PICASSO - An Aid to an End-User Facility

P. G. Sorenson
J. A. Wald
Dept. of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

Abstract

A characterization of an end user and a discussion of an end-user facility are presented. A language for aiding the data-base administrator in designing forms for user's of an end-user facility is described in a tutorial manner. The paper concludes with an outline of work related to the form's approach and other types of user-oriented query language facilities.

Introduction

In this paper we describe PICASSO, a system that has been developed at the University of Saskatchewan to aid the data-base administrator in establishing a proper end-user facility for a data-base management system. Our concept of an end-user facility is consistent with that described in the Progress Report on the Activities of the CODASYL End User Facility Task Group [1976]. In the first section of the paper we elaborate on a characterization of an end user and establish the environment in which such a person is perceived to operate. The second section briefly describes the role of the data-base administrator and identifies the need for a facility such as PICASSO. The third section presents the PICASSO language and illustrates its major concepts with examples. The final two sections set the effort on PICASSO in a perspective with other research being undertaken and outlines future developments in the area of end-user systems.

The End User

Codd[1974], among others, has expressed some concern regarding the lack of a proper interface between a certain class of end user - which he refers to as "casual users" - and the data-base management system with which the user interacts. He characterizes the casual user as follows:

> "The casual user is one whose interaction with the system is irregular in time and not motivated by his job or social role. Such a user cannot be expected to be knowledgeable about computers, programming, logic, or relations. ... The class of casual user is quite broad - it includes almost all of institutional management (private and public) and housewives."

Codd's attempts to rendezvous with the casual user are both valid and commendable. However, not only is the data processing industry neglecting the casual user, it is also ignoring the end user who interfaces daily with the computer in personnel, accounting, payroll, marketing, production, etc. departments throughout the business community. Let us seek a clearer characterization of this type of end user.

The End User Facility Task Group (henceforth referred to as EUFTG)[1976] has expended a great deal of effort attempting to classify end users. A summary of their classification is as follows:

a) The end user is a person engaged in a job that directs or supports the direction of the following activities: planning, management, execution,monitoring, or evaluation.

b) An end user has an established need to collect and manipulate data that are directly related to his/her job requirements and experiences.

c) An end user is competent in his/her job but is not necessarily a data processing expert and should not expect to become proficient in programming or any other data processing speciality.

d) The end user must be willing to understand that his/her view of the data is defined and that there is a finite set of operations he/she may invoke.

To satisfy the needs of the end user, the EUFTG selected a form-oriented approach, an approach which they believe allows different users of a data base to have their own logical perception of the data. Certainly there is ample justification for this approach since forms have been and are commonly used in a business environment for person to person communication(Silas[1976]). Obciously we concur with this approach and support it with the design of the PICASSO form generation system. It should be noted, however, that certain classes of end users require additional end-user facilities. This point is conceded by the EUFTG and is accomodated in their proposed conceptual environment for an end-user facility as shown in Figure 1. The pragmatics, which represents the interface between the end user and the end-user facility (i.e., the EUF is the shaded portion of the diagram), is a subsystem that converts a user's manipulative request to a canonical representation which is sent to the EUF. In many instances, a user request can be directly converted to a canonical representation - a representation containing statements in the data manipulation language (DML) of the host DBMS. However, in the remaining cases, a dialog type of facility must be created to assist

in acquiring the necessary information from the end user. A much greater effort must be expended to comprehend and then manipulate this dialog into a canonical expression of the user's need. Procedures for handling errors and user's requests for additional information should be incorporated in the pragmatics. Later in the paper, in a section entitled "Related Work", we discuss user interfaces at a more general level and describe some alternatives to the form's approach.

Before concluding this section let us examine the nature of query languages (i.e., the languages which provide an interface between the user and his/her database) that are available for the end user. Currently, the query languages which are associated with most data-base systems are tied to the data-base organization. In hierarchical systems, the notions of "next son" and "next brother" are necessarily intrinsically embedded in the query languages that are available (e.g., IMS's DL/I (IBM [1975]). In the network approach as exemplified primarily by the CODASYL data-base management system (CODASYL[1971]), the linkages required to establish associations between two schemas are realized through the concept of a set. To access properly such a data base the user must be aware precisely of what sets exist. Query languages for relational data bases have been oriented towards the relational algebra (CODD[1971]) or the relation calculus (CODD[1971]).

Unfortunately end users are not capable and/or not interested in learning such highly technical languages. In addition, these languages have been somewhat general purpose and thereby do not reflect the particular environment in which the user works. Certainly substantial research in the area of end user systems is justified and more will be said in general concerning this topic in the last section. Let us now examine the role of the data-base administrator in an end-user facility.

Data-Base Administrator in an End-User Facility

In a form-oriented end-user facility the data-base administrator assumes the traditional responsibilities including data definition, control over access, and integrity maintenance. Because the end user's perception of information in the data base is by means of forms, the data-base administrator will also be responsible for evaluating the user's information needs and assuring that the proper forms are defined to the system and made accessible to the user.

Referring to Fig. 1, we see that the data-base administrator establishes the user's information needs, as expressed in forms, by using the data definition facilities (i.e., the EUF DDL). The forms defined by the data-base administrator are called perceptual forms and these can best be viewed as canonical representations onto which user level forms are mapped. It is the responsibility

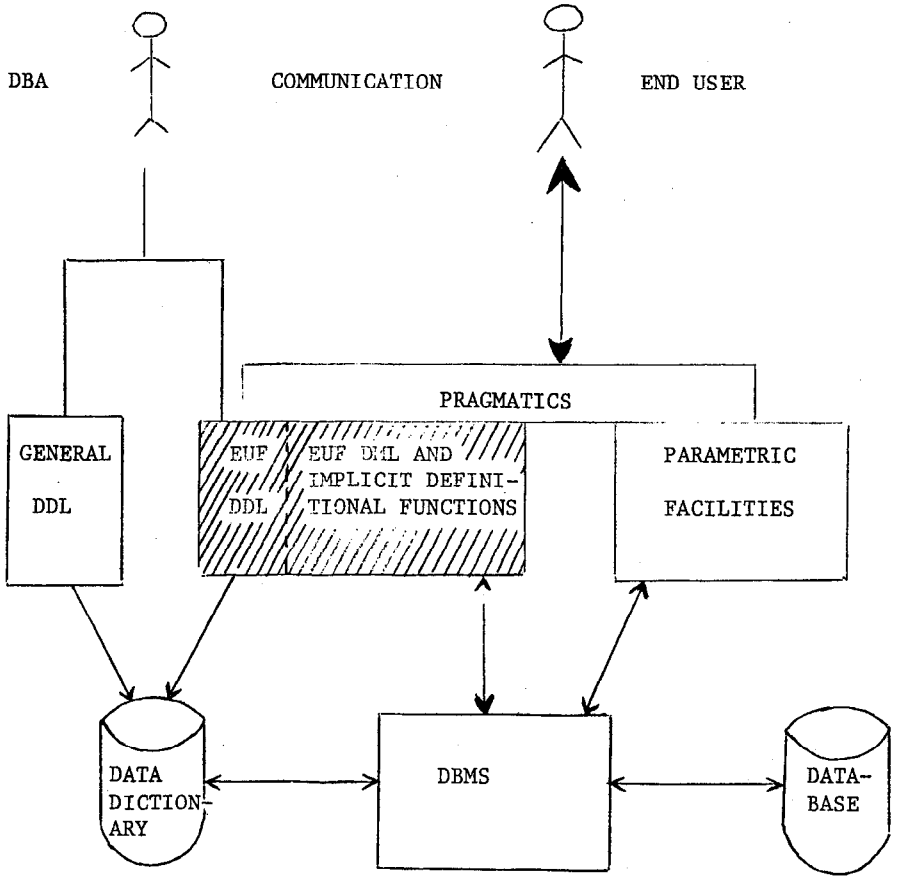

Figure 1 CODASYL EUF Task Groups, Conceptual Environment

of the data-base administrator to ensure that the information items basic to the perceptual form are inserted in the data dictionary. A remaining duty is to create a user profile through the data manipulation language (DML) facilities. Part of this profile includes a definition of the types of access a user can engage in when interacting with the data-base system.

From this brief review of the data-base administrator responsibilities it is clear that the creation of proper user form is one of the important additional duties in a form-based end-user facility. The success of such a facility is directly dependent on how well the data-base administrator can communicate with the end user and understand his/her needs. The development of a system to create and to modify forms easily and thereby enable the data-base administrator to identify quickly user requirements is imperative to the success of an EUF. PICASSO is a system to aid in form's design. Let us now enter into a tutorial description of the system.

Form Generation by PICASSO

In this section we describe in a tutorial fashion how PICASSO can be used to create a form. While the material is presented at a relatively informal level, an attempt is made to define precisely the syntax of the language and this is done with the aid of the BNF metalanguage. Throughout the discussion we assume that the forms are created on a display terminal device. While an attempt has been made to keep PICASSO device independent, the initial version interacts with an HP 2640 display terminal.

In PICASSO a form is viewed as a logical collection of words, lines and fields to be filled in as exemplified in Fig. 2. If we let C designate the set of all characters that can be present in a form, then this set will consist of the usual keyboard alphabet and a line drawing alphabet which is hard-wired or simulated in software. Every location on the form is associated with a character from C and if we let A be the set of addresses (i.e., character locations) on the form, then a form is a one-to-many mapping F: $C \rightarrow A$.

A. Form Addressing

Each location on a form has a unique address given by an ordered pair of numbers. The origin of the form is denoted as (0,0) and is at the top left corner as exhibited in Fig. 3. An address (x,y) refers to the location where row x intersects column y. The row co-ordinate is measured from top to bottom with the unit length equal to the length of a character field. (In PICASSSO, each character field is assumed to occupy a constant area. For some of the more sophisticated display terminals this requirement appears to be restrictive; nevertheless, it is a convenient method of addressing as will become clear in the discussion to follow.) Similarly, the column co-ordinate is measured from left to right with the unit length equal to the width of a character field. The co-ordinates of a legal address must each be non-negative integers.

Addressing relative to the origin immediately identifies an address on the form. A cursor is moved to the address identified. The addressing can be direct or relative with respect to either or both of the co-ordinates. Such addressing is used when the particular address of a geometry (various geometries such as lines, boxes, and textual strings will be discussed shortly) is known or is easily calculated from the present position of the cursor. The following syntactic description and examples illustrate how addressing is achieved in PICASSO.

```
<address> ::= address (<integer>,<integer>)
    <row> ::= row (<integer>)
 <column> ::= column (<integer>)
<integer> ::= + <usi> | -<usi> | <usi>
    <usi> ::= unsigned integer
```

These types of addressing are illustrated in the following examples.

```
address(1,5)  <=> move cursor to row 1; column 5
row(1)        <=> move cursor to row 1; do not
                  change columns
column(5)     <=> move cursor to column 5; do not
                  change rows
address(+2,3) <=> move cursor down 2 rows; and to
                  column 3
address(4,-6) <=> move cursor to row 4; and to the
                  left 6 columns
row(-8)       <=> move cursor up 8 rows
column(+2)    <=> move cursor to the right 2 columns
```

It is obvious that the addressing instructions contribute very little towards the design of the form per se. Addressing only establishes a base location from which the predominate shapes of the form; that is, the lines, textual fields, boxes, and tables, are described. Henceforth these shapes will be referred to as geometries. In the remainder of this section, we will examine these geometries.

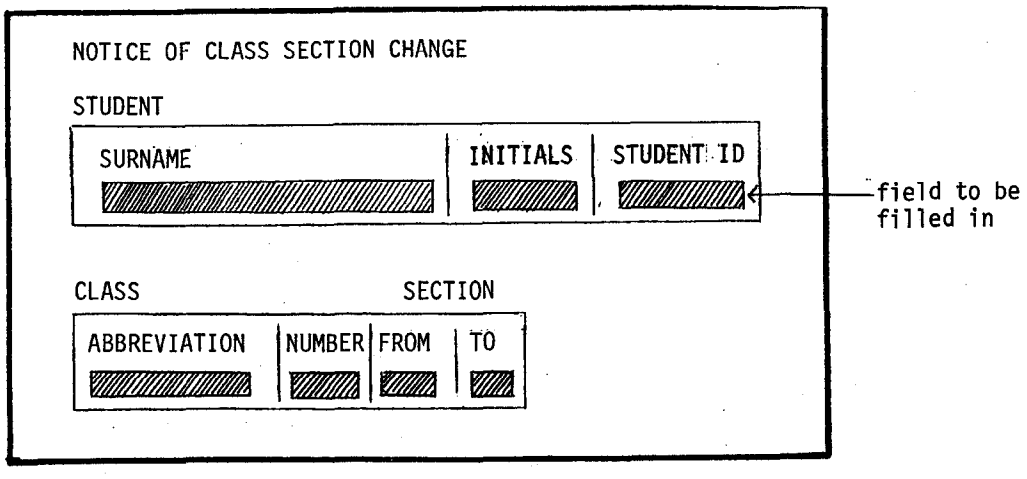

Figure 2 A typical form

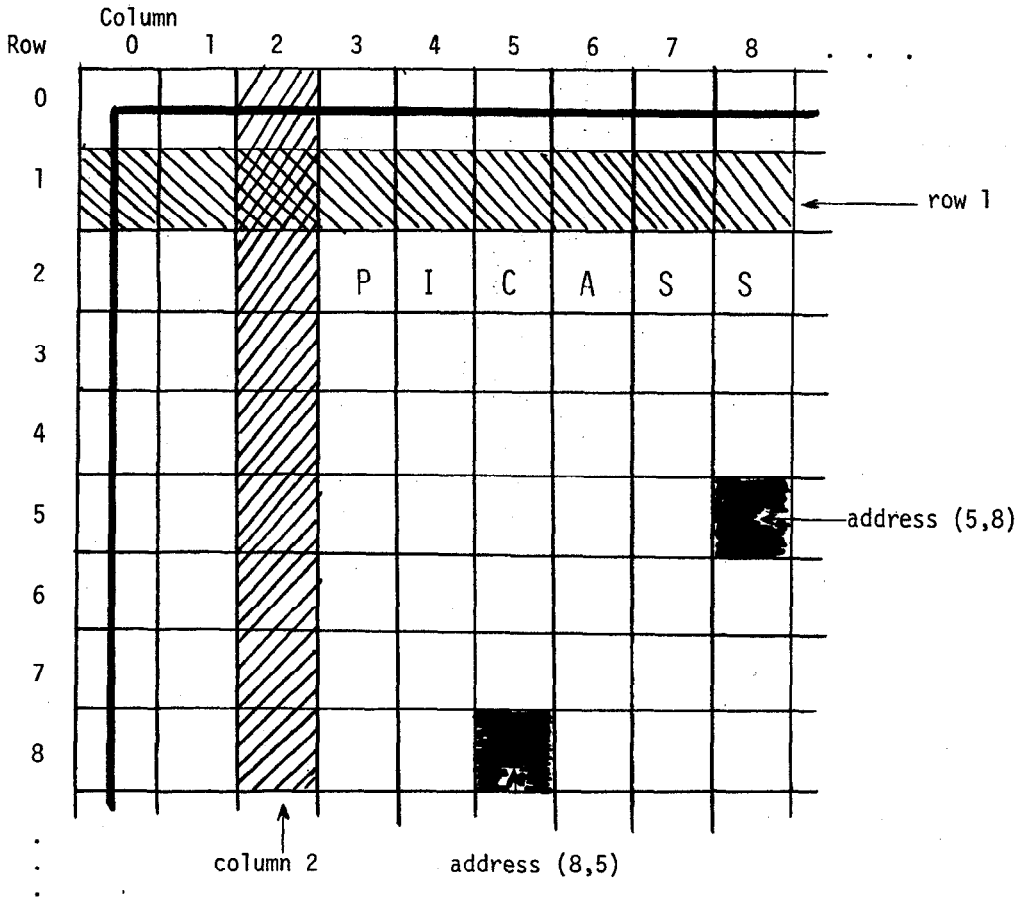

Figure 3 Example illustrating form addressing
Note that ⌐, —, ∣ are characters from the line drawing alphabet.

B. Line Geometries

The two line geometries in PICASSO are the obvious ones which describe horizontal and vertical lines. The horizontal line instruction generates a line horizontally starting from the current cursor position and moving right. The vertical line instruction produces a vertical line beginning at the current cursor position and proceding downwards. The line is described in segments by the length and type of each segment. The length of the segment is in columns or rows. The type is one of bold, blank, double or single (not all of these are available on some display terminals and may have to be simulated, if possible). The two line geometry instructions have the following format.

```
<horizontal> ::= horizontal <type> (<usi>)
  <vertical> ::= vertical <type> (<usi>)
      <type> ::=  bold | blank | double
                | single
       <usi> ::= unsigned integer
```

An example which illustrates these instructions is given in Fig. 4. In the example program, the first instruction sets the cursor at position (1, 1). The second instruction draws a double horizontal line five in length. After resetting the cursor to (2, 2) a vertical single line, a vertical bold line and horizontal double line are drawn as shown in Fig. 4. Note that the position of the cursor after a horizontal or vertical instruction is respectively at the location one column to the right or one row below the last character in the line. In the example, the final cursor position would be (6, 7).

C. Textual Geometries

Recall from the beginning of this section that a form consists, in part, of a collection of words and fields to be filled in by the end user. The blank fields to be filled should be sensitive to the user's information; that is, one can visibly insert the information into the space provided. The rest of the form, which generally contains predefined textual fields and lines, should be insensitive to the user's input and thus it should be impossible for the user to destroy the form inadvertently. In PICASSO, all fields are protected (unchanged) except those which are declared unprotected.

A single instruction, the write instruction, is provided to write text and unprotected fields on the form. A prescribed sequence of characters are printed from left to right starting at the position of the cursor. Fields which are unprotected are prefixed by the "∿" operator. Column addressing in included to facilitate an efficient and simple way of generating a string of text which is delimited by several blank characters.

```
<write> ::= write ( <parm list> )
<parm list> ::= <parm> | <parm> , <parm list>
<parm> ::= <text> | <column>
<text> ::= "string" | blank ( <usi> ) | ∿"string"
         | ∿blank ( <usi> )
<usi> ::= unsigned integer
```

The following set of instructions generates

```
address (1,1)
horizontal double (5)
address (2,2)
vertical single (1)
vertical bold (3)
horizontal double (5)
```

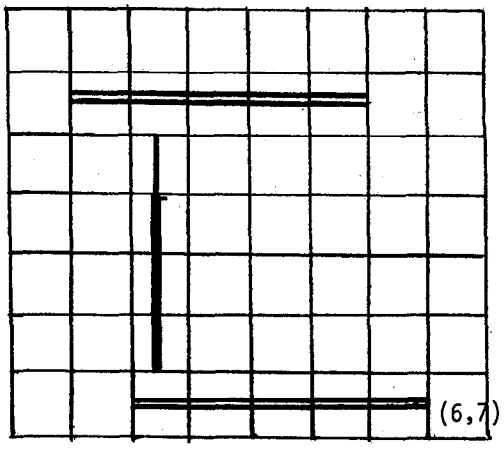

Figure 4 Example illustrating the line drawing instructions

the form shown in Fig. 5. In this example, the shaded areas are the unprotected fields which are to be filled in by the user. The "shadow text", dd/mm/yy, indicates to the user how the data field is to be filled in.

```
address(1,1)
     write("Fill in all blanks",column(+5),
           "Press 'return' when done")
address(2,1)
     write("Date: ",~"dd","/",~"mm","/",~"yy")
address(3,1)  write("Name:  ",~blank(20))
```

```
Fill in all blanks      Press 'return' when done
Date: dd/mm/yy
Name:
```

Figure 5. An example form illustrating the write instruction.

D. Box Geometries

It is a very common practice in forms design to group certain logically related items together in a box (for example, see Kaiser [1968] and Tavernier [1972]). This approach is illustrated in Fig. 2 by the grouping of student and class information in separate boxes. This grouping concept is often generalized to include the design of boxes within boxes; thereby, illustrating that within a logical grouping of data there exist subgroups. In fact, a direct analogy can be drawn between a box and a grouping identifier within a record structure of some programming languages. For example, the PL/I record structure for a class change record might be

```
DECLARE 1 CLASSCHG,
          2 STUDENT,
            3 SURNAME CHARACTER (25),
            3 INITIALS CHARACTER (2),
            3 STUD_ID CHARACTER (6),
          2 CLASS,
            3 ABBREV CHARACTER (8),
            3 NUMBER CHARACTER (4),
            3 FROM FIXED (2),
            3 TO FIXED (2);
```

The similarities between the record structure and the boxes in Fig. 2 are obvious.

To create a box in PICASSO we use a box instruction. The instruction generates a box whose top left corner coincides with the most recent cursor position. The position of the cursor is then treated as a new origin so that all subsequent addresses and creations of geometries can be made relative to this point. The box, itself, is described by its inner dimensions and the type of line which forms the box. Therefore, syntactically the box instruction is

```
<box> ::= box <type> ( <usi> x <usi> )
<type> ::= bold | blank | double | single
```

As an example, the sequence of instructions to the left of Fig. 6 generates the box represented in the accompanying figure.

The presence of a blank type supports our view that a box is more than a collection of physical lines. It is a logical unit. Once a box is established the user can ignore the rest of the form and concentrate on filling in the inside of the box before proceeding to another box at the same level.

Every instance of the box instruction changes the position of the current origin and previous origins can only be recovered when all boxes generating subsequent origins are completed. The completion of a box is indicated by an end of box instruction.

```
<end of box> ::= end box
```

A PICASSO program which creates the form in Fig. 2 is as follows.

```
comment(PICASSO program for generating the class
        change form.)
box   bold(17x61)
  address(1,5)
      write("NOTICE OF CLASS SECTION CHANGE")
  address(3,5)  write("STUDENT")
  address(4,5)
  box  single(4x52)
       address(1,2)  write("SURNAME ", column(30),
                           "INITIALS", column(41),
                           "STUDENT ID")
       address(2,2)  write(~blank(25), column(33),
                           blank(2), column(43),
                           ~blank(6))
       address(1,28)  vertical single(2)
       address(1,34)  vertical single(2)
  end box
  address(9,5)  write("CLASS", column(35),
```

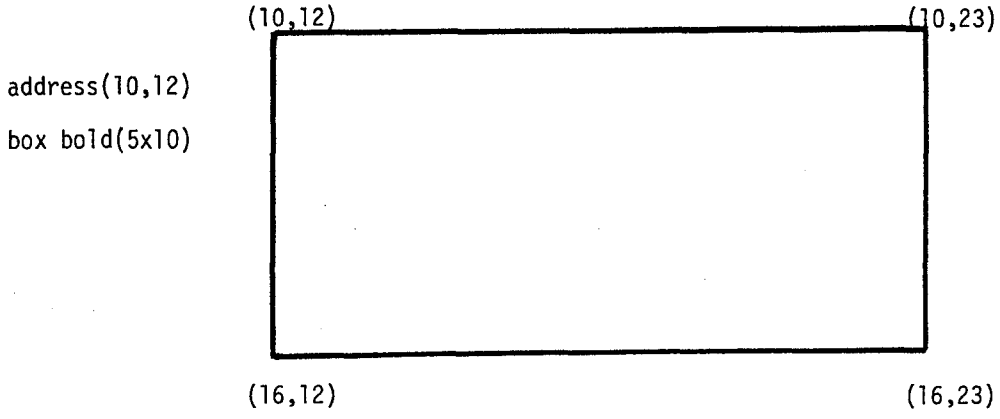

Figure 6 Example illustrating the construction of a box.

```
                                "SECTION")
  address(10,5)
  box single(4x37)
        .
        .   comment(Instructions for creating the
        .           class section box would fall
        .           here.)
  end box
end box
```

E. Midpoint Addressing and Centering Textual Geometries

In designing a form - in particular, in titling parts of a form - it is convenient to be able to center textual geometries with respect to a particular address. In PICASSO, this operation is achieved through the use of the center instruction which has the following form:

 ::= center(<text list>)
<text list> ::= <text> | <text>, <text list>

where <text> has the same syntax and semantics as described in the textual geometry subsection.

As an example, suppose we wish to center the text "NOTICE OF CLASS SECTION CHANGE" with respect to character position 31 in the form as given in Fig. 2. This can be accomplished with -

```
address(1,31)
center("NOTICE OF CLASS SECTION CHANGE")
```

It is a very common situation that the address about which a text string is to be centered is the midpoint of a geometry. To locate such a midpoint address with respect to a given geometry we use the following instruction.

<midpoint addressing> ::= midpoint
(<m-geometry>)
<m-geometry> ::= horizontal | vertical
| box

We can illustrate midpoint addressing by redoing the previous example assuming a current origin of (0,0) for the box of dimension (17x61) as:

```
row(1) midpoint(horizontal)
center("NOTICE OF CLASS SECTION CHANGE")
```

In this example the midpoint is found with respect to the last box (or entire screen area if no box has been defined) in the horizontal direction of the current row.

Midpoint addressing relieves the form's designer of the rudimentary task of computing the middle of a given geometry. This idea is exemplified again in the following program which creates the form in Fig. 7.

```
address(5,5)
box single(13x28) row(1) midpoint(horizontal)
    center("PICASSO", ∿blank(2))
    midpoint(box) center("midpoint of box")
end box
```

Note that midpoint addressing need not always be used in conjunction with centering. For example, it is convenient to be able to find the midpoint of a geometry and use relative addressing in creating other geometries based on this midpoint.

F. Table Geometry

The final geometry we examine is the table geometry. This type of geometry is particularly useful in designing forms requesting lists of items say for inventory control. Fig. 8 (taken from Tavernier [1972] Fig. 5.1) is an example of such a form.

In PICASSO a table is composed of rows and columns which vary in size and which are subscribed by using vertical and horizontal lines. These lines may vary in type (i.e., double, single, bold or blank lines). The fields in a table can be blank, or contain protected and unprotected text. Only tables of a normal form can be described by the PICASSO table instruction. By a normalized table, we mean all rows must have the same number of columns and all columns must have the same number of rows. Unnormalized tables can be achieved by modifying a normalized skeleton using other PICASSO instructions involving blank geometries.

At first glance, it appears that a table can be generated using the previously described instructions. This is true; however, the process would be tedious and complex. In addition, the intersection of a horizontal and a vertical line geometry creates a gap at the intersection point as is illustrated in Fig. 2. In the construction of a table we know precisely where horizontal and vertical lines intersect and at these positions we can make use of join characters such as "├", "┤", "+", "┴", or "┬" instead of the line characters "|" or "-". For terminal devices without such special line drawing characters, the problem of line separations at intersection points will persist unless

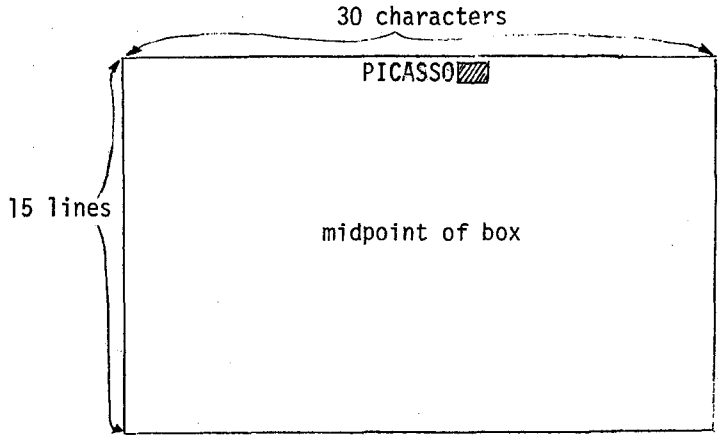

Figure 7 Centering and midpoint addressing example form

Part number
Description
Cost code
Danger level
Bin number

Date	Contract	Quantity in	Quantity out	Balance	Date	Contract	Quantity in	Quantity out	Balance

Figure 8 Stock and record form

some suitable character such as the "+" operator can be used.

A grammar describing the table instruction follows. Additional restrictions will be made as the constituent parts are discussed.

```
<table> ::= table <type> ( <usi> x <usi> )
            column <logical column list>
            row <logical row list>
<logical column list> ::= <logical size>
                        | <logical size> <logical
                          column list>
<logical size> ::= <usi> ( <usi> , <type#> )
<logical row list> ::= <logical row> | <logical row>
                       <logical row list>
<logical row> ::= <logical size> (<physical row
                  list>)
<physical row list> ::= <physical row> | <physical
                        row> , <physical row list>
<physical row> ::= blank | unprotected
                     | ( <field list> )
<field list> ::= <field> | <field> , <field list>
<field> ::= blank | unprotected | "string"
<type#> ::= <type> | #
```

An instance of a table declaration in the table construct is: table bold(22x30). This declares the inside dimension of the table to be 22 character positions long by 30 character positions wide and the border of the table to be bold line type.

Before proceeding with the discussion of table geometries the terms physical column, physical row, logical column and logical row must be defined. A physical column is a column one character position in width and a physical row is a row one character position in height. Logical columns and logical rows are the columns and rows formed in a table and these can be one or more physical columns or physical rows in width and height, respectively.

With reference to the BNF description given

earlier, a <logical column list> describes the width in character positions of each logical column and the line type of the vertical line to the right of it. An instance of <logical size> description is: 2(3, single). For a column definition, this designates logical column 2 to be 3 character positions wide with a single vertical line on its right boundary. There should be one <logical size> entry in the <logical column list> for each logical column. We insist that the order of the <logical column list> entries be in increasing magnitude of the logical column number, the first <usi> in a <logical size>. A restriction on the <logical column list> is that the <type#> of the last <logical size> be set to '#'. This is understandable since the vertical line to the right of the last logical column is the border and the border has already been described in the table declaration. It is expected that the width of the table given in the table declaration be consistent with the width of the table determined by the logical columns. Therefore, the sum of physical column lengths, the second <usi> in <logical size>, plus the number of vertical lines must equal the column dimension given in the table declaration.

The <logical row list> describes the width of each logical row and the line type of the horizontal line below it. It also describes the contents of each field in each physical row. A field, in this sense, is the intersection of a logical column with a physical row. An instance of a <logical row list> is:
4(3,double)(blank,("lit1",blank,"lit3"),unprotected).
This describes a logical row to be 3 physical rows wide with a double horizontal line below it. The first physical row (of logical row 4) is blank in each field. The second physical row has the string "lit1" in its first field; a blank in its second field; and the string "lit3" in its third field. The third physical row is unprotected in each field.

The instance: blank,("lit1",blank,"lit3"), unprotected is a physical row list . The ith <physical row> in a <physical row list> corresponds to the ith physical row of a logical row. When a physical row is 'blank' or 'unprotected' then all the fields of the corresponding physical row are blank or unprotected respectively.

If the <physical row> is a <field list>, i.e., ("lit1",blank,"lit3"), then there is a one-to-one mapping from the <field list> to the physical row. When a <field> is 'blank' or 'unprotected', then the corresponding field of the physical row is blank or unprotected. When <field> is a string, then that string is automatically centered in the corresponding field of the physical row.

As with the <logical column list>, the <type#> of the last <logical row> in the <logical row list> must be '#'. Also, the sum of the physical row lengths plus the number of horizontal lines must equal the row dimension given in the table declaration.

The following instructions describe the table in Fig. 9. The blank physical rows are marked with arrows at the side of the table.

```
table bold(11x26)
column 1(8,double)
       2(8, single)
       3(8,#)
row    1(3,single)(blank,(blank,"title1","title2",
         blank)
       2(3,single)(blank,("title3",unprotected,
         unprotected),blank)
       3(3,#)(blank,("title4",unprotected,unprotect-
         ed),blank)
```

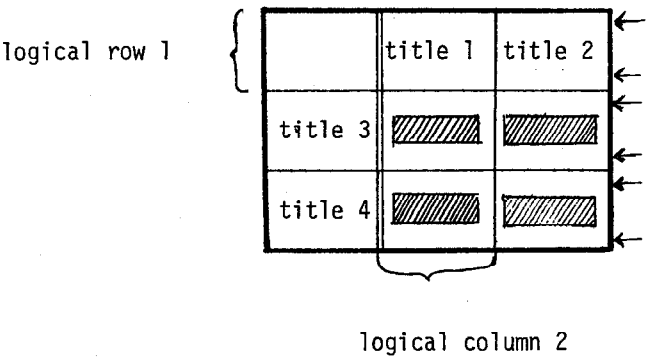

Figure 9 Typical normalized form.

G. Geometry Enhancement

In the illustrations of forms so far the unprotected fields have been shaded. This shading represents an inverse video enhancement of a blank field. Other types of enhancements incorporated in PICASSO are blinking, underlining and half-bright - obviously not all of these are available on all video terminals.

In PICASSO the default enhancement of all protected geometries is normal brightness while the default enhancement for unprotected fields in inverse video. The default may be changed by either a local or global instruction. A local instruction provides enhancement for a single geometry specification only and the instruction appears immediately prior to specification. It has the following form

```
<local enhancement> ::= local ( <enhance list> )
<enhance list> ::= blinking | inverse |
                   bright | underlined
```

As an example,

```
local(blinking,inverse)
box single(4x8)
```

produces a box which is displayed in inverse video with blinking characters.

The global instructions, "global" and "~global", are used to enhance a series of geometries. These enhancements stay in effect until the next "global" or "~global" instruction is encountered. Syntactically, these instructions appear as

```
<global enhancement> ::= global ( <enhance list> )
<unprotected global enhancement> ::= ~global
                                     ( <enhance list> )
```

The following examples illustrate these instructions.

global(blinking) <=> all protected geometries (until the next global instruction) have a blinking enhancement.

~global(inverse,underlined) <=> all unprotected geometries (until the next ~global instruction) will have an enhancement of blinking underlined characters.

This concludes our tutorial discussion of PICASSO. We have omitted some features such as the ability to abbreviate commands and to apply instructions repetitively. However, the description should give an appreciation of the language and how it can be applied. Let us now compare PICASSO to some of the specifications outlined in

the CODASYL End User Facility Report.

Comparison with the EUF Report

In the CODASYL EUF Report, a number of objects are defined which allow an end user to conceptualize, at a logical level, data into the following classes: data base, file, folder, form, group and item. The concepts of form, group, and item have a physical realization in PICASSO's form, box (and table), and textual geometries, respectively. Hence a DBA should have no problem in translating a user's view of the data to these primitives.

The notions of data base, file, and folder have not been handled, as yet, in PICASSO. Logistically, there should be little difficulty in adding commands for handling these higher level objects. The major difficulty arises when attempting to realize these concepts in the underlying physical files and physical data base.

Facilities describing how data should be manipulated (e.g., adding, deleting, updating data) have been ignored in this discussion. Similarly, a data manipulation language (DML) has not been proposed by the EUF Task Group. Certainly such language features must be intimately tied to the pragmatics of the system.

Probably the most important consideration when viewing PICASSO in relation to the EUF Report is whether it is a system capable of easily describing the example forms given in the appendices of the Report. We feel that given a basic understanding of PICASSO these forms can be quickly generated by a data-base administrator who need not learn the intricacies of working with line drawing sets for a particular terminal device.

Related Work

The efforts related to PICASSO are part of a more general investigation into the subject of user-oriented query languages (henceforth referred to as UQUELS). We can characterize this class of languages in the following manner:

> "A UQUEL is a self-contained, high-level query language which allows the user to interact with a data base in order to accomplish a specific task. The UQUEL does not necessarily retain a view of the data inherent to the specific data base of data-base approach." (Malakoe [1977])

Therefore, a UQUEL should not be designed for an underlying data-base management system.

Three general methods of interacting with the user are apparent. These are question-answering dialogue (e.g., Codd[1974]), simple predicate based systems (e.g., Zloof[1975]) and the form's based approach (e.g., EUF Task Group Report [1976]).

For some applications, an appropriate end user facility should contain features from all of these types of UQUELS. The blending together of these types of interaction must be handled in the pragmatics section of the EUF. Just how this can best be accomplished is part of our general investigation.

Probably the strongest statement related to the area of end user query languages is by Codd [1971].

> "Many users need query languages specialized to their applications. The high cost of supporting a great variety of these languages and their translators suggest that as many as possible of the common services in these translators be identified and programmed once and for all."

In a second part of our general investigation, we are attempting to identify these general properties described by CODD. In so doing we plan to support the fast, relatively efficient, and inexpensive generation of UQUELS for special applications through the use of UQUEL writing systems for each class of UQUEL.

Conclusion and Further Development

In this paper we have introduced a system for aiding in the design of forms for an end-user facility. While the various form generating commands have been only briefly described, it should be clear that PICASSO is a system which can greatly assist the data-base administrator.

Further work is needed to complete the system. Particular attention must be focused on extending the form primitives to accomodate data objects such as folder, and file as suggested by the EUF Task Group. The system is being implemented as part of a form's interface to the INGRES relational data-base system (Stonebraker[1975]) on a PDP-11/40. It is planned that the system be applied to a number of application areas and hence a number of different data bases.

As stated previously, PICASSO was designed to be independent of any type of display terminal or operating system. To this end, we plan to implement the system on several types of display terminals and, initially, in two different operating environments: the PDP-11/40 and IBM S370/158.

Addendum - The name PICASSO

We decided to avoid the more traditional approach of using abbreviations or acyronyms when selecting a name for our language. Any resenblance between the name of this language and a certain artist whose work is reknown for its varied and unusual geometric qualities is not coincidental.

Acknowledgements

This research was sponsored in part by the National Research Council of Canada, Grant No. A9290. We would like to thank Janet Morck for her able assistance in typing the manuscript.

Bibliography

A Progress Report on the Activities of the CODASYL End-User Facility Task Group, FDT Bulletin of ACM SIGMOD, Vol. 8, No. 1, 1976, pp. 1-19.

CODASYL Data Base Task Group, April 1971 report, ACM, New York

Codd, E.F.: "Relational Algebra:, Courant Computer Science Symposia 6, Data Base Systems, New York, Prentice Hall, New York, 1971.

Codd, E.F.: "A data base sublanguage founded on the relational calculus", Proc. 1971. ACM-SIGFIDET Workshop on Data Description Access and Control, ACM, New York, pp. 35-68.

Codd, E.F.: "Seven Steps to Rendezvous with the Casual User", Data Base Management, J.W. Klimbie and K.L. Koffeman(eds), North-Holland Publishing Co., Eindhoven, The Netherlands, 1974, pp. 179-200.

IBM, Information Management System/Virtual Storage (IMS/VS), General Information Manual, GH20-1260-3, 1975.

Kaiser, J.B.: Forms Design and Control, American Management Association, Inc, 1968.

Malakoe, G.: "User Oriented Query Languages - UQUELS", Master Thesis, University of Saskatchewan, to be submitted in 1977.
Silas, J.: "Good Forms Design Cuts Costs", Journal of Systems Management, Dec. 1976, pp. 38-21.
Stonebraker, M.R.: "Getting Started in INGRES - A Tutorial", University of California, Berkeley, ERL Mem., No. ERL-M518, Apr. 1975.
Tavernier, G: Basic Office Systems and Records, Gower Press, Epping, G.B., 1972.
Zloof, M.M.: "Query by Example", Proc. Nat. Computer Conf., AFIPS Press, Vol. 44, 1975, pp. 431-438.