F. Rashid. Virtual Terminal Management in a Multiple Process Environment. Proceedings of the Seventh Symposium on Operating Systems Principles, ACM, December, 1979, pp. 86-95.

28. Daniel E. Lipkie, Steven R. Evans, John K. Newlin, and Robert L. Weissman. "Star Graphics: An Object-Oriented Implementation." *ACM Computer Graphics 16*, 3 (July 1982), 115-124.

29. Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, and Alan Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

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

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

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

33. William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

34. David L. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM 15*, 12 (December 1972).

35. Harvey L. Poppel. "Who needs the office of the future?" *Harvard Business Journal 60*, 6 (November-December 1982), 146-155.

36. Brian K. Reid. *Scribe: A Document Specification Language and its Compiler.* Ph.D. Th., Computer Science Department, Carnegie-Mellon University, October 1980.

37. 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.

38. G. Robertson, D. McCracken and A. Newell. The ZOG Approach to Man-Machine Communication. Tech. Rept. CMU-CS-79-148, Carnegie-Mellon University, October, 1979.

39. 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.

40. L.A. Rowe and K.A. Shoens. A Form Application Development System. Manuscript dated December 1981, received by private communication February 1982

41. J. Seybold. Xerox's 'Star'. In *The Seybold Report*, Seybold Publications, Media, Pennsylvania, 1981.

42. Mary Shaw. "The Impact of Abstraction Concerns on Modern Programming Languages." *Proceedings of the IEEE 68*, 9 (September 1980), 1119-1130.

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

44. Jimmy A. Sutton and Ralph H. Sprague, Jr. A Study of Display Generation and Management in Interactive Business Applications. Tech. Rept. RJ 2392 (#31804), IBM San Jose Research Laboratory, November, 1978.

45. W. Teitelman. A Display-Oriented Programmer's Assistant. Proceedings 5th International Joint Conference on Artificial Intelligence, 1977, pp. 905-915.

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

47. 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.

48. D. L. Weller, E. D. Carlson, G. M. Giddings, F. P. Palermo, R. Williams and S. N. Zilles. "Software Architecture for Graphical Interaction." *IBM Systems Journal 19*, 3 (1980), 314-330.

49. Peter C.S. Wong and Eric R. Reid. "Flair -- User Interface Dialogue Design Tool." *ACM Computer Graphics 16*, 3 (July 1982), 87-98.