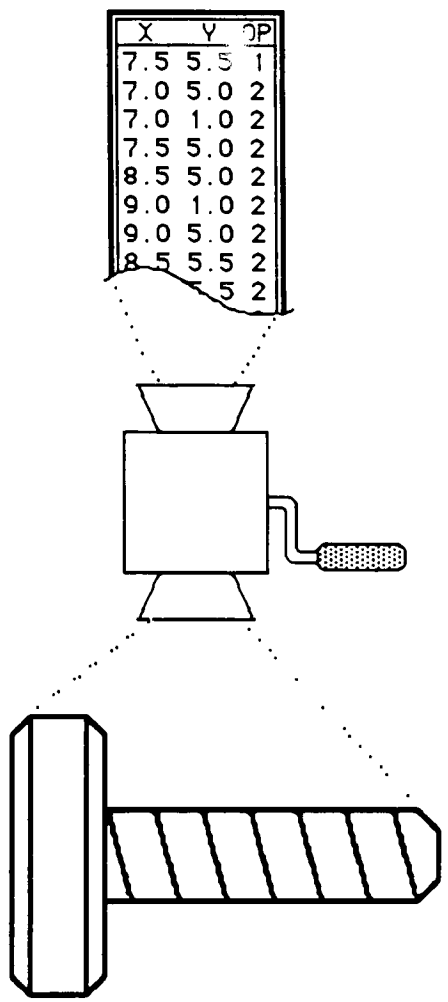
1987

An image storage system using a relational database management system to facilitate picture data handling

Daryl G. Johnson

An Image Storage System Using a Relational Database Management System to Facilitate Picture Data Handling



by
Daryl G Johnson

A thesis submitted in partial fulfillment

of the

Master of Science in Computer Science Degree Program

at

Rochester Institute of Technology

To my wife, Linda

A complete list of trademarks appears in Appendix F.

List of Figures

Table of Contents

Approvals

Thesis Advisor:

Evelyn P. Rozanski
Professor, Acting Director
School of Computer Science and Technology
Rochester Institute of Technology

Thesis Committee Members:

Dr. Rayno D. Niemi
Associate Professor Undergraduate Department
School of Computer Science and Technology
Rochester Institute of Technology

Dr. Lawrence A. Coon
Associate Professor Undergraduate Department
School of Computer Science and Technology
Rochester Institute of Technology

Abstract

AN IMAGE STORAGE SYSTEM USING A RELATIONAL DATABASE MANAGEMENT SYSTEM TO FACILITATE PICTURE DATA HANDLING

by
Daryl G Johnson

The Image Storage System (ISS) is a general purpose graphics facility for the storage and retrieval of pictures. ISS utilizes the capabilities of the Mistress Relational Database Management System, the Relation Interpreter Package and the CORE Graphics Package. The system was developed to illustrate that by utilizing a database system for the storage of the actual picture description, users can incorporate graphics into their applications very simply without becoming a graphics or programming expert. Through the database, image data can be shared between applications and changes or additions to the image will not require program modification. As a consequence of using a data base management system, graphical as well as non-graphical information may be stored together. An important out come of this is that the graphical data may be accessible for non-graphical purposes as well.

The database schema is illustrated and its flexibility and user adaptability is demonstrated. The graphics image interpreter is described as well as the underlying Relation Interpreter Package on which it is built.

Introduction

The Problem

There are four significant problems in the area of graphical applications that will be addressed here:

- Incorporation of Graphics into an Application
- Image Maintenance
- Structuring of Images
- Association between Graphical and Non-Graphical Data

Incorporation of Graphics into an Application

For many applications, the inclusion of even simple graphics can clarify the application development process and facilitate communication with the user. A serious deterrent to the wide spread use of computer graphics in the general applications environment is the difficulty in using the current development support tools. Generally, users shy away from or never even consider adding graphics to their applications because of the complexity of the tools that are available to them.

Defining a conceptual model, designing appropriate data structures, and developing the necessary algorithms are problems faced in all varieties of programming and graphics applications are no exception. To include graphics in an application, users must familiarize themselves with the intricacies of a graphics package of some kind. In addition, the user may be responsible for developing routines to perform complex tasks such as, translation, scale and rotation of objects, which are often not part of a graphics software package.

The casual user typically can visualize the picture he would like to add to an application, but has no knowledge of graphics programming, and furthermore, does not wish to acquire it. The user should be able to generate a picture from within an application with a simple command such as:

```
DrawPicture( "database", "wheelbarrow" )
```

Image Maintenance

If the image is intended to be shared by several users or many different applications, a duplication of effort in writing the programs to generate the image, or at least a duplication of the code, can result. Later, if corrections or enhancements to the image are needed, the problem of maintaining the many programs or many copies of a program generating this image can become unmanageable.

Structuring of Images

Most complex images are the combination and repetition of other less complex images. To construct these complex images we usually subdivided the picture into components that each could be considered an image by itself. We construct those simpler images and then assemble them to form the more complex picture. Quite often the simpler images are used frequently and we make a copy of them for each use rather than reconstructing the image from scratch. ISS should allow the user to build an image up from parts in just such a natural fashion.

Association between Graphical and Non-Graphical Data

Another problem that arises is that the specification of the graphics image is typically handled as a special case and stored separate from the non-graphical data associated with the application. This makes the management of the overall problem more difficult. The separation also makes the use of the graphical data for non-graphical purposes much more difficult. It would be logical to store the graphical specification in the same database as the non-graphical data, thus maintaining the entire application as a cohesive unit. Generally, data management facilities are not included in graphics software systems and vice versa.

A Solution

The problems outlined clearly indicate a need for a system which makes it simpler for users to add graphic output to their applications and to share and maintain images efficiently.

To accomplish this, the image specification is treated as data and stored in a relational data base along with any non-graphical data. A comprehensive system has been designed to generate a picture from an image specification stored in a database rather than writing custom programs for every picture to be generated. Facilities have been built into this system to allow the formulation of images from other images thus providing for a hierarchical organization of a picture. Figure 1-1 shows a conceptual overview of the components of the Image Storage System (ISS) system.

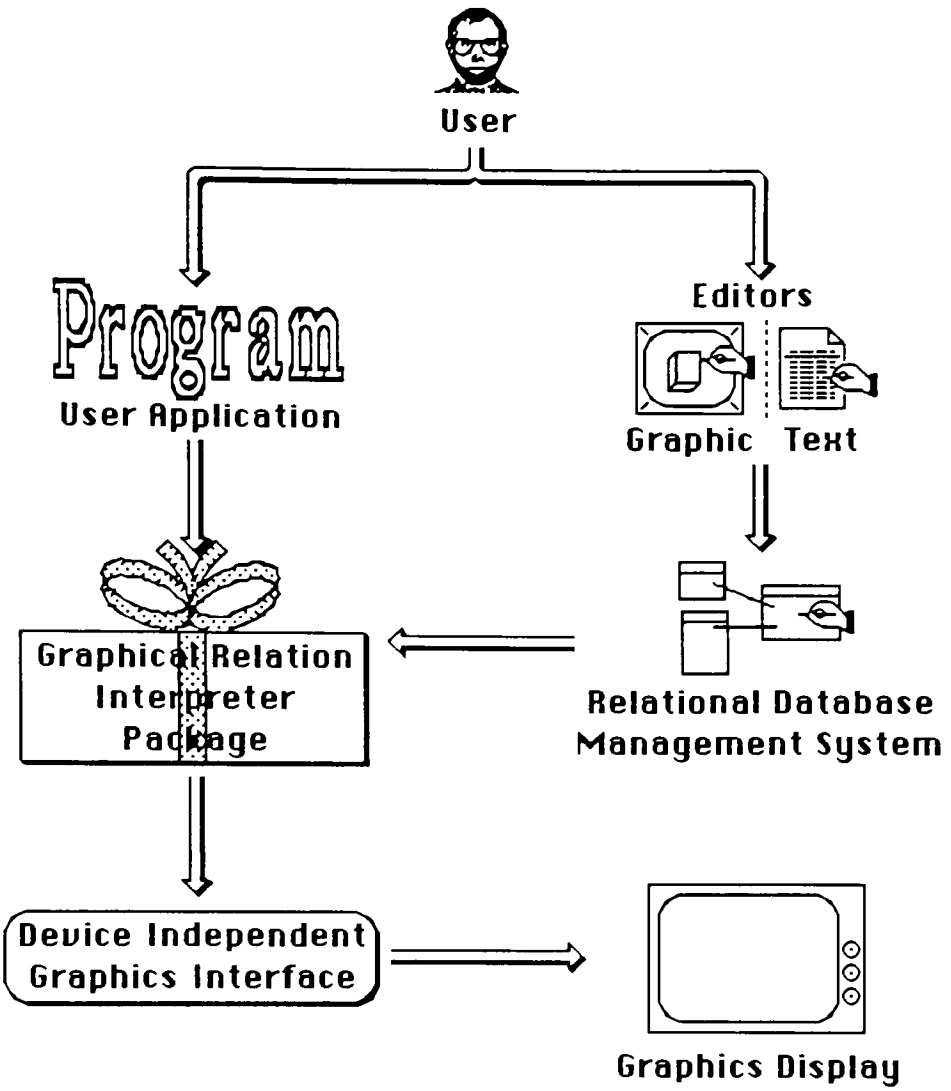


Figure 1-1. ISS system conceptual overview.

Graphics Relation Interpreter Package (GRIP)

General Description

Most people find that it is easier to understand and place information in a chart or tabular form than it is to read or write a program even if the same information and instructions are presented in both cases. This familiarity with tables can in part be attributed to the prevalent use of two dimensional paper and fill-in-the-blank forms and the ability to visually see the relationships between the columns and rows for the table. This idea is well supported by the wide spread use of tabular spread sheet programs where the users manipulate a table of data specifying various relationships between the rows and columns of the table. The users of these types of programs come from all application areas, such as financial, business and engineering, and from all levels of computer experience. The underlying concept is that these users are not programming but manipulating data. The concept of not programming appeals to the typical user inexperienced in the areas of computer programming.

Tables for Graphics

For a graphics application, a user who wished to specify an image to be drawn might create a table similar to that in figure 2-1.

X	Y	XYOP
3.000000e+00	2.500000e+00	mov
6.000000e+00	2.500000e+00	draw
6.000000e+00	3.500000e+00	draw
3.000000e+00	3.500000e+00	draw
3.500000e+00	2.500000e+00	draw
3.500000e+00	3.500000e+00	move
4.000000e+00	2.500000e+00	draw
4.000000e+00	3.500000e+00	MOVE
4.500000e+00	2.500000e+00	DRAW
4.500000e+00	3.500000e+00	Move
5.000000e+00	2.500000e+00	Draw
5.000000e+00	3.500000e+00	move
5.500000e+00	2.500000e+00	draw
5.500000e+00	3.500000e+00	move
6.000000e+00	2.500000e+00	draw

Figure 2-1. *Table named* ***threads***

From this table, most users could determine the intent of the data presented, taking into account the descriptive column headings and the form of the contents. The data in this table represents a collection of lines drawn in a cartesian coordinate system, more specifically, an image of the shaft of a bolt outlining the threads. This detailed description is not immediately available from the table but the image of this object, shown in figure 2-2, is.

Figure 2-2. *Picture described by table* ***threads***

The problem now is communicating the intent of the data presented in this table to a computer or graphics device to perform this task. We, as

humans, can discern the purpose of the numbers in each column of the table from the name of each column. Yet, other names for these columns might have been appropriate such as *A* and *B*, *height* and *width* or *latitude* and *longitude*. It would be inconvenient to place the restriction on the user to be able to use only x and y to represent graphics coordinates, and yet, it would be an immense job and waste of resources to define every possible name which could be used. The problem of ambiguity also exists. Column headings such as *width* might be used to describe the size of men's trouser cuffs and not be intended for a graphical location. It appears that trying to discern intent from the name of a column is difficult at best and has limitations.

Semantics

Since interpreting the column titles to determine the intended use of the data can be both a monumental as well as ambiguous task, some additional information is necessary. The approach is to associate with each column of a table a code which could describe the intent or, more generally, the semantic meaning of the data in that column. Since it is maintained independently from the name, the semantic code puts no restrictions on the name. Most relational databases do not provide any facilities for maintaining a semantic definition for the columns of tables or relations. The semantic meaning of each column or attribute must then be maintained separately. There are many possible schemes for maintaining the semantics but it would be convenient to store the semantics in the same database as the tabular data. This organization is an aid in maintaining and in keeping logically related information within close proximity. In order to store the semantic description in a database it is necessary to create another table in

the database. The second table will describe the data table and its semantic meanings. Figure 2-3 shows a table describing the columns of the ***shaft*** data table and defining the semantic codes for each column.

attr number	attr name	attr syntax	attr semantics
1	x	float	8
2	y	float	9
3	xyop	char(4,1)	10

Figure 2-3. *Structure table _shaft*

GRIP is built upon the Relation Interpreter Package (RIP) which will be described in detail in the next section. It is helpful here to introduce some of the terminology and concepts that are the basis of RIP and of particular interest to GRIP users. Figure 2-4 provides a brief overview of the components of the GRIP system. The table describing the format and semantics of a data table is called a structure table or relation. All Mistress tables are distinguished by unique names [Rh82e]. The name of the data table in our example was **shaft**. The convention used by the Relation Interpreter Package for the naming of structure tables is to use the name of the data table being described prefixed with an underscore (_). Consequently, the structure table which describes the **shaft** data table is named **_shaft** and when a data table is to be interpreted, the structure table must be retrieved first to determine the semantics of the attributes of the data table.

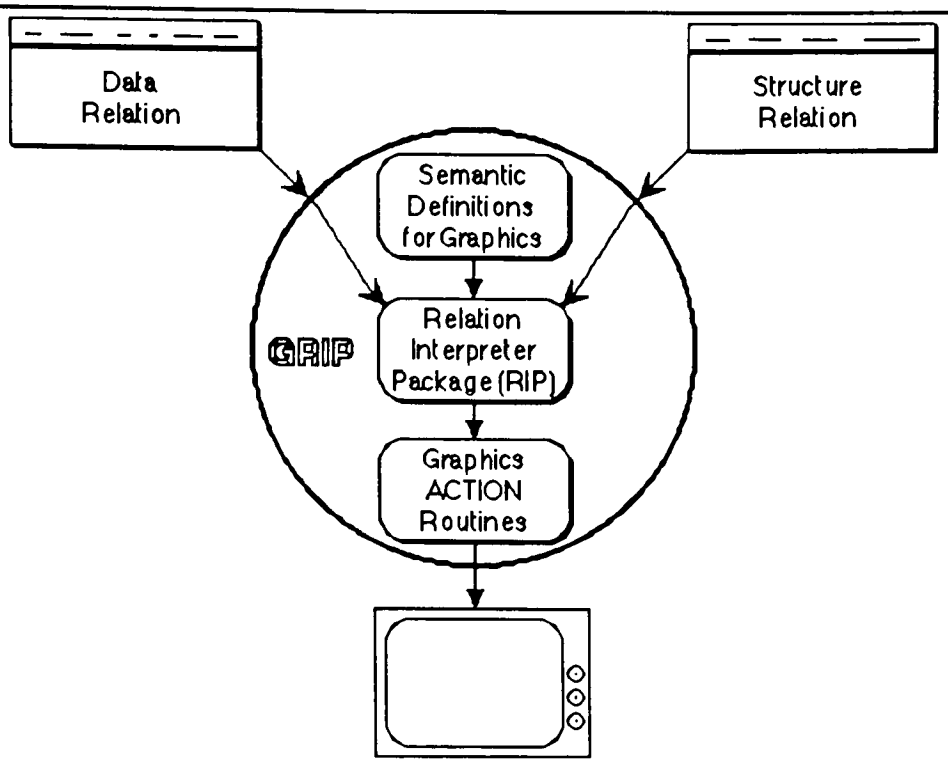


*Figure 2-4. Overview of **GRIP** system.*

Figure 2-5 shows the semantics currently available through the Graphics Relation Interpreter Package (GRIP) and the integer codes used to represent them.

Semantic Code	Semantic Meaning
8	X coordinate to be used with next xy operation
9	Y coordinate to be used with next xy operation
10	XY operation to be performed
11	Recursively draw this named sub-picture
12	Set a new color to be used from now on
13	Apply new instance transformation sequence

*Figure 2-5. Available semantic meaning definitions for **GRIP***

This set can easily be extended to include many more graphics primitives. The section on the Relation Interpreter Package (RIP) describes how this can be accomplished. GRIP functions basically as an intelligent, configurable table walker. Once the structure table has been read, the interpreter begins retrieving data from the data table one row at a time beginning with the first row and examining the columns one at a time from left to right. If that column has a graphical semantic meaning, it is interpreted. It should be noted here that not all columns of a data table must be described in the structure table and therefore need not have any semantic meaning. The interpreter will merely skip over those columns unnoticed. Also, there may be columns in the data relation which have a semantic meaning defined for them that does not pertain to GRIP. The section on the Catalog Relation Interpreter Package illustrates such a situation. GRIP will skip over these columns as well. Therefore, GRIP only interprets those columns with a semantic meaning pertaining to graphics.

As GRIP scans each row from left to right retrieving each column with a graphical semantic meaning, the data item retrieved is processed and the interpreter proceeds to the next data item. In this way, GRIP is somewhat immune to different ordering of the columns in the table, extraneous columns and widely separated columns of data. For instance, if the ordering of columns x and y were exchanged in the example data table, or if there had been many other columns intermingled between x, y and xyop, GRIP would still process them correctly. If the xyop column had preceded the x and y columns, the results would have been different as the xyop would have been performed using the previous set of x and y. One

can think of the ordering as the subjects and objects appearing first followed by the verb that acts upon them as in the German language.

Using GRIP

The next several paragraphs will describe the features of the GRIP system. The features of GRIP center around the semantics that have been defined and how to make use of them. The various predefined semantics will be explained and demonstrated with examples. The purpose is to acquaint potential users with what can be done with GRIP and how to accomplish it. To utilize GRIP, the data and structure relations describing the image must first be created in a Mistress database. Figures 2-1 and 2-3 are examples of a typical structure and data relation pair. Figure 2-6 shows a sequence of instructions which may be used to display that image from within a C application program.

```
        /* unless you were already using graphics, you must
         * initialize the CORE graphics package */
Gsetup();

        /* if other than the default window and viewport
        * settings are desired they should go here */
createretainedsegment( 1 );

        /* begin the graphic interpreter looking for
        * table "threads" in database "DB_name"
        * the 'y' indicated that this is the first time
        * Gdraw is called and matrix stack
        * initialization is to be performed. */
status = Gdraw( "DB_name", "threads", 'y' );

        /* and then some clean up */
closeretainedsegment();
Gpause();    /* give the user a chance to see the image */
deleteretainedsegment( 1 );
Gclose();
```

Figure 2-6. *Example using GRIP.*

Features of GRIP

Drawing

The semantics making up the drawing facility are *set the X coordinate*, *set the Y coordinate*, *set the color* and *draw operation*. The first three are self descriptive and only affect the value of some variables that are later used by the *draw operations*. The contents of the columns become the new values of the variables. The values of the *draw operation* column that have been predefined are **move** and **draw**. If you examine the source for the routine *xyoperation* found in Appendix C, you will see that only the

first letter of the column entry is examined. For the **move** operation "**m**", "**M**" and "**1**" are all the same and for the **draw** operation "**d**", "**D**" and "**2**" are all the same. Expanding this collection of operations is a simple matter of adding new cases to the case statement along with the appropriate code to perform the new operation.

Transformations

A semantic meaning of *transform* has been provided. The transformation specification facility is a very general one.

When GRIP begins scanning a row of a table, the current matrix on the top of the matrix stack is duplicated. This provides a working copy of the current instance matrix for the row to make instance modifications to which will affect only that row.

The contents of this column are treated as a sequence of transformations to be applied in the order specified to a temporary identity matrix. After all indicated transformations have been applied, this instance matrix is applied to the matrix currently on the top of the matrix stack. All coordinate conversions use the matrix on the top of the matrix stack to convert to world coordinates. The transformations made are in effect until the completion of the scan of the current row of the table. When finished scanning a row, GRIP checks to see if the top of the matrix stack has been "**modified**" by a transformation, if so the matrix on the top of the stack is replaced by a fresh copy of the original matrix.

The format for the transformation specification string is:

[r(*FLOAT***) | s(***FLOAT,FLOAT***) | t(***FLOAT,FLOAT***)] ***

where *FLOAT* represents any legal floating point number.

There can be any number of rotate, scale and/or translate terms and they may be mixed in any order desired to accomplish any possible transformation. For instance rotating 90 degrees about a point other than the origin, say (x,y), can be accomplished by the following transformation sequence:

t(x,y)r(90)t(-x,-y)

Picture Nesting

When applying the concept of modularity to image design, the ability to structure pictures as collections of other pictures proves to be a very powerful tool. A semantic meaning of sub-picture has been provided to accomplish such tasks. When GRIP encounters a column with a semantic meaning of sub-picture, the character string retrieved from that column is interpreted as the table name for a recursive call to GRIP using the current database. The instance matrix stack is pushed down and the top matrix duplicated. This is done so that any transformations needed by the sub-picture will not affect the rest of the image. GRIP begins interpreting this new table and when completed the matrix stack is popped and interpretation of the current table continues. Pictures may be nested to any level.

Initial vs Subsequent Calls to GRIP

On the first call to GRIP the third argument to Gdraw typically should be a 'y' indicating that the internal instance matrix stack should be cleared. The user may pre-load an initial instance matrix for use by GRIP using the routine Gapplytransform. On subsequent calls to Gdraw,

particularly within an application where the user might already be using graphics the user may not wish to have the matrix stack cleared. Below is an example of a call to Gapplytransform that initializes the instance stack to apply a translate of x=10 and y=20 and scale the picture to twice the size.

```
Gapplytransform( "t(10.0,20.0)s(2,2)");
```

Relation Interpreter Package (RIP)

General Description

The Relation Interpreter Package (RIP) is a generalized application development system for projects utilizing a relational database system for data storage. RIP provides a uniform way to store, retrieve and process data. The database system used in this version of RIP is the Mistress Relational Database Management System [Rh82a] and specifically the MR Host Language Interface [Rh82d].

The MR interface was selected because it provides more user control and feedback than the MX or Standard C Language Interface [Rh82c]. The basic difference is that the MR interface requires a deeper understanding of the data storage and retrieval mechanisms than the other interfaces, but at the same time allows much greater flexibility and optimization of storage and retrieval of data. Although attention has been given to efficient operation, the prime concern has been to create a flexible environment where applications can be developed quickly and simply. The user should not need to learn the language interface to the database; however, a novice understanding of database technology and the concept of organizing data in tables or relations is necessary. A familiarity with query languages is helpful. Figure 3-1 provides an overview of the interpreter.

When the interpreter (RIP) is combined with a set of structure relations, action routines and a main program to invoke and configure RIP, you have the equivalent of a complete application program. The various

components mentioned here will be explained in detail in the rest of this section.

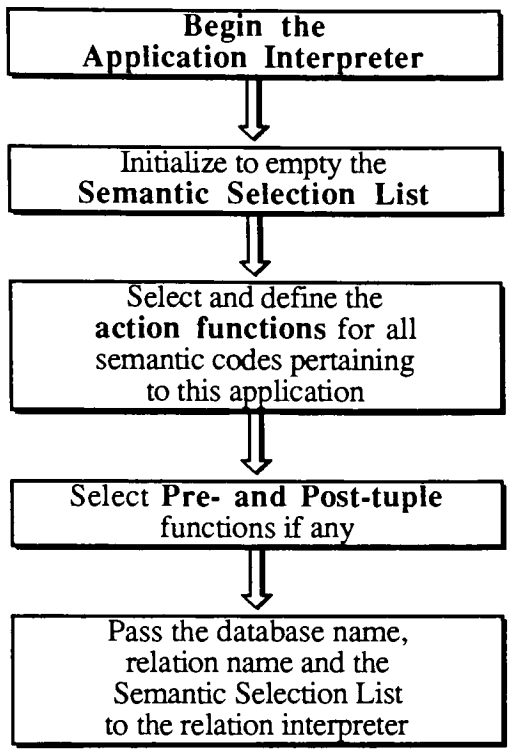


Figure 3-1. Overall structure of interpreter.

Semantics

Tersely, RIP is a table driven relation walker which utilizes semantic definitions of the attributes of a relation as defined in the database. When an application program is developed, some knowledge or assumptions about the syntax and structure of the data are usually made in order to access the data properly. These assumptions are based on information which is not extractable directly from the database itself, but must be retrieved from some external source. This external source could be design specifications or user documentation and is not typically available to the application program.

The program then embodies the designers interpretation of the semantic meaning of the data or how it is to be used. Database management systems generally support the syntax but not the semantic definition of the data.The database management system maintains the knowledge that the "**amount**" column contains only floating point numbers with two decimal digits, but it does not know that it is the dollar value of money orders, for instance. The semantic definition and structure are left to the applications designer to be fixed into the applications programs which use the database. Consequently the programs depend heavily on the assumption that the organization of the data in the database will remain fixed over the program's useful life. Under this assumption changes to the structure of the database must be restricted once applications have been developed which access the database. Those changes could have a ripple effect on all applications related to the data base and involve a great deal of effort just to analyze the impact, let alone make the necessary accommodations

The inability of current database systems to maintain more information than the syntax of the data it holds imposes some limitations on the flexibility of the organization of the database and the application programs that use them. A significant improvement would be the inclusion of semantic information along with the syntactic information kept by the database for each attribute. An application could then retrieve data from a relation by semantic meaning (i.e. miles per hour) rather than just by attribute name or the fourth column of the table. This ability would make the application programs less dependent on the physical organization of the database and relieve some of the restrictions on the organization of the

database . With this improvement the database can be said to be **self-descriptive.**[We76]

> *If semantic information and a description of the syntax is stored along with data in a data base, then that data is said to be **self-describing**.* [We76]

The Structure Relation

Any information intended to be recognized by RIP must be self-describing. Associated with each relation intended to be interpreted by RIP, there exists another relation which describes the structure, syntax, and semantic meaning for each field the user wishes to have recognized. A naming convention for this associated relation, called the **structure relation**, is the name of the data relation prefixed with an underscore (_). For example, if the data relation were named "**cardboard**" then the structure relation would be named "**_cardboard**". This convention is fixed into RIP but could easily be modified should it become necessary. The semantic meanings are represented in the database as an integer code.

The format of all structure relations is the same; the contents which describe the actual layout of the data relations differ. The format of the structure relations is the only fixed database characteristic of the RIP database interface. Figure 3-2 illustrates the Mistress command to display the format of a structure relation and the format of a sample structure relation associated with a data relation named "head". Each row of a structure relation describes an attribute of the associated data relation which has some semantic meaning attached to it. Not all attributes of a data relation must have a semantic meaning. Those that do not will be ignored by the interpreter. The semantic definition of an attribute includes the

relative number of the attribute in the data relation, the name of the attribute, its syntax and an integer code which indicates the semantic meaning to be associated with the field. The syntax and name are redundant in that they are already stored in the database system. They are included here because most database systems, including **Mistress**, do not allow the user to retrieve this information readily.

```
* display table _head;
***  _head  ***

attr_number                    short integer
attr_name                      char  (31, 1)
attr_syntax                    char  (20, 1)
attr_semantics                 short integer
```

Figure 3-2. *Format of a structure relation.*

Configuration of RIP

To interpret a relation, RIP first needs to know which semantic codes are to be recognized for this application and what action is to be take when they are found. The action taken is a call to a C routine with a single argument: a character string containing the contents of the attribute retrieved from the relation. The user is responsible for developing those routines, called **action routines**, specific to his application. To configure RIP, a set of routines are available for selecting the semantic codes to be considered for this application and associating user written **action routines** with each semantic code thereby creating a semantic **SelectionList** . The last chapter of this section describes those routines which the applications developer would use to configure RIP to solve a

specific problem. Not every attribute in the relation need be defined and included in the **SelectionList**.

Selected Attribute Description List

The semantic selection list defines the entire set of semantic codes pertaining to an application. The list also contains the pointers to the functions describing the action to be taken for those semantics. The Semantic Selection List applies to the application in general. Information must be compiled describing the attribute retrieval sequence for the specific relation being interpreted. This is the first step taken by the interpreter when it is called upon to process a specific data relation.

The information is contained in a linked list structure called the **Selected Attribute Description (SAD) List**. The SAD list describes the actual fields to be retrieved from the data relation, their retrieval order, and associates the proper action function to be used. In addition, several data structures required by Mistress for proper attribute retrieval are maintained by the SAD list. Figure 3-3 depicts the SAD List organization.

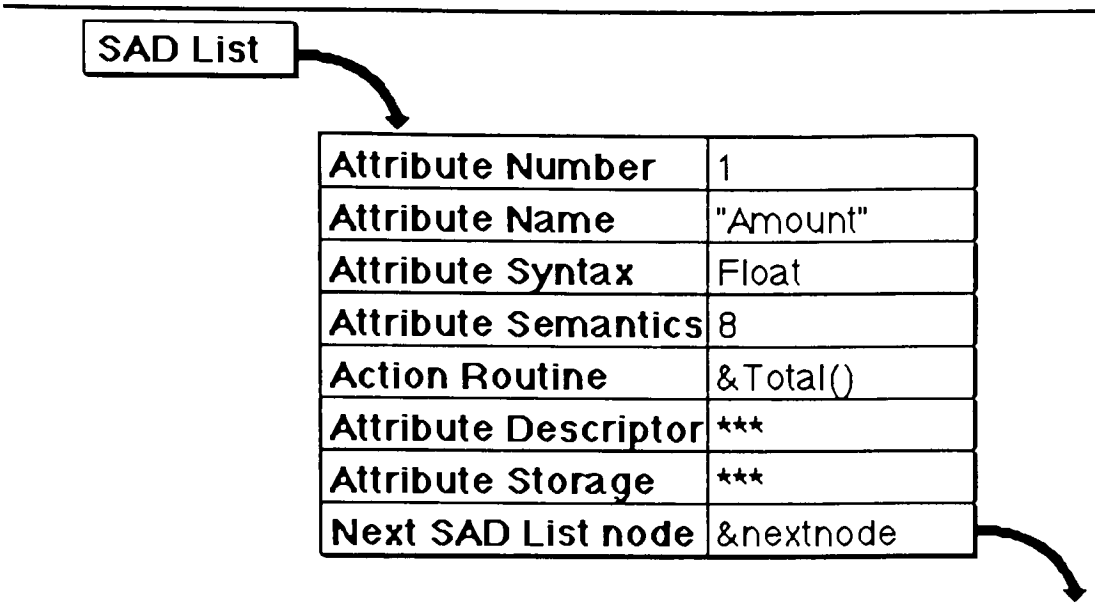


Figure 3-3. SAD List organization.

The *get_structure* routine is responsible for the construction of the SAD List. The *get_structure* routine retrieves the structure relation sorted by attribute number so that the attribute definitions are retrieved and inserted into the SAD list in the same order they are in the data relation. As each attribute definition is retrieved, the semantic code is looked up in the semantic selection list. If the function entry is not NULL, a new node is added to the end of the SAD list representing that field. A link list approach was chosen because it places no predetermined size limitations on the number of fields that could be processed.

Pre- and Post- Tuple Routines

Besides the actions taken for each attribute with semantic meaning, some applications need to be able to specify actions to be taken both before and after the processing of a row or tuple of a relation. Often it is desirable to perform some standard initialization sequence before beginning the scan

of a new tuple and some closing actions after finishing a scan of a tuple. Grip is an example where a pre-tuple routine is specified to prepare the instance matrix on the top of the matrix stack before the row is evaluated. Two routines, SelectPreTuple and SelectPostTuple, are provided to define the routines to be executed before and after scanning each tuple, respectively.

Ordered Relations

Mistress does not directly support ordered relations. The need to support duplicate tuples and maintain some tuple ordering is not uncommon. The graphics application is an example were tuple ordering is important. Therefore, rudimentary capability for tuple ordering has been provided within the interpreter system. This is in the form of a special semantic code of minus one (-1) which can not be defined or undefined by the user. If an attribute of a relation is defined in the structure relation with a semantic code of minus one, the whole relation will be sorted by the contents of that attribute in ascending order before the interpreter begins scanning the tuples. The ordering attribute is typically just a sequential index such as an integer that can indicate the desired order of the rows but it could be something non-integer such as a list of names of presidents of the USA. Because the Mistress sorting facility automatically adapts to the syntax of the attribute being sorted upon, the user is free to choose what ever syntax, integer, float, character, etc., that Mistress provides which best fits the application. The ordering semantic is not optional, if an ordering attribute is found defined for a relation it will be used. If more than one ordering attribute is defined for a relation, the first will be used and all

others will be ignored. If no ordering attribute is defined the rows of the table are retrieved as they are physically stored in the database.

The Interpreter

Once the structure and semantics has been gathered, the interpreter then walks through a relation retrieving only the attributes named in the SAD list in the order specified in the SAD list. One aspect of the design of the interpreter is that different sets of attributes can be defined for different invocations of the interpreter within the same tuple. The attributes will appear to be invisible to an interpreter if their semantic code is not defined in the SAD List that the interpreter is using. In fact, those undefined attributes will not even be retrieved by the interpreter decreasing the tuple retrieval overhead. Since not all attributes need to have a semantic meaning defined for them, those attributes without a semantic meaning are also invisible to all interpreters and not retrieved.

Consequently, the interpreter depends only on the defined semantics for the attributes in a relation; the actual names of the attributes are irrelevant. The retrieval mechanism of Mistress adds an interesting twist to the RIP system. When an attribute is retrieved from a relation, it is represented as a character string regardless of the syntax defined in the database. That means, if a floating point number is stored in the database with a syntax of float, it will be retrieved and converted to a character string representing the floating point value before it is presented to the user. The application is required to convert the character string back into a numeric quantity if that is what is needed. This is convenient if the user was

intending to work with the character representation but introduces considerable overhead if the numeric representation is needed.

Recursive Interpreters

It is possible to establish a hierarchy of relation interpretations. This occurs when an action routine for a semantic includes a request to interpret another relation to complete its action. If the same semantic definitions, i.e. the same interpreter, are to be utilized, the semantic selection need not be redone. The interpreter can be called passing the old semantic selection list since it is general to the application and contains nothing specific to any relation.

If the action is to have the relation interpreted with a new set of semantic definitions, a new semantic selection list can be created and the interpreter called passing this new list, and once completed, resume interpretation of the original relation using the original interpreter semantic definitions. The Graphics Relation Interpreter Package shows an example of recursive interpreters and the Catalog Relation Interpreter Package shows an example of one interpreter calling another interpreter with different semantic definitions.

Pascal Usage

The RIP system is not directly callable from PASCAL because of RIP's usage of pointers to functions. The standard PASCAL language does not support function pointers. It is possible to design application interpreters callable from PASCAL and with action routines written in PASCAL. The semantic selection routine calls must be made within a C

routine and a set of C routines each calling one of the PASCAL action routines is required. This circumvents the problem of manipulating pointers to functions from within PASCAL but complicates the development process and sacrifices the convenience of the tools designed to configure RIP. Appendix E illustrates some examples of main programs which invoke specific configurations of interpreters. The two examples are **draw** and **catalog**.

RIP Routines

The following paragraphs provide an explanation of how to use the routines designed to configure and invoke the interpreter RIP. It has been organized as a user reference.

initSelectionList(List)

Initialize the array of semantic codes to all undefined. Currently the range of available semantic codes is 1 <= code <= 127, but this too can easily be changed. The data structure for List must be created by the user and is an array of pointers to functions returning integer values. The following is an example:

```
typedef int (*PFI) ();  /* pointer to a function returning an integer */
PFI List[ MAXSCODE+1 ]; /* semantic Selection List */
```

SelectSemantic(List, SemanticCode, Function)

Function is a pointer to a routine returning an integer value. **SelectSemantic** enters the **Function** pointer into the **List** of semantic definitions associating it with semantic code **SemanticCode**. Only one

function pointer can be defined for a semantic code at any one time. This is insured because the data structure used for List is an array with the semantic code used as a subscript. If a null pointer is given as Function, this acts as to undefine the semantic code for that application. Typically the **List** of semantic definitions is built by the interpreter when it examines the structure relation. This routine is used if the developer wishes to add or modify a semantic definition.

`SelectPreTuple( Function )`

SelectPreTuple registers the function **Function** as the routine to be executed immediately before beginning each tuple retrieval. If **Function** is NULL, no routine will be called before each tuple scan. There can only be one pretuple routine defined, therefore if there is more than one call to **SelectPreTuple** only the last one will be used.

`SelectPostTuple( Function )`

SelectPostTuple registers the function **Function** in the same manner as **SelectPreTuple** except the function specified is executed immediately after retrieving the last attribute from each tuple. See **SelectPreTuple**.

`interpret(database_name, relation_name, SelectionList)`

Interpret begins the structure relation retrieval and subsequently scans the tuples of the data relation. Figure 3-4 illustrates the sequence of events involved in scanning the data relation and calling the action routines.

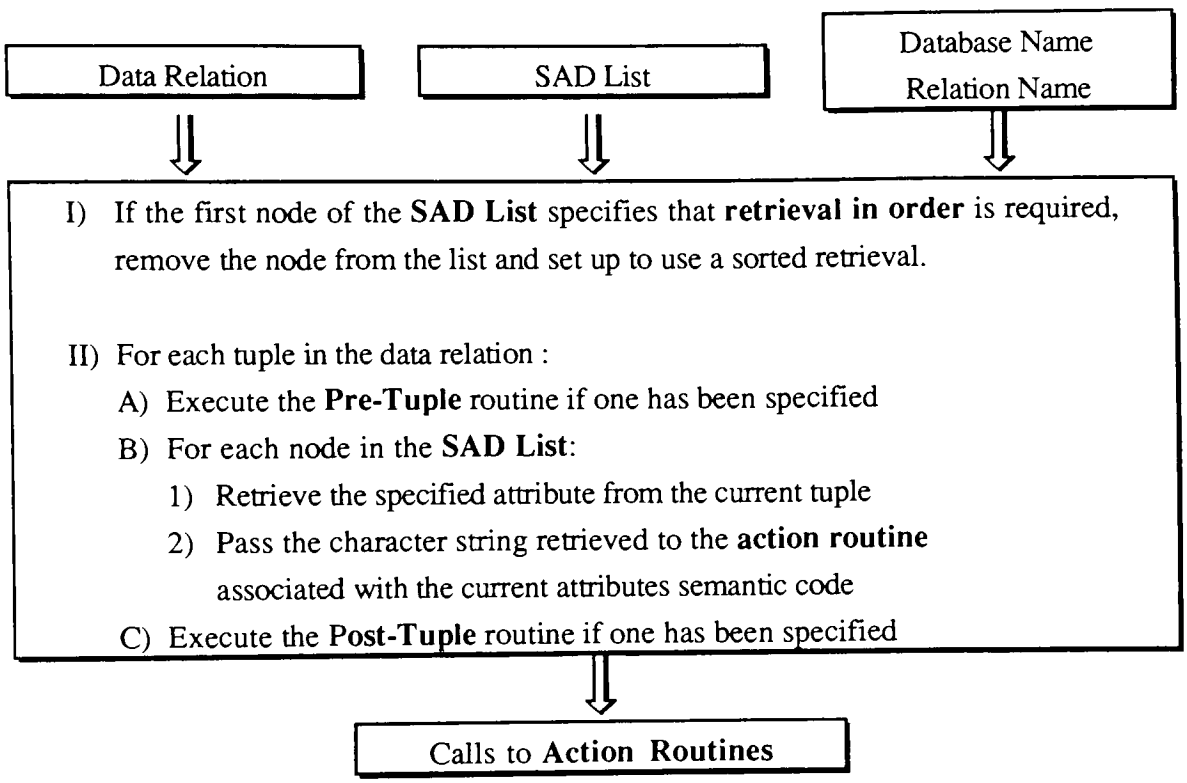


*Figure 3-4. Scanning data relation and calling **action routines***

Catalog Relation Interpreter Package (CRIP)

The Catalog Relation Interpreter Package (CRIP) was developed to illustrate the versatility of the Relation Interpreter Package as well as the ease of use of the Graphics Relation Interpreter Package. CRIP is a system for viewing pages from an online catalog of products. The description of the products is stored in a table in a Mistress database. The description can include the name of the part, the identification number of the part, a brief description and a per unit price. The product description can also specify the name of another table which contains the specification of a picture of the product. CRIP makes use of GRIP to interpret this picture table thereby generating a picture of the product.

CRIP also illustrates how rapid application development can be accomplished through the design of modular interpreters to handle subsets of the primary problem. It would have been inefficient and redundant to have CRIP directly interpret the graphics data. Since the base interpreter (RIP) is general purpose and its function is defined in a data structure, i.e. the Semantic Selection List and the Selected Attribute Description List, having one application interpreter make use of another is not inconsistent at all.

Semantics

The semantic codes defined for CRIP are presented in figure 4-1. The semantic code 100 through 103 simply place the information retrieved

from the tuple at some particular location on the simulated catalog page. Semantic code 104 adjusts the graphic window and viewport to confine the picture of the product to some area on the catalog page. Then GRIP is called to generate the image in that area.

Semantic Code	Semantic Meaning
100	Part identification number
101	Part name
102	Part description (textual)
103	Per unit price
104	Part picture (name of table)

***Figure 4-1**. CRIP defined semantic codes.*

An Example Database for CRIP

Figure 4-2 shows the data relation for a sample catalog called **cat**. **Cat** contains two entries from a small parts catalog. Each tuple of **cat** represents one page from the catalog. The catalog is displayed one page at a time. After the user has viewed a page, the user presses any key and a carriage return to see the next catalog page. Figure 4-3 shows the structure relation associated with catalog **cat**.

part_number	part_name	part_description	part_price	Part_picture
a1232g14-01	ACME best machine screw	High quality hardened steel with machine ground threads	$1.23	bolt
n01-001-01	ACME hex nut	Hardened steel nut with hand machined threads	$0.32	nut

***Figure 4-2**. The **cat** data relation.*

attr_number	attr_name	attr_syntax	attr_semantics
1	part_number	char(10,1)	100
2	part_name	char(30,1)	101
3	part_description	char(60,1)	102
4	part_price	dollar	103
5	part_picture	char(20,1)	104

***Figure 4-3**. The cat structure relation.*

CRIP also illustrates the utility of the **pre-** and **post-tuple** routine specification.

Page_begin

is specified as the pre-tuple routine and performs the function of setting up the default window limits and creating the initial segment for each page.

Page_hold

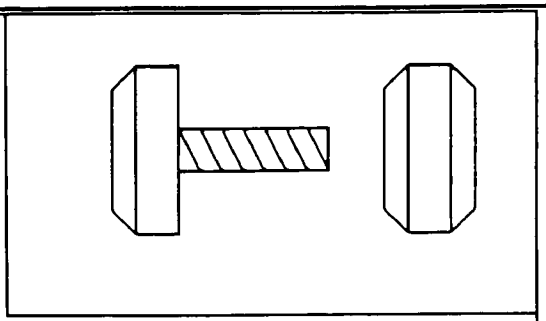
is set up as the post-tuple routine and performs a pause until key pressed operation to hold the page on the graphics screen. Once a key has been pressed page_hold goes on to delete all segments created specifically for that page.

To begin the Catalog Relation Interpreter Package on the **cat** relation, the following Unix command is entered:

```
catalog DATABASE_NAME cat
```

The results of which are shown in figure 4-4.

Part #: a123g14-01
Name: ACME best machine screw
Single unit price: $1.23



Description:
High quality hardened steel with machine ground threads.

Figure 4-4. First page from catalog cat.

Conclusions

The main thrust of this study was to examine how a relational database system could be utilized for picture storage and retrieval. Four main goals were established for the study.

- Incorporation of Graphics into an Application
- Image Maintenance
- Structuring of Images
- Association between Graphical and Non-Graphical Data

The system described herein provides a flexible and easy to adapt interface to a relational database system. The Graphics Relation Interpreter Package (GRIP) has illustrated that a relational database can provide a manageable and effective storage medium for graphical data. The **catalog** and **draw** programs illustrate the simplicity of including graphics in an application through the GRIP facility. And the use of tables to store the picture descriptions in relational databases is a form already familiar to many users. This makes the use of a facility such as GRIP more likely by the typical user.

The sub-picture semantic of GRIP illustrates how picture structuring can be accomplished. This provides an object oriented approach to picture development and facilitates the production and use of libraries of predefined images such as windows and other building materials, circuit components, furniture and hosts of other collections.

CRIP illustrates that a system can be developed that would allow graphical data to co-inhabit a relational database with non-graphical information. The catalog example shows how non-graphical information such as price and description could be included in the same database, in deed the same relation, as the picture description.

Efficiency vs. Flexibility

During the design of the Relation Interpreter Package and the Graphics Relation Interpreter Package, care was taken to make the system as time and space efficient as possible. But efficiency was not the main concern. The foremost goal was to provide a flexible environment for image storage and retrieval. Wherever these two goals came into conflict, flexibility won out. This tends to follow the philosophy of relational database systems, i.e. provide greater access to the data at the expense of some speed and storage.

All direct references to the database management system, **Mistress**, are confined to the Relational Interpreter Package (RIP). RIP can provide the basic building blocks for a whole family of application programs and interpreters, such as GRIP and CRIP. This structure insulates the applications programs from the actual structure and syntax of the database system being used to be changed and the only programming changes required would be to the underlying RIP package. The application interpreter packages would need only be recompiled.

Future Enhancements

The Graphics Relation Interpreter Package provides a simple mechanism for the incorporation of graphics into applications. The Relation Interpreter Package also provides a simple facility for creating new application interpreters. To complete the Image Storage System and fulfill the goal of making not only user access to graphical images easy but user generation of images equally as simple, a facility for efficient and natural image specification is needed. An object oriented drawing tool which would generate the specification of the image directly into a relation would be of great assistance.

Future enhancements to GRIP should include expanding the defined semantics to include 3-d capability and bit-mapped operations and the addition of controls to allow the user to determine to what depth nested pictures are to be resolved. This would allow the images to be displayed with varying degrees of detail. An example would be blueprints of a building. One level of detail would be the landscape layout which would position the building. The next level would layout the foundation and superstructure. After that successive levels could give greater and greater detail of the design and construction.

A windowing package on the order of CURSES [Ar00] or MWindow [To83] only designed to allow both textual and graphical information to be generated to graphic terminals should be developed and integrated. In conjunction with GRIP, this graphic window package would provide an even more powerful application development tool.

All in all, GRIP has provided a tool which a novice programmer/user can more easily include graphics in an application and maintain the application image over time.

Bibliography

Ab80 Abhyankar, R.B.; Kashyap, R.L., "Pictorial Data Description and Retrieval With Relational Languages," **Proceedings of the Workshop on Picture Data Description and Management**, August 1980, pp. 57-60.

Ar00 Arnold, K.C., "Screen Updating and Cursor Movement Optimization: A Library Package," **Department of Electrical Engineering and Computer Science**, University of California, Berkley.

Be79 Becerril, J.L.; Casajuana, R.; Lorie, R.A., "GSYSR: A Relational Database Interface for Graphics," **Data Base Techniques for Pictorial Applications**, June 1979, pp. 459-74.

Ch80a Chang, N.S.; Fu, K.S., "A Relational Database System for Images," **Pictorial Information Systems**, Springer-Verlag, Berlin, Germany, 1980, pp. 288-321.

Ch80b Chang, S.K.; Lin, B.S.; Walser, R., "A Generalized Zooming Technique for Pictorial Database System," **Pictorial Information Systems**, Springer-Verlag, Berlin, Germany, 1980, pp. 257-87.

Ch77 Chang, S.K.; Donato, N.; McCormick, B.H.; Reuss, J.; Rocchetti, R., "A Relational Database System for Pictures," **Proceedings of the Workshop on Picture Data Description and Management**, April 1977, pp. 142-9.

Da77 Date, C.J., **An Introduction to Database Systems**, Addison-Wesley Publishing Company, 1977.

Fe78 Ferrin, Thomas; Langridge, Robert, "Interactive Computer Graphics with the UNIX Time-Sharing System," , pp. 320-328.

Fr82 Frank, A., "MAPQUERY: Database Query Language for Retrieval of Geometric Data and their Graphical Representation," **Computer Graphics**, Vol. 16, No. 3, July 1982, pp. 199-207.

Ha81 Hardwick, M., "Graphical Data Strucures," **Computer Graphics (USA)**,Vol. 15, No. 4, Dec. 1981, pp. 376-404.

Jo75 Joyce, J.D.; Oliver, N.A., "Prelimiary User's Manual for REGIS Information System," **GMC Research Report GMR-2008, Research Laboratories**, October 1975.

Jo76 Joyce, J.D.; Oliver, N.A., "REGIS - A Relational Database Manager with Graphics and Statistics," **GMC Research Report GMR-2009, Research Laboratories**, January 1976.

Ka81 Kalay, Y. E., "Interactive Shape Generation and Spatial Conflict Testing," **ACM IEEE Eighteenth Design Automation Conference Proceedings**, 1981, pp. 75-81.

Ke78 Kernighan, B.W.; Ritchie, D.M., **The C Programming Language**, Prentice-Hall, Inc., 1978.

Ko81 Koffman, E.B., **Problem Solving and Structured Programming in PASCAL**, Addison-Wesley Publishing Company, 1981.

Ku74 Kunii, T.S.; Weyl, S.; Tenebaum, J.M., "A Relational Database Schema for Describing Complex Pictures with Color and Texture," **Proc. of the Second International Joint Conference on Pattern Recognition**, Lyngby-Copenhagen, Denmark, August 1974.

Li Licklider, J.C.R., "User-Oriented Interactive Computer Graphics," **User-Oriented Design of Interactive Graphics Systems**, ACM, New York, NY, pp. 89-96.

Li80 Lin, B.S.; Chang, S.K., "GRAIN A Pictorial Database Interface," **Proceedings of the Workshop on Picture Data Description and Management**, August 1980, pp. 83-88.

Lo79 Lorie, R.A.; Casajuana, R.; Becerril, J.L., "GSYSR: A Relational Database Interface for Graphics," **IBM Research Report RJ2511, Computer Science**, April 1979.

Mc77 McKeown, D.M., Jr.; Reddy, D.R.; "A Hierarchical Symbolic Representation for an Image Database," **Proceedings of the Workshop on Picture Data Description and Management**, April 1977, pp. 40-4.

Mc82 McKeown, D. M.; Denlinger, J. L., "Graphical Tools for Interactive Image Interpretation," **Computer Graphics**, Vol. 16, No. 3, July 1982, pp. 189-198.

Me77 Meder, H.G.; Palermo, F.P., "Data Base Support and Interactive Graphics," **Proc. Third Int. Conf. on Very Large Data Bases**, Oct. 1977, pp. 390-402.

Mi79 Miller, I.M., "Data Bases and Structures for a Gamut of Graphic Application Abstract," **Data Base Techniques for Pictorial Applications**, June 1979, pp. 441-457.

Ne81 Neely, Joel; Stewart, Steve, "Fundamentals of Relational Data Organization," **BYTE**, Vol. 6, No. 11, November 1981, pp. 48-60.

Ng79 Ng, N., "A Graphical Editor For Programming Using Structured Charts," **Exploding Technology/Responsible Growth: COMPCON 79**, San Francisco, February 1979, pp. 238-43.

Ng78 Ng, N., "A Graphical Editor for Programming Using Structured Charts," **IBM Research Report RJ2344, Computer Science**, September 1978.

Ne79 Newman, W.M., **Principles of Interactive Computer Graphics**, McGraw-Hill, Inc., 1979.

Pa79a Palermo, F.; Weller, D., "Picture Building System," **Exploding Technology/Responsible Growth: COMPCON 79**, San Francisco, February 1979, pp. 235-7.

Pa79b Palermo, F.; Weller, D., "Picture Building System," **IBM Research Report RJ2436, Computer Science**, 1979.

Pa79c Palermo, F.; Weller, D., "Some Database Requirements for Pictorial Applications," **Data Base Techniques for Pictorial Applications**, June 1979, pp. 555-567.

Ph79 Phillips, R.J.; Beaumont, M.J.; Richardson, D., "AESOP: An Architectural Relation Database," **Computer Aided Design (GB)**, Vol. 11, No. 4, July 1979, pp. 217-26.

Rh82a **Mistress: Relational Datahase Management System**, Rhodnius, Inc., 1982.

Rh82b **Mistress: The Query Language**, Rhodnius, Inc., 1982.

Rh82c **Mistress: The Host Language Interface**, Rhodnius, Inc., 1982.

Rh82d **Mistress: The MR Routines**, Rhodnius, Inc., 1982.

Sh79 Sharman, G.C.H., "A Picture Drawing System Using A Binary Relational Database," **Data Base Techniques for Pictorial Applications**, June 1979, pp. 495-508.

Ta79 Takao, Y.; Itoh, S.; Iisaka, J., "An Image-Oriented Database System," **Data Base Techniques for Pictorial Applications**, June 1979, pp. 527-38.

Te77 Teitelman, W., "A Display Oriented Programmer's Assistant," **Tech. Report CSL77-3**, Xerox Palo Alto Research Center, 1977.

To83 Torek, C., "The Maryland Window Library," **Department of Computer Science**, University of Maryland, 1983.

Un79 Uno, S.; Matsuka, H., "A General Purpose Graphics System for Computer Aided Design," **Computer Graphics**, Vol. 13, No. 2, 1979, pp. 25-32.

Un80 Uno, S.; Matsuka, H., "A Relational Database for Design Aids System," **Proceedings of the Workshop on Picture Data Description and Management**, August 1980, pp. 89-94.

We79a Weller, D.; Carlson, E.; Giddings, G.; Palermo, F.; Williams, R.; Zilles, S., "Software Support for Graphical Interaction," **IBM Research Report RJ2670, Computer Science**, October 1979.

We79b Weller, D.W.; Palermo, F.P., "Database Requirements for Graphics," **Exploding Technology/Responsihle Growth: COMPCON 79**, San Francisco, February 1979, pp. 231-4.

We79c Weller, D.W.; Palermo, F.P., "Database Requirements for Graphics," **IBM Research Report RJ2435, Computer Science**, January 1979.

We76 Weller, D.L.; Williams, R., "Graphic and Relational Database Support for Problem Solving," **Computer Graphics**, Vol. 10, No. 2, July 1976, pp. 183-9.

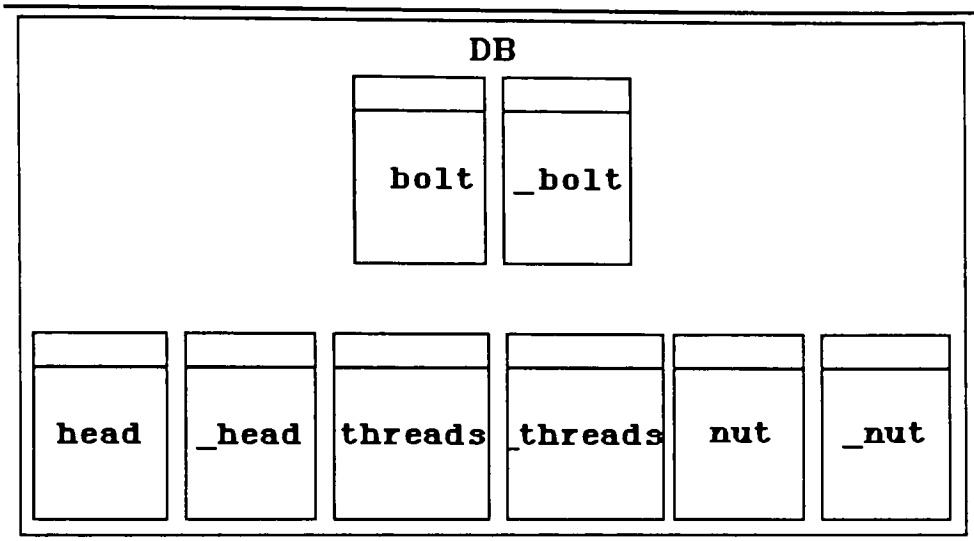
Wi74 Williams, R., "On the Application of Relational Data Structures in Computer Graphics," **Proceedings of IFIP Congress '74**, Stockholm, Sweden, pp. 722-6.

Wi71 Williams, W.G., "A Survey of Data Structures for Computer Graphics Systems," **Computing Surveys, ACM**, Vol. 3, No. 1, March 1971, pp. 1-21.

Appendices

Appendix A - A Sample GRIP Database

The following are the database contents for a sample image. The image is of a bolt composed of three parts: a head, threaded shaft and a nut. This example illustrates many of the features of GRIP.



Overview of the database layout

```
*** DB ***

_bolt
_head
_nut
_shaft
bolt
head
nut
shaft
```

A List of the Tables in Database DB

attr number	attr name	attr syntax	attr semantics
1	color	char(7,1)	12
2	part	char(20,1)	11

The Structure Relation _bolt

color	part
red	head
yellow	shaft
blue	nut

The Data Relation bolt

attr number	attr name	attr syntax	attr semantics
1	x	float	8
2	y	float	9
3	xyop	integer	10

The Structure Relation _head

x	y	xyop
2.000000e+00	5.000000e-01	1
1.000000e+00	1.500000e+00	2
1.000000e+00	4.500000e+00	2
2.000000e+00	5.500000e+00	2
3.000000e+00	5.500000e+00	2
3.000000e+00	5.000000e-01	2
2.000000e+00	5.000000e-01	2
2.000000e+00	5.500000e+00	2

The Data Relation head

attr number	attr name	attr syntax	attr semantics
1	x	float	8
2	y	float	9
3	xyop	integer	10

The Structure Relation _nut

x	y	xyop
7.500000e+00	5.500000e+00	1
7.000000e+00	5.000000e+00	2
7.000000e+00	1.000000e+00	2
7.500000e+00	5.000000e-01	2
8.500000e+00	5.000000e-01	2
9.000000e+00	1.000000e+00	2
9.000000e+00	5.000000e+00	2
8.500000e+00	5.500000e+00	2
7.500000e+00	5.500000e+00	2
7.500000e+00	5.000000e-01	2
8.500000e+00	5.500000e+00	1
8.500000e+00	5.000000e-01	2

The Data Relation nut

attr number	attr name	attr syntax	attr semantics
1	x	float	8
2	y	float	9
3	xyop	char(4,1)	10

The Structure Relation _shaft

x	y	xyop
3.000000e+00	2.500000e+00	move
6.000000e+00	2.500000e+00	draw
6.000000e+00	3.500000e+00	draw
3.000000e+00	3.500000e+00	draw
3.500000e+00	2.500000e+00	draw
3.500000e+00	3.500000e+00	move
4.000000e+00	2.500000e+00	draw
4.000000e+00	3.500000e+00	MOVE
4.500000e+00	2.500000e+00	DRAW
4.500000e+00	3.500000e+00	Move
5.000000e+00	2.500000e+00	Draw
5.000000e+00	3.500000e+00	move
5.500000e+00	2.500000e+00	draw
5.500000e+00	3.500000e+00	move
6.000000e+00	2.500000e+00	draw

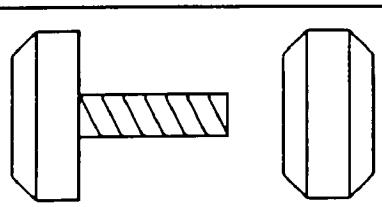
The Data Relation shaft

The following image can be generated using the **draw** program from Appendix E with the Unix command:

draw "DB" "bolt"

or from with in a C program with:

DrawPicture("DB", "bolt");



The Image Produced from Table bolt

Appendix B - Source Code for RIP

File name: rip.h

```
/* ------------------------------------------------------------
 *
 * rip.h
 *   RIT Interpreter Package for Self-Describing Mistress Databases
 *   program header
 *
 ------------------------------------------------------------
*/

#include <mscc.h>
#include <stdio.h>
#include <signal.h>

/* Constant definitions */
#define    READONLY    'r'
#define    ASCENDING   'a'
#define    NOMOREARGS  0
#define    NOQUALIFIER 0
#define    MINSCODE    0
#define    MAXSCODE    128
#define    TRUE        1
#define    FALSE       0
#define    PRIVATE     static

/* type definitions */

typedef int (*PFI) ();          /* Pointer to a function returning an integer
                                */

typedef struct anode {          /* Selected Attribute Description List(SAD) */
      int number;               /* attribute number in relation */
      char *name;               /* attribute name in relation */
      char *syntax;             /* format of the attribute */
      int semantics;            /* semantics code for this attribute */
      PFI Function;             /* function to be used for this attr */
      addr desc;                /* attribute desciptor used by Mistress */
      addr value;               /* retrieved value storage for Mistress */
      struct anode *next;             /* pointer to next node in this list */
} SAD_NODE, *SAD_PTR;


/* Routine definitions */
char *copystr();
```

File name: rip/int.c

```
/* ---------------------------------------------------------------
-
 *
 * rip
 *    Relation Interpreter Package for Self-Describing Mistress
 *    Databases.  These are the utility routines for use in writing
 *    any application interpreter
 *
 ---------------------------------------------------------------
*/

#include "rip.h"

#ifdef RDEBUG
int Rdebug = 1;    /* set to 1 for debug output */
#endif

/* ---------------------------------------------------------------
-
 *
 * interpret
 *    Using the list of functions for selected attributes, determine
 *    which exist in the selected relation (SADLIST) and scan the
 *    data relation retrieving the selected attribute and calling
 *    the associated function.
 *
 *---------------------------------------------------------------
*/

int
interpret( database_name, relation_name, List )
char *database_name;     /* Database name to look for relation in */
char *relation_name;     /* Relation to be interpreted */
PFI List[];              /* array of functions for selected attributes */
{
      SAD_PTR get_struct();
      SAD_PTR SADList;

      /* create the Selected Attribute Description list from the
       * structure relation.
       */
      SADList = get_struct( database_name, relation_name, List );

      if ( (NULL <= ((int) SADList)) && (((int) SADList) <= MRERDUP) )
      {
            /* oops, had problems getting structure definition */
#ifdef RDEBUG
            if (Rdebug)
                  printf("Rip: get_struct returned SADList=%d", SADList );
#endif
            return( (int) SADList );
      }
```

```
        return( scan_relation( database_name, relation_name, SADList, List[0],
                List[MAXSCODE] ));
}
```

File name: rip/scan_rel.c

```
/* ------------------------------------------------------------------
-
 *
 *scan_relation
 *    Fetch the attribute values specified in the Selected Attribute
 *    Description list from the data relation and pass these values
 *    to the selected function.
 *
 ------------------------------------------------------------------
*/

#include "rip.h"

int
scan_relation( DB_name, REL_name, List, PreTuple, PostTuple )
char *DB_name;          /* Database name to look for relation in */
char *REL_name;         /* Relation name */
SAD_PTR List;           /* pointer to list of selected attributes */
PFI PreTuple;           /* function to be called before tuple scan */
PFI PostTuple;          /* function to be called after tuple scan */
{
addr REL_desc, REL_record;
addr qualifier;
int ordered;                    /* flag - is this relation ordered */
addr ordered_desc;

SAD_PTR Temp;

      /* open data relation    */
      REL_desc = mrtopen( DB_name, REL_name, READONLY );

      /* Did we find and open the relation OK    */
      switch (REL_desc) {
            case MRERDBEXIST:
            case MRERDBACCESS:
            case MRERNAME:
            case MREREXIST:
            case MRERACCESS:
            case MRERNULL:
            case MRERDUP:
                  return ( (int) REL_desc );
                  break;
            default:
                  break;
            }
```

```
/* create necesary descriptor pointers to access relation */
REL_record = mrmkrec ( REL_desc );

/* is there an ordering field specified in the selection list */
if (List->semantics == -1) {
      /* set up for ordered retrieval */
      ordered_desc = mrngeta( REL_desc, List->name );
      ordered = TRUE;
      Temp = List->next;
}
else {
      ordered = FALSE;
      Temp = List;
}

while (Temp != NULL)
{
      Temp->desc = mrngeta( REL_desc, Temp->name );
      Temp->value = mrspv( Temp->desc );
      Temp = Temp->next;
}

/* set up retieval of data relation -?-  sorted by sequence field */
if (ordered) {
      qualifier = mrsrtbegin ( NOQUALIFIER, 's', REL_record, CHARNIL,
                        ordered_desc, ASCENDING, NOMOREARGS );
}
else {
      qualifier = mrgetbegin ( NOQUALIFIER, REL_record, CHARNIL);
}

while ( mrget ( qualifier ) )
{
      /* call pre-tuple function if any */
      if (PreTuple != NULL)
            (*PreTuple)();

      /* scan next tuple from relation */
      if (ordered)
            Temp = List->next;       /* skip ordering node */
      else
            Temp = List;             /* begin with first node */
      while (Temp != NULL)
      {
            mrgetv ( REL_record, Temp->desc, Temp->value );
            (*Temp->Function)(Temp->value);
            Temp = Temp->next;
      }

      /* call post-tuple function if any */
      if (PostTuple != NULL)
            (*PostTuple)();
}

mrgetend ( qualifier );

/* free descriptor pointers */
if (ordered)
```

```
            Temp = List->next;          /* skip ordering node */
        else
            Temp = List;                /* begin with first node */
        while (Temp != NULL)
        {
            mrfree( Temp->value );
            Temp = Temp->next;
        }

        mrfrrec ( REL_record );

        mrclose ( REL_desc );

        return ( 0 );
}
```

File name: rip/get_struct.c

```
/* ------------------------------------------------------------------
-
 *
 * get_struct
 *    Determine list of attribute fields to be read from data
 *    relation.
 *
 ------------------------------------------------------------------ */

#include "rip.h"

SAD_PTR
get_struct( DB_name, REL_name, SelectionList )
char *DB_name;          /* Database name to look for relation in */
char *REL_name;         /* Relation name */
PFI SelectionList[];    /* pointer to list of selected semantics */

{
      char S_REL_name[32];
      addr S_REL_desc, S_REL_record;
      addr A_num_desc,  A_nam_desc,  A_syn_desc,  A_sem_desc;
      addr A_num_value, A_nam_value, A_syn_value, A_sem_value;
      addr qualifier;
      int sem_value;

      SAD_PTR List;           /* new Selected Attribute Description list */
      SAD_PTR Last;           /* pointer to the last node in the list */
      SAD_PTR Temp;

      SAD_PTR newSADnode();

      S_REL_name[0] = '_'; S_REL_name[1] = '';

      /* create structure relation name from data relation name   */
      strcat( S_REL_name, REL_name );

      /* open structure relation    */
      S_REL_desc = mrtopen( DB_name, S_REL_name, READONLY );

      /* Did we find and open the relation OK   */
      switch (S_REL_desc)     {
            /* known mistress error codes */
            case NULL:
            case MRERDBEXIST:
            case MRERDBACCESS:
            case MRERNAME:
            case MREREXIST:
            case MRERACCESS:
            case MRERNULL:
            case MRERDUP:
                  return ( (SAD_PTR)  S_REL_desc );
                  break;
            default:
                  break;
            }
```

```
/* initialize the SAD list */
List = NULL;
Last = NULL;

/* create necesary descriptor pointers to access relation */
S_REL_record = mrmkrec ( S_REL_desc );

A_num_desc = mrngeta ( S_REL_desc, "attr_number" );
A_nam_desc = mrngeta ( S_REL_desc, "attr_name" );
A_syn_desc = mrngeta ( S_REL_desc, "attr_syntax" );
A_sem_desc = mrngeta ( S_REL_desc, "attr_semantics" );

A_num_value = mrspv ( A_num_desc );
A_nam_value = mrspv ( A_nam_desc );
A_syn_value = mrspv ( A_syn_desc );
A_sem_value = mrspv ( A_sem_desc );

/* set up retieval of structure relation sorted by attribute number */
qualifier = mrsrtbegin ( NOQUALIFIER, 's', S_REL_record, CHARNIL,
                  A_num_desc, ASCENDING, NOMOREARGS );
while ( mrget ( qualifier ) )
{
      mrgetv ( S_REL_record, A_sem_desc, A_sem_value );

      sem_value = atoi( A_sem_value );
      if ( 0<sem_value && sem_value<MAXSCODE ) {
            /* semantic value is within normal limits */
            /* is this attribute pertinent to the application at hand */
            if ( SelectionList[ sem_value ] != NULL )
            {
                  mrgetv ( S_REL_record, A_num_desc, A_num_value );
                  mrgetv ( S_REL_record, A_nam_desc, A_nam_value );
                  mrgetv ( S_REL_record, A_syn_desc, A_syn_value );

                  /* create and fill new node for list */
                  Temp = newSADnode();
                  Temp->number = atoi( A_num_value );
                  Temp->name = copystr( A_nam_value );
                  Temp->syntax = copystr( A_syn_value );
                  Temp->semantics = sem_value;
                  Temp->Function = SelectionList[ sem_value ];
                  Temp->desc = NULL;
                  Temp->next = NULL;

                  /* install new node at the end of the SAD list */
                  if (List == NULL)
                  {
                        /* this is the first node */
                        List = Temp;
                        Last = Temp;
                  }
                  else
                  {
                        /* this is not the first node */
                        Last->next = Temp;
                        Last = Temp;
                  }
            }
```

```
            }
            /* is this an field for sequencing the tuples */
            else if ( sem_value == -1 ) {
                  /*ordering field */
                  /* ignore it if it is not the first one found */
                  if ( List == NULL || List->semantics != -1 ) {
                        /* this is the first one found */
                        /* retrieve the rest of the tuple and */
                        mrgetv ( S_REL_record, A_num_desc, A_num_value );
                        mrgetv ( S_REL_record, A_nam_desc, A_nam_value );
                        mrgetv ( S_REL_record, A_syn_desc, A_syn_value );

                        /* create and fill new node for list */
                        Temp = newSADnode();
                        Temp->number = atoi( A_num_value );
                        Temp->name = copystr( A_nam_value );
                        Temp->syntax = copystr( A_syn_value );
                        Temp->semantics = sem_value;
                        Temp->Function = SelectionList[ sem_value ];
                        Temp->desc = NULL;
                        Temp->next = NULL;

                        /* install ordering node at the beginning of list */
                        if ( List == NULL ) {
                              List = Last = Temp;
                        }
                        else {
                              Temp->next = List;
                              List = Temp;
                        }
                  }
                  /* else ignore this semantic field */
#ifdef RDEBUG
                  else if (Rdebug)
                        fprintf(stderr,"get_struct: illegal semantic code %d
found in DB %s, relation %s",
                              sem_value, DB_name, REL_name );
#endif
            }
      }

      mrgetend ( qualifier );

      /* free descriptor pointers */
      mrfree ( A_sem_value );
      mrfree ( A_nam_value );
      mrfree ( A_num_value );

      mrfrrec ( S_REL_record );

      mrclose ( S_REL_desc );

      return ( List );
}

/* create a new SAD list node */
static SAD_PTR
newSADnode()
```

```
{
      return( (SAD_PTR) malloc( sizeof( SAD_NODE )));
}

/* print out the contents of the SAD list */
void
printSAD( List )
SAD_PTR List;
{
      printf("");
      printf("Selected Attribute Description List (SAD)");

      while (List != NULL)
      {
            printf("%5d %s %s %d 0x%x 0x%x",
                  List->number,
                  List->name,
                  List->syntax,
                  List->semantics,
                  List->Function,
                  List->desc);

            List = List->next;
      }
      printf("");
}

char *
copystr( string )
char *string;
{
      char *temp;
      char *calloc();

      temp = calloc( strlen( string )+1, sizeof( char ));
      strcpy( temp, string );
}
```

File name: rip/select.c

```
/* ------------------------------------------------------------------
 *
 * select.c
 *   manage the semantic selection list
 *
 ------------------------------------------------------------------
-
 *
 * void initSelectionList( SelectionList )
 *                          Empty the semantic selection list
 *
 * int SelectSemantic( SelectionList, SemanticCode, Function )
 *                          add another selection to the list
 ------------------------------------------------------------------
*/

#include "rip.h"

#ifdef SDEBUG
int Sdebug = 0;    { set to 1 for debug output }
#endif

void
initSelectionList( SelectionList )
PFI SelectionList[];
{
      int index;

      /* Empty the selection list by setting each element of the array
       * of pointers to functions to NULL.
       * An array is used because the eventual use of this information
       * will be to look up the function pointer by semantic code.
       */
      for (index=0; index<=MAXSCODE ; index++)
      {
            SelectionList[ index ] = (PFI) NULL ;
      }

      return;
}

/*------------------------------------------------------------------
*
 *
 * SelectSemantic
 *   add a semantic code to the Semantic Selection List
 *   and also specify the function to be called for that
 *   semantic attribute.
 *
 *         ERROR CODES:
 *            1 - Semantic Code <= MINSCODE or >= MAXSCODE
 *            2 - Function pointer = NULL
 *
```

```
 ------------------------------------------------------------------
*/
int
SelectSemantic( SelectionList, SemanticCode, Function )
PFI SelectionList[];
int SemanticCode;
PFI Function;
{
      int Done;

      if (SemanticCode <= MINSCODE || SemanticCode >= MAXSCODE)
      {
#ifdef SDEBUG
            if (Sdebug) fprintf(stderr, "SelectSemantic: code %d returned %d",
                        SemanticCode, 1 );
#endif
            return( 1 );
      }

      /* Using the Semantic code as the index place the function pointer
       * to be executed for that semantic in the selection array.
       * If a duplicate entry is found replace it with the new selection
       * function, no error is generated.
       */

      SelectionList[ SemanticCode ] = Function;

      return( 0 );
}

/*------------------------------------------------------------------
 *
 * SelectPreTuple
 *    define the function to be executed before each tuple is scaned
 *
 ------------------------------------------------------------------
*/

void
SelectPreTuple( SelectionList, Function )
PFI SelectionList[];
PFI Function;
{
      SelectionList[0] = Function;
}

/*------------------------------------------------------------------
 *
 * SelectPostTuple
 *    define the function to be executed after each tuple is scaned
 *
 ------------------------------------------------------------------
*/

void
SelectPostTuple( SelectionList, Function )
PFI SelectionList[];
```

```
PFI Function;
{
      SelectionList[MAXSCODE] = Function;
}

/*-----------------------------------------------------------------
 *
 * PrintList
 *    a debugging tool for the selection list
 *
 -----------------------------------------------------------------
*/
void
printList( SelectionList )
PFI SelectionList[];
{
int index;

      printf("");
      printf("Listing of all non-NULL entries in the Selection List");
      printf("Semantic Code     Function Address");
      printf("===============   ===============");

      for (index=0; index<=MAXSCODE; index++)
      {
            if ( SelectionList[ index ] != NULL )
            {
                  printf("%-18d0x%x", index, SelectionList[ index ]);
            }
      }

      printf("");
}
```

File name: rip/makefile

```
# makefile
#
#     RIT Interpreter Package for Self-Describing Mistress Databases
#

PRINTER = rpr
CFILES = int.c scan_rel.c get_struct.c select.c
CHDRS = rip.h
OBJS = int.o scan_rel.o get_struct.o select.o
LIBACCT = /acct/f1/dgj/thesis/prog/interp/

#-----------------------------------------------------------------------


riplib:     $(LIBACCT)rip.h $(OBJS)
      ar rucv riplib $(OBJS)
      chmod 755 riplib
      ranlib riplib

.c.o: ;
      mscc -g -c $*.c

update:     $(LIBACCT)rip.h $(OBJS)
      rm riplib
      ar rucv riplib $(OBJS)
      chmod 755 riplib
      cp riplib $(LIBACCT)riplib
      ranlib $(LIBACCT)riplib
      ranlib riplib


$(LIBACCT)rip.h: rip.h
      cp rip.h $(LIBACCT)rip.h

#-----------------------------------------------------------------------

print :
      cpr -h $(CHDRS) $(CFILES) | $(PRINTER)
      pr makefile | $(PRINTER)
```

Appendix C - Source Code for GRIP

File name: grip.h

```
#include "rip.h"

typedef
      struct matrixn {        /* transformation matrix      */
            double a,b,c,d,e,f;           /* truncated matrix elements */
            int contaminated;       /* flag indicating if matrix has been
                                    changed */
            struct matrixn *last;   /* backwards pointer along stack */
      } MATRIX_NODE, *MATRIX_PTR;

/* routine definitions */
MATRIX_PTR G_ident();
MATRIX_PTR G_matmult();
MATRIX_PTR G_DeleteTopOf();
MATRIX_PTR G_DuplicateTopOf();
void G_ReduplicateTopOf();
void G_matprint();
```

File name: grip/Gdraw.c

```
/*-----------------------------------------------------------------------
 *
 * Gdraw.c
 *
 *-----------------------------------------------------------------------
*/

#include <math.h>
#include "grip.h"

PRIVATE     char *database_name;
PRIVATE     PFI List[ MAXSCODE+1 ];
PRIVATE     float next_x, next_y;
PRIVATE     MATRIX_PTR MatrixStack = NULL;

int x_coord(), y_coord(), xy_operation(), sub_picture(), Set_Color();
int G_applytrans();
int G_ready();

#ifdef GDEBUG
int Gdebug = 1;    /* set to 1 for debug output */
#endif

int
Gdraw( DB_name, relation_name, initialize )
char *DB_name;
char *relation_name;
char initialize;
{
      int status;

      database_name = DB_name;

      /* define semantic meanings to be used by the interpreter for
       * the Graphics application.
       */
      initSelectionList( List );
      if (status = SelectSemantic( List, 8, x_coord ))
            return( status );
      if (status = SelectSemantic( List, 9, y_coord ))
            return( status );
      if (status = SelectSemantic( List, 10, xy_operation ))
            return( status );
      if (status = SelectSemantic( List, 11, sub_picture ))
            return( status );
      if (status = SelectSemantic( List, 12, Set_Color ))
            return( status );
      if (status = SelectSemantic( List, 13, G_applytrans ))
            return( status );

      SelectPreTuple( List, G_ready );

      /* do we continue with the current stack or clear it? */
      /* force initialization if stack is empty */
      if (MatrixStack == NULL)
```

```
            initialize = 'y';
      switch (initialize)
      {
      case 'y': case 'Y':
            /* initialize the matrix stack */
            MatrixStack = G_ident();
            break;
      default:
            /* otherwise continue with previous stack contents */
            break;
      }

      return( Gdraw1( relation_name ));
}

/*-----------------------------------------------------------------
-
 *
 *    once the List of select semantic codes has been prepared,
 *    Gdraw1 can be used to perform the graphical
 *    interpretation.
 *
 -----------------------------------------------------------------
*/

Gdraw1( relation_name )
char *relation_name;
{
      int status;

      MatrixStack = G_DuplicateTopOf( MatrixStack );
      status = interpret( database_name, relation_name, List );
      MatrixStack = G_DeleteTopOf( MatrixStack );
      return( status );
}

/*-----------------------------------------------------------------
 *
 *    graphics routines to be called by the interpreter
 *
 *-----------------------------------------------------------------
*/

int
x_coord( X )
char *X;
{
      extern float next_x, next_y;
      double atof();

      next_x = atof( X );
#ifdef GDEBUG
      if (Gdebug) printf("x_coord: next_x=%f", next_x );
#endif

      return( 0 );
}
```

```
int
y_coord( Y )
char *Y;
{
      extern float next_x, next_y;
      double atof();

      next_y = atof( Y );
#ifdef GDEBUG
      if (Gdebug) printf("y_coord: next_y=%f", next_y );
#endif

      return( 0 );
}

int
xy_operation( operation )
char *operation;
{
      extern float next_x, next_y;
      int atoi();

#ifdef GDEBUG
      if (Gdebug) printf("xy_operation: next_x=%f, next_y=%f, xyop=%s",
                  next_x, next_y, operation );
#else
      switch( operation[0] ) {
            case 'M':         /* move absolute */
            case 'm':
            case '1':
                  G_moveabs2( next_x, next_y );
                  break;

            case 'D':         /* draw absolute */
            case 'd':
            case '2':
                  G_lineabs2( next_x, next_y );
                  break;

            default:
                  return( 1 );
                  break;
      }
#endif

      return( 0 );
}

int
sub_picture( name )
char *name;
{
      int status;

      status = Gdraw1( name );

      return( status );
```

```
}

int
Set_Color( Color )
char *Color;
{
      switch (Color[0]) {
            case '0': /* dark */
            case '1': /* red */
            case '2': /* green */
            case '3': /* yellow */
            case '4': /* blue */
            case '5': /* magenta */
            case '6': /* cyan */
            case '7': /* white */

            case 'D': case 'd': /* dark */
            case 'R': case 'r': /* red */
            case 'G': case 'g': /* green */
            case 'Y': case 'y': /* yellow */
            case 'B': case 'b': /* blue */
            case 'M': case 'm': /* magenta */
            case 'C': case 'c': /* cyan */
            case 'W': case 'w': /* white */
                  setcolor( (int) Color[0] );
                  break;

            default:
#ifdef GDEBUG
      if (Gdebug) fprintf(stderr, "Set_Color: unrecognizable color '%s',
ignored.",Color);
#endif
                  break;
      }
}

int
G_ready()
{
      /*
       *
       * if the matrix on the top of the matrix stack has been
       * contaminated, then delete it and replace it with a fresh
       * copy of the matrix under it.
       *
       */

      if (MatrixStack->contaminated) {
            G_ReduplicateTopOf( MatrixStack );
      }
}

#include "Gapplytrans.c"
#include "Gmatutility.c"
```

File name: grip/Gutility.c

```
/* default settings to get CORE started */

#define TRUE 0

Gsetup()
{
      initializecore( "buffered", "no-input", "2d" );
      initializeviewsurface("TERM");
      selectviewsurface("TERM");
      setndcspace2(1.0,0.625);

      setwindow( 0.0, 1.0, 0.0, 0.625);
      setwindowclipping(TRUE);
      setviewport2(0.0,1.0,0.0,1.0);
}

Gclose()
{
      deselectviewsurface("TERM");
      terminateviewsurface("TERM");
      terminatecore();
}

Gpause()
{
      while ( getchar() == '');
}
```

File name: grip/Gmatutility.c

```
/*-----------------------------------------------------------------
 *
 * matrix utility routines for matrix manipulation
 *
 -----------------------------------------------------------------
*/
/* construct and return a pointer to a identity matrix */
MATRIX_PTR
G_ident ()
{
      MATRIX_PTR temp;

      temp = (MATRIX_PTR) malloc(sizeof(MATRIX_NODE));

      temp->a = 1;
      temp->b = 0;
      temp->c = 0;
      temp->d = 0;
      temp->e = 1;
      temp->f = 0;

      temp->contaminated = FALSE;
      temp->last = NULL;

      return(temp);
}

/* print the contents of a matrix */
void
G_matprint(mat)
MATRIX_PTR mat;
{
      char *format;

      if (mat != NULL) {
            format = "[%13g %13g %13g]";

            printf(format,mat->a,mat->d,0.0);
            printf(format,mat->b,mat->e,0.0);
            printf(format,mat->c,mat->f,1.0);
            printf("%s",mat->contaminated?"contaminated":"clean");
      }
}

MATRIX_PTR
G_matmult (X,Y)
MATRIX_PTR X,Y;
{
      MATRIX_PTR Z;

      /*
       *
       * multiply the two incoming matricies together: Z = X*Y
       * assumption: the last column in always [ 0 0 1]
       * get space for the resultant matrix
```

```
         *
         */
        Z = (MATRIX_PTR) malloc(sizeof(MATRIX_NODE));

        Z->a = X->a*Y->a + X->d*Y->b;
        Z->b = X->b*Y->a + X->e*Y->b;
        Z->c = X->c*Y->a + X->f*Y->b + Y->c;
        Z->d = X->a*Y->d + X->d*Y->e;
        Z->e = X->b*Y->d + X->e*Y->e;
        Z->f = X->c*Y->d + X->f*Y->e + Y->f;

        Z->last = Y->last;
        Z->contaminated = TRUE;

        /* and return the final instance matrix */
        return(Z);
}

/*-----------------------------------------------------------------
 *
 * G_DeleteTopOf
 *    Delete the matrix that is on the top of the stack.
 *
 -----------------------------------------------------------------
*/

MATRIX_PTR
G_DeleteTopOf( stack )
MATRIX_PTR stack;
{
        MATRIX_PTR temp;

        if (stack == NULL) {
                /* nothing to delete */
                return( NULL );
        }
        else {
                temp = stack->last;
                free( stack );
                return( temp );
        }
}

/*-----------------------------------------------------------------
 *
 * G_DuplicateTopOf
 *    duplicate the matrix that is on the top of the stack.
 *
 -----------------------------------------------------------------
*/

MATRIX_PTR
G_DuplicateTopOf( stack )
MATRIX_PTR stack;
{
        MATRIX_PTR temp;
```

```
        if (stack == NULL) {
                /* nothing to duplicate */
                return( NULL );
        }
        else {
                temp = (MATRIX_PTR) malloc( sizeof( MATRIX_NODE ) );
                temp->a = stack->a;
                temp->b = stack->b;
                temp->c = stack->c;
                temp->d = stack->d;
                temp->e = stack->e;
                temp->f = stack->f;
                temp->contaminated = FALSE;
                temp->last = stack;

                return( temp );
        }
}

/*-----------------------------------------------------------------------
 *
 * G_ReduplicateTopOf
 *    delete the top of the matrix stack and duplicate the
 *    the matrix that was immediately below it.
 *          Equivalent to but more efficient than:
 *                stack = G_DeleteTopOf( stack );
 *                stack = G_DuplicateTopOf( stack );
 *
 -----------------------------------------------------------------------
*/

void
G_ReduplicateTopOf( stack )
MATRIX_PTR stack;
{
        MATRIX_PTR temp;

        temp = stack->last;

        stack->a = temp->a;
        stack->b = temp->b;
        stack->c = temp->c;
        stack->d = temp->d;
        stack->e = temp->e;
        stack->f = temp->f;

        stack->contaminated = FALSE;
}

/*-----------------------------------------------------------------------
 *
 * the primative move and draw routine which may change
 * with different physical graphics devices or packages
 * - this code is written specifically for the CORE Graphics
 *    Package at RIT
 *
 -----------------------------------------------------------------------
```

```
*/

G_moveabs2( x, y )
float x, y;
{
      moveabs2( MatrixStack->a*x+MatrixStack->b*y+MatrixStack->c,
              MatrixStack->d*x+MatrixStack->e*y+MatrixStack->f );
}

G_lineabs2( x, y )
float x, y;
{
      lineabs2( MatrixStack->a*x+MatrixStack->b*y+MatrixStack->c,
              MatrixStack->d*x+MatrixStack->e*y+MatrixStack->f );
}
```

File name: grip/Gapplytrans.c

```
/*-----------------------------------------------------------------
 *
 * G_applytrans
 *    parse a character string defining a sequence of
 *    transformations to be accummulated and concatenate
 *    with the current matrix on the top of the stack.
 *    The string may contain any number of occurances of
 *    scale, rotation and translation in any order.
 *    Transformations are evaluated from left to right.
 *          The formats are:
 *               T(1.1,2.2) or t(1.1,2.2) - translate
 *               S(1.1,2.2) or s(1.1,2.2) - scale
 *               R(90)      or r(90)      - rotate
 *
 *    NOTES:
 *    The rotate angle is in degrees.
 *    The instance matrix is developed and finally multiplied
 *    by the current matrix on the top of the stack.
 *
 -----------------------------------------------------------------
*/

void
G_applytrans( transform )
char *transform;
{
      float x,y,angle;
      MATRIX_PTR Q;
      double tmp,sine,cosine,radians;
      MATRIX_PTR G_matmult(),G_ident();

      /* apply transformations from transform string */
      /* first create the new matrix to accumulate the */
      /* transformations into: Q = I */
      Q = G_ident();

      /* now decode and apply each transformation in order */
      while (*transform != '') {
      /* next character describes transformation to be performed */
            switch (*transform++) {
            case 't': /* Q = Q*T */
            case 'T': sscanf(transform,"(%f,%f)",&x,&y);
                      Q->c = Q->c+x;
                      Q->f = Q->f+y;
                      break;

            case 's': /* Q = Q*S */
            case 'S': sscanf(transform,"(%f,%f)",&x,&y);
                      Q->a = Q->a*x;
                      Q->b = Q->b*x;
                      Q->c = Q->c*x;
                      Q->d = Q->d*y;
                      Q->e = Q->e*y;
```

```
                Q->f = Q->f*y;
                break;

        case 'r': /* Q = Q*R */
        case 'R': sscanf(transform,"(%f)",&angle);
                radians = angle*3.14159/180.0;
                sine = sin(radians);
                cosine = cos(radians);

                tmp = Q->a*cosine + Q->d*sine;
                Q->d = -Q->a*sine + Q->d*cosine;
                Q->a = tmp;

                tmp = Q->b*cosine + Q->e*sine;
                Q->e = -Q->b*sine + Q->e*cosine;
                Q->b = tmp;

                tmp = Q->c*cosine + Q->f*sine;
                Q->f = -Q->c*sine + Q->f*cosine;
                Q->c = tmp;
                break;

        default : break;
        }
    }

    /* now put these transformations together with the
     * natrix on the top of the stack.
     */
    /* matrix: M = Q*N  */
    MatrixStack = G_matmult( Q, MatrixStack );

    /* get rid of the Qorary matrix */
    free(Q);
}
```

File name: grip/makefile

```
# makefile
#
#     RIT Graphics Interpreter Package
#

PRINTER = rpr
CFILES = Gdraw.c Gutility.c Gmatutility.c Gapplytrans.c
CHDRS = grip.h
OBJS = Gdraw.o Gutility.o
LIBACCT = /a1/f1/dgj/thesis/prog/interp/

#-----------------------------------------------------------------

griplib: rip.h $(CHDRS) $(OBJS)
      ar rucv griplib $(OBJS)
      chmod 755 griplib
      ranlib griplib

.c.o: ;
      mscc -g -c $*.c

Gdraw.o: Gdraw.c Gmatutility.c Gapplytrans.c grip.h

Gutility.o: Gutility.c grip.h

update:     rip.h $(CHDRS) $(OBJS)
      rm griplib
      ar rucv griplib $(OBJS)
      chmod 755 griplib
      cp griplib $(LIBACCT)griplib
      ranlib $(LIBACCT)griplib
      ranlib griplib
      cp grip.h $(LIBACCT)grip.h

rip.h: $(LIBACCT)rip.h
      cp $(LIBACCT)rip.h rip.h

#-----------------------------------------------------------------

print :
      cpr -h $(CHDRS) $(CFILES) | $(PRINTER)
      pr makefile | $(PRINTER)
```

Appendix D - Source Code for CRIP

File name: crip/Catalog.c

```
/*-----------------------------------------------------------------------
 *
 * Catalog.c
 *
 -----------------------------------------------------------------------
*/

#include "grip.h"

char *database_name;
PFI List[ MAXSCODE+1 ];
int part_num(), part_name(), part_description(), part_price(), part_picture();
int page_begin(), page_hold();

#ifdef CDEBUG
int Cdebug = 0;    { set to 1 for debug output }
#endif

int
Catalog( DB_name, relation_name )
char *DB_name;
char *relation_name;
{
      int status;

      /* set the global copy of the database name */
      database_name = DB_name;

      /* define semantic meanings to be used by the interpreter for
       * the Catalog application.
       */
      initSelectionList( List );
      if (status = SelectSemantic( List, 100, part_num ))
            return( status );
      if (status = SelectSemantic( List, 101, part_name ))
            return( status );
      if (status = SelectSemantic( List, 102, part_description ))
            return( status );
      if (status = SelectSemantic( List, 103, part_price ))
            return( status );
      if (status = SelectSemantic( List, 104, part_picture ))
            return( status );

      SelectPreTuple( List, page_begin );
      SelectPostTuple( List, page_hold );

      page_layout();

      return( interpret( DB_name, relation_name, List ));
}
```

```
/*-----------------------------------------------------------------
 *
 *    routines to be called by the interpreter
 *
 -----------------------------------------------------------------
*/

int
part_num( pn )
char *pn;
{
      moveabs2( 0.5, 6.0 );
      text( "Part #: " );
      moveabs2( 1.5, 6.0 );
      text( pn );

      return( 0 );
}

int
part_name( pn )
char *pn;
{
      moveabs2( 0.5, 5.75 );
      text( "Name: " );
      moveabs2( 1.2, 5.75 );
      text( pn );

      return( 0 );
}

int
part_description( pd )
char *pd;
{
      moveabs2( 0.5, 3.25 );
      text( "Description:" );
      moveabs2( 0.5, 3.00 );
      text( pd );

      return( 0 );
}

int
part_price( price )
char *price;
{
      moveabs2( 0.5, 5.50 );
      text( "Single unit price: " );
      moveabs2( 2.75, 5.50 );
      text( price );

      return( 0 );
}

int
part_picture( picture )
char *picture;
```

```
{
      int status;

      /* place the picture in the upper right hand corner */
      closeretainedsegment();
      setviewport2( 0.50, 1.00, 0.50, 1.0 );
      setwindow( 0.0, 10.0, 0.0, 6.25 );
      createretainedsegment( 3 );

      status = Gdraw( database_name, picture, 'y' );

      /* change things back to the way they were */
      closeretainedsegment();
      setviewport2( 0.0, 1.0, 0.0, 1.0 );
      setwindow( 0.0, 10.0, 0.0, 6.25 );
      createretainedsegment( 4 );

      return( status );
}

void
page_layout()
{
      setwindow( 0.0, 10.0, 0.0, 6.25 );
      createretainedsegment( 1 );

      /* draw the page form */
      setcolor( "green" );
      moveabs2( 0.0, 0.0 );
      lineabs2( 10.0, 0.0 );
      lineabs2( 10.0, 6.25 );
      lineabs2( 0.0, 6.25 );
      lineabs2( 0.0, 0.0 );

      moveabs2( 10.0, 3.125 );
      lineabs2( 5.0, 3.125 );
      lineabs2( 5.0, 6.25 );
      setcolor( "white" );

      closeretainedsegment();
}

void
page_begin()
{
      setwindow( 0.0, 10.0, 0.0, 6.25 );
      createretainedsegment( 2 );
}

void
page_hold()
{
      Gpause();

      closeretainedsegment();
      deleteretainedsegment( 2 );
      deleteretainedsegment( 3 );
```

```
        deleteretainedsegment( 4 );
}
```

File name: crip/makefile

```
# makefile
#
#      RIT Catalog Interpreter Package
#

PRINTER = rpr
CFILES = Catalog.c
CHDRS =
OBJS = Catalog.o
LIBACCT = /a1/f1/dgj/thesis/prog/interp/

#-------------------------------------------------------------------------------

criplib: grip.h $(OBJS)
      ar rucv criplib $(OBJS)
      chmod 755 criplib
      ranlib criplib

.c.o: ;
      mscc -g -c $*.c

update:     grip.h $(OBJS)
      rm criplib
      ar rucv criplib $(OBJS)
      chmod 755 criplib
      cp criplib $(LIBACCT)criplib
      ranlib $(LIBACCT)criplib
      ranlib criplib

grip.h: $(LIBACCT)grip.h rip.h
      cp $(LIBACCT)grip.h grip.h

rip.h: $(LIBACCT)rip.h
      cp $(LIBACCT)rip.h rip.h

#-------------------------------------------------------------------------------

print :
      cpr -h $(CHDRS) $(CFILES) | $(PRINTER)
      pr makefile | $(PRINTER)
```

Appendix E - Examples of main programs for GRIP and CRIP

File name: catalog.c

```
/*-----------------------------------------------------------------
 *
 * catalog
 *    catalog is a sample main program invoking the catalog
 *    interpreter package and illustrating how one application
 *    interpreter can use another.
 *
 -----------------------------------------------------------------
*/

#include <math.h>
#include "rip.h"

main(argc,argv)
int argc;
char *argv[];
{
      int error;

      Gsetup();
      setwindow( 0.0, 10.0, 0.0, 6.25 );
      error = Catalog( argv[1], argv[2] );
      Gpause();
      deleteallretainedsegments();
      Gclose();

      printf(" Catalog returned %d", error );
}
```

File name: draw.c

```
/*-----------------------------------------------------------------
 *
.*draw
 *    draw is a sample main program to display graphics images
 *    stored in a Mistress database in GRIP form.  draw illustrates
 *    the few lines of code necessary to include a GRIP image in
 *    an application.
.*
 -----------------------------------------------------------------
*/
#include <math.h>
#include "grip.h"

main(argc,argv)
int argc;
char *argv[];
{
      int error;

      Gsetup();
      setwindow( 0.0, 10.0, 0.0, 6.25 );
      setwindowclipping(1);
      createretainedsegment( 1 );

      error = Gdraw( argv[1], argv[2], 'y' );

      closeretainedsegment();
      Gpause();
      deleteretainedsegment( 1 );
      Gclose();

      printf(" Gdraw returned %d", error );
}
```

File name: makefile

```
#
#      RIT Interpreter Package for Self-Describing Mistress Databases
#

TESTING = -g
PRINTER = rpr
CORE = /a2/misc/graphics/core/corelib

#-------------------------------------------------------------------------------

all : draw catalog

draw : draw.c rip.h griplib riplib
      mscc $(TESTING) -o draw draw.c \
            griplib riplib $(CORE) -lm
#     strip draw

catalog : catalog.c rip.h criplib griplib riplib
      mscc $(TESTING) -o catalog catalog.c \
            criplib griplib riplib $(CORE) -lm
#     strip catalog

test : test.c rip.h riplib
      mscc $(TESTING) -o test test.c riplib

#-------------------------------------------------------------------------------

print : PRT_draw PRT_catalog PRT_MAKE

PRT_draw : draw.c
      cpr -h draw.c | $(PRINTER)
PRT_catalog : catalog.c
      cpr -h catalog.c | $(PRINTER)
PRT_MAKE : makefile
      pr makefile | $(PRINTER)
```

Appendix F - Trademarks

Trademarks

The following names are registered trademarks of the indicated companies.

Unix™	AT&T Information Systems
Mistress™	Rhodnius