INCENSE: A SYSTEM FOR DISPLAYING DATA STRUCTURES

Brad A. Myers*
Xerox Palo Alto Research Center, California

ABSTRACT

Many modern computer languages allow the programmer to define and use a variety of data types. Few programming systems, however, allow the programmer similar flexibility when displaying the data structures for debugging, monitoring and documenting programs. *Incense* is a working prototype system that allows the programmer to interactively investigate data structures in actual programs. The desired displays can be specified by the programmer or a default can be used. The default displays provided by Incense present the standard form for literals of the basic types, the actual names for scalar types, stacked boxes for records and arrays, and curved lines with arrowheads for pointers. In addition to displaying data structures, Incense also allows the user to select, move, erase and redimension the resulting displays. These interactions are provided in a uniform, natural manner using a pointing device (*mouse*) and keyboard.

CR Categories and Subject Descriptors: D.2.2 **[Software Engineering]**: Tools and Techniques - Structured Programming; User Interfaces; D.2.5 **[Software Engineering]**: Testing and Debugging - Debugging Aids; Monitors; D.3.3 **[Programming Languages]**: Language Constructs - Abstract Data Types; E.1 **[Data Structures]**; I.3.4 **[Computer Graphics]**: Graphics Utilities; I.3.6 **[Computer Graphics]**: Methodology and Techniques Interaction Techniques.

General Terms: Design, Human Factors, Languages.

*Author's current address: Three Rivers Computer Corporation, 720 Gross Street, Pittsburgh, Pennsylvania, 15224

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0-89791-109-1/83/007/0115 $00.75

I. Introduction

Many modern computer languages allow the programmer to define and use different data types. Few programming systems, however, allow the programmer similar flexibility when displaying these data structures in debugging, monitoring and documenting programs. *Incense*, a system written in and for the Pascal-like language *Mesa* [19], allows the programmer to design and use pictorial representations for the display of data structures.[1]

Strongly typed languages, such as Pascal [14] and Mesa, allow the programmer to define new types using the basic types supplied by the language. These new types can then be used to declare variables. Other languages, such as CLU [18] and Smalltalk [22], have the definition of types and their operations as the central programming paradigm. A data display system like Incense would be a great asset to programmers using a language of either type.

Modern Computer Science theory promotes the value of information hiding, where only the relevant information at a particular level should (or can) be accessed at that level. Most modern languages promote this method of programming. When debugging, however, the programmer is typically restricted to viewing the data at a very basic level. Just as the user of a real number does not want to have to examine the bit patterns used to represent the number, the user of a Ring Buffer or Hash Table generally does not want to see the records, arrays, and pointers used to implement them. Incense therefore allows the programmer to define the display for any type and have that display used whenever data of that type is shown. Since a programmer may not want to define displays for all (or possibly any) of the types used in his program, an extensive set of default displays is also provided.

[1]Incense was designed and implemented at the Xerox Palo Alto Research Center by the author. A full description can be found in [20].

While the importance and utility of user-defined displays is clearly evident for any computer environment, the ability to create genuine pictures enhances Incense's effectiveness considerably. The importance of pictures can be understood by looking at a typical programmer's bulletin board or note pad. When debugging a program or explaining it to another person. the programmer is likely to draw a plethora of pictures representing typical cases or those under consideration. These pictures allow the situation to be more easily understood. It is clear, for example, that Figure 1 is easier to understand than Figure 2, and that the programmer is more likely to visualize and draw the former. A guiding principle of Incense is to automatically create displays that would be similar to those the programmer might have drawn on paper.

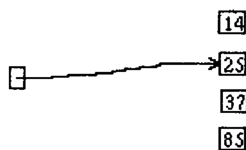


Figure 1.
Pictorial representation of a POINTER TO INTEGER[2].

```
                @1275:  14
                @1276:  25
p: 1276↑        @1277:  37
                @1278:  85
```

Figure 2.
Typical display for a POINTER TO INTEGER in a character oriented system.

In some cases an even more pictorial representation than Figure 1 may be useful. For example, bar graphs, icons, tables and *analogical* pictures[3] are much more evocative than the numbers on which they are based. Frequently, these pictures will provide the right amount of detail and can be understood much more quickly. For example, a *percent-done thermometer* (Figure 3), as a representation for an iteration variable, allows instant recognition of how much of the loop has been completed.

With the progress in computer hardware, creating these pictures for debugging has become increasingly economical. Many personal computers, such as the Xerox Alto [24] and the Three Rivers PERQ [21], come with powerful graphic displays that can be easily used by all programs to make dynamic pictures.

[2]This figure and all others in this paper except 2, 3, 4 and 18 were created by Incense and taken directly from the screen.

[3]Pictures understood by *analogy* with the physical world.

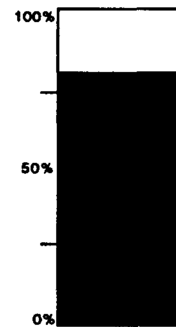


Figure 3.
"Percent-done thermometer" as an analogical representation for an iteration which is 80% complete.

The desired situation would be for the computer to create or verify programs automatically and thereby eliminate the need for debugging and for the displays of data structures. Dijkstra claims that good programmers could avoid all debugging if they would only use structured programming techniques [6]. Debugging, however, is still very much a problem for programmers. In fact, van Tassel claims that "a bug-free program is an abstract theoretical concept" [25, p. 117]. The programmer currently spends a great portion of his debugging time trying to understand the state of the machine. If the computer automatically produced pictures of the data structures, the programmer would be freed of one of the more laborious parts of the debugging process.

Thus, better tools for debugging are needed. Since the necessary support hardware is now generally available, the time is ripe for a development of systems that support pictorial, user-defined displays of data structures such as provided by Incense.

II. Previous work.

The utility of graphical representations for program concepts has long been known. Flow charts were an early attempt at representing computer information graphically. In addition, it was felt that they should be produced automatically by a computer since the hand made flowcharts were expensive and slow to produce, inaccurate when drawn, and hard to maintain [16][10].

A number of systems have been built which can create pictures for a program after the execution has finished. Usually, code must be added to the source program to define the output and then the program must be run through some sort of pre- or post-processor (examples are [4][26][3]). Some excellent algorithm demonstration movies have been produced using this technique [3][13], frequently at a fairly high cost. Other systems have produced high quality dynamic real-time displays for a limited set of types. For example, there have been a number of tree display programs (e.g. [23]) and LISP list displays [7][17], but these are typically restricted to a single, simple type of value at each node. Knowlton [15] describes a system for L-6 that displays the language's arbitrary field definitions.

III. The Environment for Incense.

Incense was written on the Alto personal computer at Xerox PARC. The Alto has a 875 by 603 pixel raster scan display and a "mouse" pointing device with three buttons. A cursor on the screen follows the mouse's movements. Incense was written for the Mesa language. The compiler for Mesa produces extensive symbol tables for each program that provide full type information for all constants and variables.

IV. The Incense system.

Unlike the systems described above, Incense can automatically generate pictures for *any* data type during the execution of an actual Mesa program. This is done in real time without modifications to the source program. The user need only specify at debug time the string name of a variable to get a full pictorial display. If desired, new, pictorial displays can easily be created and associated with data structures. These user-defined displays can eliminate unnecessary detail or more graphically depict the abstraction that the data structure is implementing. The pictures are then displayed in real time while the program is running. Monitoring ("animation") or static visualization of program execution at a high conceptual level is therefore possible. All input from the user is supplied in a similar high level by using the "mouse" pointing device to draw rectangles or select displays.

A unique feature of Incense is the ability to provide multiple displays, called *Formats*, for a single data structure. The user might provide both iconic and detailed representations so that detail can be eliminated when not needed but still be available to implementers. For each Format, the user can supply a number of *Subformats*, one of which is chosen automatically by Incense based on the amount of screen space given for the particular display.

V. Design Considerations for Incense.

In designing a system to implement the ideas mentioned above, two major problems had to be solved. First, an appropriate structure had to be designed that would be sufficiently powerful to allow very complex displays to be created and used. It was important, however, that this design make it easy to define and use simple pictures. Second, a design for the display of pointers had to be formulated. Pointers are a special problem since the space used to display the referent has to be specified. For records and arrays, the subpieces all fit neatly inside the box for the aggregate, but for pointers, the referent is displayed outside of the pointer's box.

As part of the solution to the first problem, *Artists* are used, as discussed below. *Layouts* were invented to solve the problems with pointers (discussed in section 7). A run-time type system, described in [20], allows Incense to discover the types of all variables in programs using the Mesa compiler's symbol tables. This makes it possible for Incense to automatically display any structure given only the name of the variable that holds it.

In order to have an element of data displayed in Incense, an *Artist* must be associated with it. An Artist is a collection of procedures and data that handles the display, erasure, and modification of the data. An Artist is created for each aggregate structure (such as a record) and for each data item, even if the data item is a field of an aggregate structure. The association between the data and the Artists to display them is maintained automatically by Incense. The memory address and type of the data, as well as other information, are stored as part of the Artist's internal data. Five classes of procedures are also stored in each Artist. There are procedures for displaying the data, drawing arrows, erasing pictures and arrows, selecting Artists, and editing the associated data structures. Each of these will be discussed below in a separate section.

VI. Display.

Although pictorial representations are easier to understand, they are not always appropriate. For example, the user of a Ring Buffer may want only to see the display of Figure 4, but the implementer may need to see the underlying basic types. Incense supports multiple views of the same data through the use of *Formats*. Each Artist has one or more Formats, each of which displays the data in a different manner. For example, an Artist for a Time record (Figure 5) might contain Formats for displaying itself as a record (a) or a clock (b). Formats are intended to be used for radically different ways of viewing a data structure. The implementer of the Artist defines the Formats, but the user specifies interactively which Format to use in a particular display. If an implementer wants to protect the internal representation of a data structure from the client, he can supply an Artist that only has a Format to display an iconic representation.

The caller of an Artist Format procedure specifies the area in which the display must fit. Thus the user[4] always specifies the size and position of the display, and Artists must be prepared to fit into any size rectangle. Other data display systems have used very different strategies. For example, in AMBIT/G, the displayed objects always take a constant amount of space [5], and in Smalltalk, the objects themselves decide how much space to take [9]. The advantage of the Incense approach is that the user's choice is never pre-empted, and aggregate structures, such as records and arrays, can accurately specify the position and size of subparts.

---

[4] In the case of nested Artists used by aggregates such as records and arrays, this is done by a procedure, as explained below.

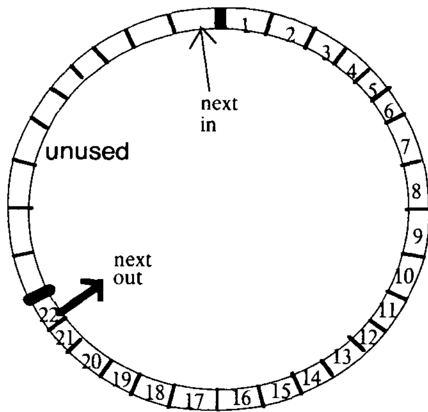


Figure 4.
Possible display for a ring buffer.
(This figure was not created by Incense).

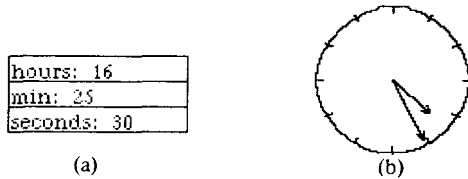
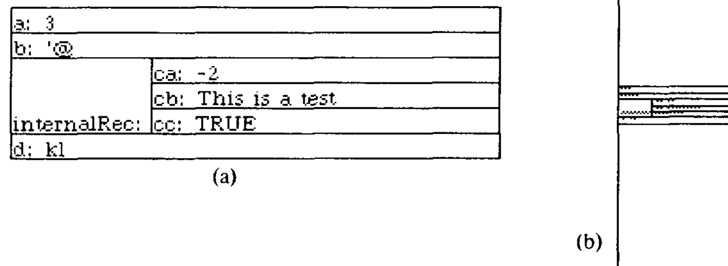


Figure 5.
Two Formats for the record Artist for:
Time: RECORD [hours, min, seconds: INTEGER];
holding the time of day:
(a) as a normal record and (b) as an analog clock.

To increase the flexibility of the displays and allow all control to be procedural, each Format contains one or more *Subformats.* Once the user specifies a Format, the system then chooses one of that Format's Subformats. This decision is based on various contextual information such as the size of the area in which the display is to fit. For example, the standard Artist Format for a Mesa record has two Subformats (Figure 6). The first displays the data at full size (a). If insufficient room for the display is provided, however, the second Subformat would be invoked. It scales the record display so that it will retain the same proportions as the original, yet still fit in the box (b). The client is not allowed to specify which Subformat should be used. Instead, the creator of the Artist associates a test with each Subformat. This test determines whether the Subformat is applicable in the current context. Since more than one of these tests may succeed, the designer also specifies an ordering of the Subformats. If none of the Subformat tests succeed, the data is simply not displayed. This may happen, for example, if the area in which the data is to be displayed is very tiny.

The basic types of Mesa, including STRINGS, INTEGERS, CARDINALS (positive INTEGERS), BOOLEANS, CHARACTERS, REALS, PROCEDURES, UNSPECIFIEDS, WORDS, and Enumerated Types (lists of names, *e.g.* {Mon, Tues, Wed, Thurs, Fri}), all have default displays which use text strings of the same form as the literals used by the programmer. These are fully interpreted as can be seen in Figure 7. Mesa uses a special representation for



(b)

Figure 6.
Two Subformats for a record: full size (a),
and scaled proportionally and centered vertically inside a bounding rectangle (b).

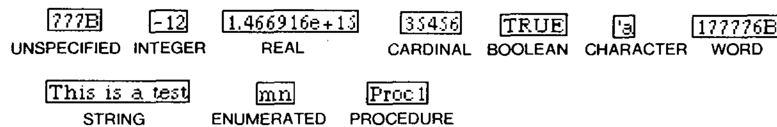


Figure 7.
Default boxed display for the basic types.

subranges of the above types. Incense hides this transformation so the programmer always sees the values he would have typed into the program. There are two Formats for Artists of each of the basic types. The first draws a box around the value (as in Figure 7). This is useful when the value is standing alone or when there is a pointer to it (see Figure 1). The other Format does not draw a box around the value. This is used in aggregate structures such as records and arrays when boxes are drawn around the entire field. For each Format of each of the basic types, there are two Subformats. One displays the value as a text string. This string is allowed to be clipped slightly (Figure 8 (a) and (b)). If there is not enough room for a meaningful portion of the value to be displayed, the second Subformat will be used which displays a grey area which has a size proportional to the string that would have been displayed (Figure 8 (c) and (d)).

Figure 8.
Demonstration that clipped strings do supply information: (a) and (b) are values of BOOLEANS. Grey areas have different lengths proportional to the size of the string: (c) is for FALSE and (d) is for TRUE.

The Subformats of some Artists, such as those for records, arrays, and pointers, can cause the display of subordinate Artists. For example, the automatically supplied Artist Subformat for a record will iterate through the fields calling the appropriate Format in the Artist for each. The record Subformat divides the rectangle allocated for the display of the record among the fields giving an appropriate portion to each. Thus, the subordinate Artists are called with the same types of arguments as the top level Artist, and the record Subformat need not know, for example, whether the subordinate is an integer, pointer, or even another record (see Figure 9).

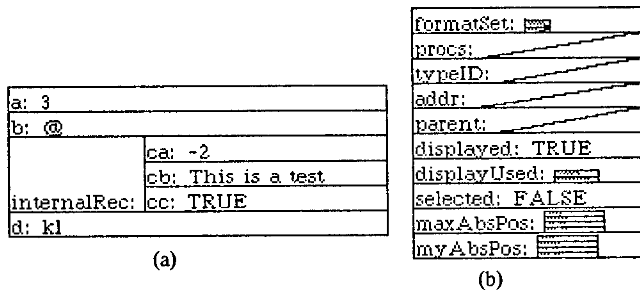


Figure 9.
Two ways records can contain other records. Full size (a) and reduced (b). The diagonal lines in (b) represent the pointer value NIL.

Each record field actually consists of two Artists: one for the field's *name* and one for the field's *value*. The field's name and the value's type are discovered automatically using the information from the Mesa symbol tables [20]. The field name is displayed using an Artist to make the interface more consistent. This has the additional advantage that standard STRING Artists can be used for the names. The field names therefore become grey if the area is too small and can be selected and redisplayed in the same manner as all other values (see section 9). It has been suggested that the field names should have special Formats that center the name vertically in the area provided, or display it at the top of the field's value. This would be a trivial modification and could, in fact, be done by the user.

Arrays are handled in exactly the same manner as records, except, of course, that there are no field names. Special Artists could be created to provide the indices of the values and/or to allow scrolling so that large arrays could be easily handled. Both record and array Artists store, as part of their internal data, the set of rectangles used to specify the fields' positions. These rectangles are defined at Artist creation time based on the types of the values, and the length of the field name strings for records. Making arrays display vertically or horizontally is therefore accomplished simply by defining the rectangles with the appropriate offsets (Figure 10). Thus two-dimensional arrays are easily displayed (Figure 11).

The rectangles and other static information stored in the Artists are called *Form Data* since they describe the *form* of the picture. Other information stored in an Artist, called *Display Data*, is generated when the Artist's associated data structure is displayed. The Display Data includes such things as the current screen position of the the picture and whether or not it is selected.

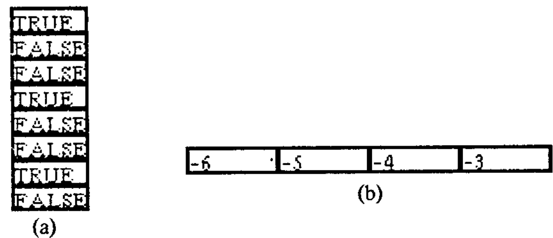


Figure 10.
Arrays oriented vertically (a) and horizontally (b).

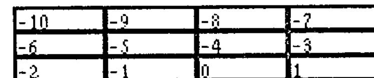


Figure 11.
Two dimensional array:
ARRAY [1..3] OF ARRAY [1..4] OF INTEGER;

VII. Displaying Arrows.

One of the most complex aspects of the Incense design was choosing the best method for displaying pointers. It was clear that they should be shown as arrows, but the difficulty was deciding where to put the data to which the pointer points (the *referent*). The simplest technique would have been to require the user to specify where each referent should go. For deep structures, however, this would be very tedious. The simplest automatic placement scheme was used by AMBIT/G where the referent simply appeared at a predetermined, fixed position relative to the parent irrespective of what might have been there previously [11]. This method is not consistent with Incense's philosophy of hierarchical rectangles and user control. The most complex technique would be to treat the screen space like a heap memory and simply allocate and free rectangles of display area. This has the potential for maximal usage of the space. Unfortunately, this two-dimensional space allocation problem is very complicated and prone to poor performance. The specialized space allocation techniques used in tree and list drawing programs are also unworkable since they assume that all nodes will be the same size. This is certainly not necessarily the case in Incense.

The algorithm finally chosen for Incense is very general and fast, but it does not allocate the screen space as well as the general allocator described above. *Layouts* were invented to hold the pointer and its referents. When the user specifies a rectangle for the display of a Layout, that rectangle is subdivided into rectangles to be used by the arrow sources and all referents. Each component rectangle is managed by a *Layout Field*. For example, a Layout for a record containing two pointers would have three fields: one for the record and one for each referent (see Figure 12). Layouts and Layout Fields are implemented as special Artists that have no associated data; they exist solely to locate and manage the various components. A user of Incense need never know of the existence of Layouts and Layout Fields since they are created and managed automatically by the system. Of course, since Layouts are a special type of Artist, the user is free to create his own type of Layouts if desired.

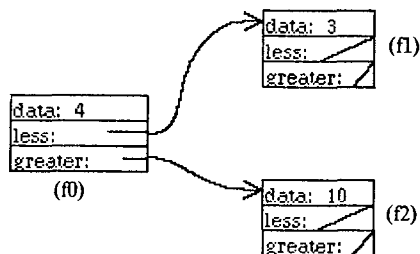


Figure 12.
Layout with 3 fields: one for record: (f0), and one for each referent: (f1) and (f2).

Currently, Layouts use a very simple scheme for assuring that all the subcomponents fit into the rectangle specified for the Layout. As with records and arrays, the rectangles for the various fields (the *Form Data*) are fully specified at Artist creation time. When the Layout is displayed, the subcomponents are simply instructed to fit into the appropriate Layout Field rectangle. In a deep recursive structure such as Figure 13, the displays get progressively smaller as the nesting level increases. This theoretically would allow an arbitrarily deep structure to be displayed in a finite space, but, in fact, display ceases after the pictures become smaller than a threshold size.

When a Artist Subformat for a pointer is called, it first checks to see if the pointer's value is NIL. If so, a diagonal line is drawn through the box for the pointer. This is consistent with the LISP community's usual pictorial representation of NIL. Otherwise, the Subformat checks every Artist on the screen to see if any are associated with the referent. If so, the pointer is referring to data that has already been displayed, and an arrow is simply drawn to this occurrence (Figures 14 and 15). This mechanism also handles the case where a pointer refers to values inside aggregates (Figure 16). If the referent is not currently displayed, the pointer Artist must arrange for the correct Layout Field's Artist to display the referent so an arrow can be drawn to it.

This process can best be explained by using an example. Suppose a record contains a POINTER TO CARDINAL and an integer (Figure 17). When the user requests a display for a record of that type, the system first notices that it contains pointers. A Layout is therefore created with two Layout Fields (Figure 18 gives the Artist hierarchy). The Layout is then ordered to display itself in the rectangle specified by the user. The Layout tells the first Layout Field to display. This Field, in turn, calls the appropriate Format in the record Artist. Now the record begins to display itself in the manner described earlier. When the pointer Artist is finally called, it will discover that its referent (the CARDINAL) has not yet been displayed. The Layout responsible for this pointer must therefore be found since only it knows where the referent should be placed. The pointer Artist will therefore send a message up the Artist hierarchy to find the Layout that will handle its referent. The message will go first to the record Artist, then to Layout Field 1, and finally to the Layout. The Layout, upon receiving this message, causes Layout Field 2 to be displayed which, in turn, causes an Artist for the CARDINAL to be created and displayed. If this had been a more complex structure, such as the recursive structure of Figure 13, more Layouts and Artists would have been created and displayed at this point. An Artist is not created for a data structure if the pointer to it is NIL. This is obviously important for recursive structures since otherwise an infinite number of Artists would be required.

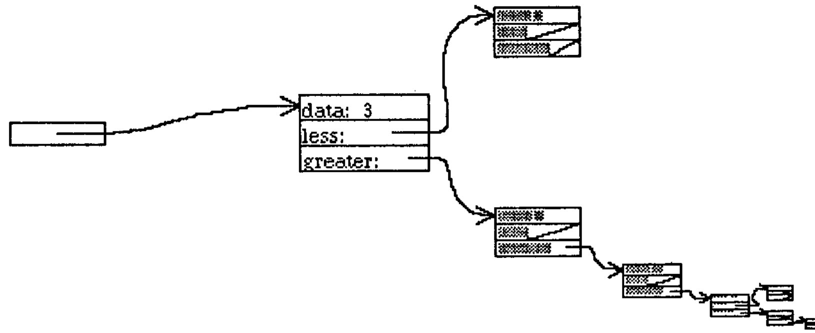


Figure 13. Deep recursive tree display demonstrating how elements get smaller. Overall structure, however, is easily understood.

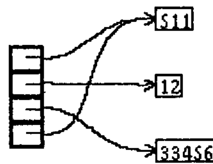


Figure 14.
ARRAY [1..4] OF POINTER with two POINTERs referring to the same value.

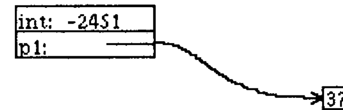


Figure 17.
Incense display for
RECORD [int: INTEGER, p1: POINTER TO CARDINAL].

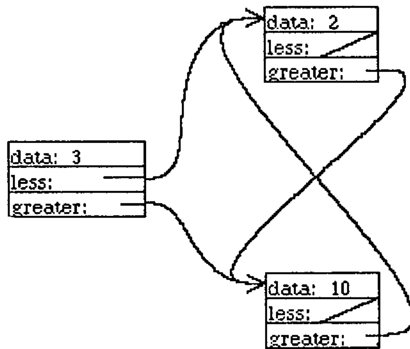


Figure 15.
This erroneous tree structure demonstrates that a pointer to previously displayed object does not generate a new copy. The second arrow is drawn to the first occurrence.

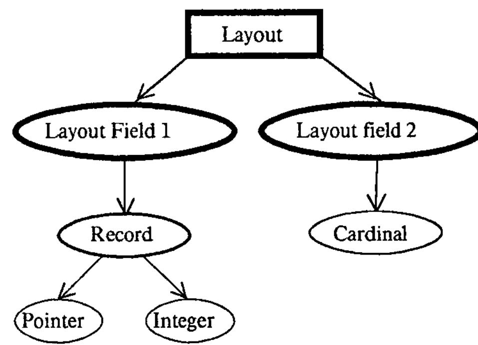


Figure 18.
Artist hierarchy that would be created for:
rec: RECORD [p1: POINTER TO CARDINAL, int: INTEGER];
(This figure was not created by Incense).

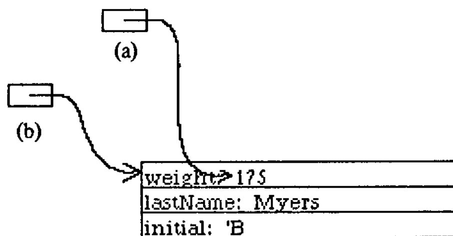


Figure 16.
Pointer to value inside a record (a) does not get confused with a pointer to the record itself (b).

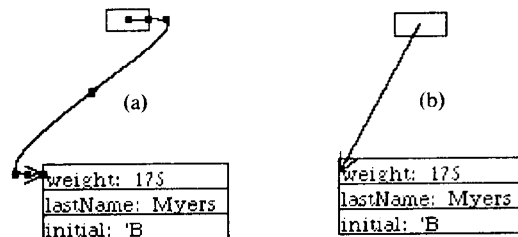


Figure 19.
Demonstration of the advantage of curved lines used in Incense (a) over straight lines (b). The control points used to specify the spline are shown as black squares in (a).

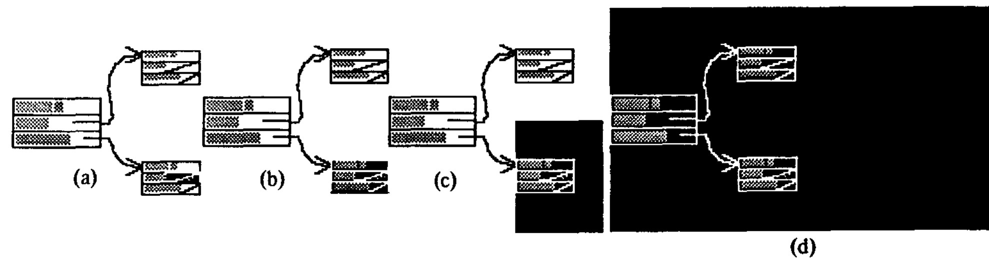


Figure 20.
Selections (shown by black areas) moving up the Artist hierarchy: from record field (a) to record (b) to Layout for record (c) to Layout for everything (d).

Upon completion of the display of Layout Field 2, the Layout returns control to the original pointer Artist with the Layout Artist as the return value. The pointer Artist now sends a different message to the Layout requesting the address of the actual Artist associated with the CARDINAL. This message is passed down the Artist hierarchy from the Layout to Layout Field 2 and finally to the CARDINAL's Artist. If there had been more Layouts, Layout Fields, and records under Layout Field 2, as there would be for recursive structures, this message would have had to pass through more levels. In any case, the POINTER now knows which Artist is associated with its referent and that the referent has been displayed. Finally, the POINTER Artist can find out where the referent is on the screen, and an arrow is drawn from the pointer's box to that of the referent.

The actual arrow itself is a curved line called a *spline* [1]. At the end of the spline, an arrowhead is drawn. The size of the arrowhead is adjusted to insure that it is never bigger than the destination box. The spline is defined by seven control points placed along the arrow's path. Three are placed in a straight horizontal line near the source and three near the destination (Figure 19). Three are needed at each end to insure that the curves approach the end points from the correct direction. Another control point is then placed between the ends to make the spline smoother. Splines were chosen rather than straight lines since they are more attractive and are less easily confused with other lines in the picture (see Figure 19).

The default destination points for arrows to Artists of the basic types are at the center of their left side. For records and arrays, however, the destination points are calculated as the center of the left side of the first field. Since the user is free to re-arrange the rectangles for the fields (see section 11), the arrow end points are stored as part of the Form Data. Thus if the fields are rearranged, the user can simply supply a new set of destination points for the arrow. When the record or array is displayed, the actual screen points for that instance are calculated. This is done before any of the fields are displayed in case a field contains a POINTER back to the record or array itself.

VIII. Erasure.

In addition to display, Artists support a number of other operations. One of these is erasure. An erased Artist[5] can be redisplayed with a new size, location, and Format. There is an erase procedure for each Subformat of an Artist since only the Subformat knows exactly how the data structure was displayed. The Artist's erase procedure therefore calls the internal erase procedure for the Format and Subformat with which the Artist was displayed. This internal erase procedure will first call the erase procedures in any subordinate Artists. Then, the screen area for the Artist will be painted white,[6] and then any arrows are erased by redrawing them with white. Arrows must be erased explicitly since they will lie outside of the rectangle for the POINTER.

IX. Selection.

Incense was designed to be used interactively. Therefore, the user needs to be able to refer to various displays on the screen to allow them to be erased, redisplayed or modified. Many systems require that the user *name* the display either by tracing from some root (*e.g.* First↑.greater↑.lesser↑.greater↑.greater↑.greater↑.value) or by using some arbitrary labels. Fortunately, Incense runs on a computer that provides a much more natural method for referring to displays. The *mouse* is a pointing device that moves a cursor around on the screen [8]. All the rectangles for the display are specified using the mouse. In addition, pressing one of the mouse buttons tells Incense to *select* the Artist to which the cursor points. The selected Artist is shown visually by video reversal (see Figure 20). There can only be one Artist selected at a time; however, due to the hierarchical nature of Artists, there are typically many Artists at any point on the screen. For example, with an integer in a record in a

[5]"Artists" here and in later sections will be used to refer both to the object that controls data display and the picture on the screen that the Artist creates.

[6]The pictures are drawn in black on a white background as shown in this paper.

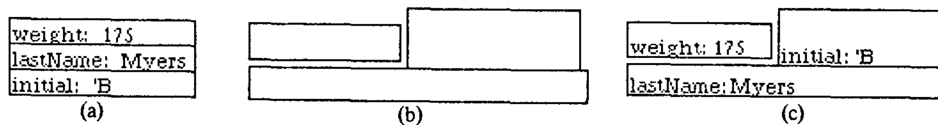


Figure 21.
Normal display for record (a), form defined by user (b), and resulting display (c).
Note that field order has been switched.

Layout, pointing to the integer could refer to the integer, the record, the Layout Field, or the Layout. Incense solves this problem by having the smallest (in screen area) Artist selected. In order to select a larger, enclosing Artist, it is simply necessary to re-select the Artist which is already selected. Thus, for example, a pointer value in a record field might be selected (Figure 20a). If some other field was then selected, the selection would simply move there. If the original pointer was selected again, however, the entire record that contained that pointer would be selected (Figure 20b). Next, the Layout containing the record would be selected (c), and finally, the Layout enclosing everything would be selected (d). If this outermost Layout was selected again, everything would be deselected.

Since Incense frequently displays values using grey areas, selection is very important. The user can display a large structure, select the part of it that he thinks is important, and then have that portion redisplayed using a larger area. Since it is so easy to select and redisplay Artists, the decision to use a simple, fast space allocation algorithm is shown to be justified.

X. Editing.

Once an Artist is selected, its value could be edited.[7] Unfortunately, editing is not yet supported by Incense. In a future version, the user might select the Artist associated with a BOOLEAN variable, type in FALSE, and have the value of the variable change. Type checking of the new values would be done along with the appropriate conversions. For pointers, it would be appropriate to allow the user to simply point to the display for the data that will be the new referent. The system would then discover the correct value for the pointer cell. Type checking would be done in this case also. Different styles for specifying new values other than type-in may also be appropriate for other specialized displays. It is the job of the edit routines in the Artist to handle these transformations, the modification of the actual memory, and the subsequent redisplay of the Artist.

[7] *Editing* is used here solely to mean modifying the *value* of some data and not for modifying its *display*.

XI. User-defined Artists.

The discussion of the previous sections has concentrated on the default displays provided by Incense. These are necessary for making the system usable and for demonstrating its feasibility and power. A major goal of Incense, however, is to make it easy for the user to specify his own Artist styles (called *Prototypes*). This can currently be done at two levels.

As was mentioned in section 6, aggregate Artists store the rectangles used for the pieces as part of the Form Data. It is very easy for the user to specify these rectangles using the mouse (see Figure 21). If desired, the user is asked to point with the mouse to the corners of each rectangle needed for the display. The rectangles define the relative size and position of the various fields. This technique can be used to eliminate some fields (by giving them rectangles of zero size) or to favor the more important fields. It would be useful to have a *Forms Editor* program to help in aligning and positioning the rectangles.

If a more radical change in display is desired (such as the clock display in Figure 5), the user simply writes a small program in Mesa defining the picture. Incense provides a rich variety of display primitives and parameterized procedures to aid in this process. Libraries of Artist Prototypes can be kept so that new Prototypes could be created through small modifications to existing Artists. It is felt that this is a better strategy than requiring the Artist designer to learn an entirely new language to define the desired pictures.

Once a Prototype has been created, the user can then associate it with a specific variable, a type, or a basic type. Thus, for:

```
Time: TYPE = RECORD [hrs, min, sec: INTEGER];
curTime: Time ← [3, 30, 14];
```

a Prototype can be associated with the variable curTime, the type Time, or the basic type RECORD. This gives the user complete control over all displays in Incense.

XII. Current Implementation and Plans for the Future.

The Incense system described here currently runs only on Xerox *Alto* mini-computers that have 128K of memory and special micro-code to handle floating point operations. Incense has successfully demonstrated the feasibility of user-defined and analogical displays for data structures. Default displays currently exist for all the basic types of Mesa, two-dimensional displays are used for records and arrays, and arrows are used for POINTERS. All of these displays are automatically created using the type of the variable, so that no matter how complex a variable is, the user need only type "Display X" and specify a rectangle using the mouse. The display of a basic type takes about 350 milleseconds, for a POINTER TO INTEGER, 1.9 seconds, and for the complex structure of Figure 13, 33.4 seconds. The mouse is used for specifying all rectangles and for selection. The selected Artist can be erased and redisplayed, but editing is not currently supported by Incense. Reasonable defaults are computed automatically for all field rectangles in aggregate structures, and the user is free to define these rectangles using the mouse if desired. The user can also define displays by writing programs in Mesa. Finally, it is easy to convert any display on the screen into a form that can be printed as was done for this paper.

There are some places in Incense where a reimplementation would allow a substantial decrease in execution time and memory usage. Also, certain parts of the system, such as the edit command, have yet to be implemented. The major aspect of Incense that remains unfulfilled, however, is the creation of special purpose Artists for various types. Designs have already been prepared for the displays of trees, list structures, stacks, processes, hash tables, ring buffers (Figure 4), large arrays and some other common data types. Iteration variables (Figure 3) and other special purpose data items could also have special displays. These would greatly enhance the attractiveness of Incense to potential users. Finally, Incense needs to be integrated with a full debugger that can take advantage of Incense's power and flexibility. The debugger would handle the specification of what data should be displayed, possibly by having the user point at the variable in a window holding the source text of the program. The debugger could call Incense frequently to incrementally monitor variables and thereby "animate" the program's execution. If this is not practical, Incense could simply be called to create static displays at breakpoints. The parameters to Incense would be the variable's string name, a symbol table context in which that variable can be found (found based on the run-time stack), and a rectangle for display. The rectangle might be supplied by the user or the debugger might have a standard window for data display. An integrated environment using Incense-style displays has been proposed in [12].

An Incense style display for data structures will be useful to programmers no matter how a debugger is organized. The pictures and the associated data structures can be dynamically rearranged and modified. The displays from Incense-like systems help the programmer monitor and understand the execution of complex programs. Since the user can specify the desired display, the pictures created can be used to provide documentation for the data structures themselves. Finally, debugging will probably be more enjoyable when using pictures rather than long strings of characters. This, combined with the higher conceptual level provided by the pictures, may make the debugging task easier and thereby increase programmer productivity.

Acknowledgments.

I would like to thank Xerox PARC CSL and the MIT Cooperative program for giving me the opportunity to work on this project. Special thanks go to Dan Swinehart and Butler Lampson of PARC for their design and implementation suggestions. John Warnock of PARC supplied the underlying graphic system that made this work possible and also helped with the design for Layouts. Ed Satterthwaite of Xerox SDD provided a great deal of support while I was creating the run-time type system needed by Incense. Finally, I would like to thank Dan Swinehart, Warren Teitelman, Ed Satterthwaite, David Reed, the referees, and many others for helpful and thorough comments on this paper.

REFERENCES

1. Ahlberg, J. H., Nilson, E. N., and Walsh, J. L. *The Theory of Splines and their Appli-cations.* New York: Academic Press (1967).

2. Baecker, Ron. Two Systems which Produce Animated Representations of the Execution of Computer Programs. *ACM SIGCSE Bul-letin.* Vol. 7, No. 1 (Feb. 1975). pp. 158-167.

3. Baecker, Ron. *Sorting Out Sorting.* 16mm color, sound, film. 25 minutes. Dynamic Graphics Project, Computer Systems Research Group, University of Toronto, Toronto, Ontario (1981). Presented at ACM SIGGRAPH Conference, Dallas, Texas (Aug., 1981).

4. Balzer, R. M. EXDAMS -- EXtendable Debugging and Monitoring System. *Proceedings AFIPS Spring Joint Computer Conference.* 34 (1969). pp 567-580.

5. Christensen, Carlos. An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language. *Proceedings of the ACM Symposium on Interactive Systems for Experimental Applied Mathematics,* Washington, D.C. (August, 1967).

6. Dijkstra, Edsger W. The Humble Programmer. *Communications of the ACM.* Vol. 15, No. 10 (Oct 1972). pp 859-866.

7. Dionne, M. S. and Mackworth, A. K. ANTICS: A System for Animating LISP Programs. *Computer Graphics and Image Processing.* Vol. 7 (1978). pp. 105-119.

8. English, W. K., Engelbart, D. C. and Berman, M. L. Display Selection Techniques for Text Manipulation. *IEEE Transactions on Human Factors in Electronics.* Vol. HFE-8, No. 1 (March 1967).

9. Goldberg, A. and Robson, D. A Metaphor for User Interface Design. *Proceedings of the 12th Hawaii International Conference on System Sciences 1979.* Vol. 1 (1979). pp. 148-157.

10. Hain, G. and Hain, K. A general purpose automatic flowcharter. *Proc. Fourth Annual Meeting of UAIDE,* New York (Oct. 1965). pp. IV-i to IV-12.

11. Henderson, D. Austin. *A Description and Definition of Simple AMBIT/G--a Graphical Programming Language..* Wakefield, MA: Massachusetts Computer Associates CA-6904-2811 (April 28, 1969). 32 pages.

12. Herot, Christopher P., Brown, Gretchen P., Carling, Richard T., Friedell, Mark, Kramlich, David, and Baecker, Ronald M. An Integrated Environment for Program Visualization. *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Infor-mation System Design and Development.* New Orleans, LA (January 26-28, 1982). H. J. Schneider and A. I. Wasserman (eds). North Holland, Amsterdam, 1982.

13. Hopgood, F. R. A. Computer Animation Used as a Tool in Teaching Computer Science. *Proceedings of the 1974 IFIP Congress,* Applications Volume. (1974) pp 889-892.

14. Jensen, K. and Wirth, N. *PASCAL User Manual and Report.* Englewood Cliffs, N.J.:Prentice-Hall (1975).

15. Knowlton, K. C. *L6: Bell Telephone Laboratories Low Level Linked List Language.* Two black and white films, sound. Bell Telephone Laboratories, Murray Hill, New Jersey (1966).

16. Knuth, Donald E. Computer Drawn Flowcharts. *Communications of the ACM.* Vol. 6 No. 9 (Sept, 1963). pp. 555-563.

17. Laaser, William. Private conversation with the author (Nov. 9, 1979). System was built using DLISP which is described in Warren Teitelman. *Display Oriented Programmer's Assistant.* Palo Alto: Xerox PARC CSL-77-3 (March 8, 1977).

18. Liskov, Barbara, Snyder, Alan, Atkinson, Russell, and Schaffert, Craig. Abstraction Mechanisms in CLU *Communications of the ACM.* Vol. 20, No. 8 (Aug, 1977). pp. 564-576.

19. Mitchell, James, *et al. Mesa Language Manual, Version 5.0.* Palo Alto: Xerox PARC CSL-79-3 (1979).

20. Myers, Brad A. *Displaying Data Structures for Interactive Debugging.* Palo Alto: Xerox PARC CSL-80-7 (June, 1980). 97 pages.

21. Rosen, Brian. PERQ: A Commercially Available Personal Scientific Computer. *IEEE CompCom Digest* (Spring, 1980).

22. Shoch, John F. An Overview of the Programming Language Smalltalk-72. *ACM Sigplan Notices.* Vol. 14, No. 9 (Sept 1979). pp. 64-73.

23. Sweet, Richard. Appendix B: Implementation De-scription. *Empirical Estimates of Program Entropy.* Palo Alto: Xerox PARC CSL-78-3 (1978). pp. 85-96.

24. Thacker, C. P., McCreight, E. M., Lampson, B. W., Sproull, R, F., and Boggs, D. R. *Alto: A Personal Computer.* Palo Alto; Xerox PARC CSL-79-11 (August 7, 1979). 50 pages. Paper also appears in Siewiorek, Bell and Newell, *Computer Structures: Readings and Examples,* second edition.

25. van Tassel, Dennie. *Program Style, Design, Efficiency, Debugging and Testing.* Englewood Cliffs, NJ: Prentice-Hall, Inc. (1974). 256 pages.

26. Yarwood, Edward. *Toward Program Illustration.* University of Toronto Computer Systems Research Group Technical Report CSRG-84 (M. Sc. Thesis) (October 1977).