EXPERIENCE WITH A GRAPHICAL DATA BASE SYSTEM

Mark Green, M. Burnell, H. Vrenjak, M Vrenjak
Unit for Computer Science
McMaster University
Hamilton, Ontario, Canada
L8S 4K1

ABSTRACT

Over the past few years there has been an increase in the use of data base systems and computer graphics in scientific and engineering applications. While considerable work has been done on both graphics and data base support very little has been done on the interaction between the two. In this paper we discuss a graphical data base system that we have used in a number of applications. We describe the structure of the data bases used in this system, the operations that can be performed on these data bases and three typical applications. Emphasis is placed on the interaction between the data base system and graphical input and output.

KEYWORDS: Graphical data bases, Interactive graphics, Computer aided design

1. Introduction

In implementing several graphical applications we encountered the following rather common problem. The data structures used by the programs were larger than the main memory of the computer being used (PDP 11/23). These data structures also had to be stored on disk between runs of the program. Typically when these problems arise a file structure is designed for the data structures and some scheme is developed for paging the file between disk and main memory. Since several applications would be using this strategy it was decided to produce one common file structure and a library of routines to manipulate it. The main motivation behind this decision was to save implementation time. When the design was finished we discovered that we had a small data base system instead of a file structure.

Originally, we had discarded the idea of using a data base system for the following reasons. First, the data base systems available for the PDP11 (under UNIX) are based on the relational model [Date 1981]. While we could encode our data in terms of the relational model this did not seem natural to us. In the relational model data is viewed as a collection or tables called relations. Each table has a number of rows, called tuples, and a number of columns, called attributes. Each entity or object represented in the data base is stored as one row in a table. The attributes in the table are the properties of the object. For example, if we were designing an electronic circuit we might have a relation for the parts in the circuit. There would be one row in the table for each part. The columns of the table would represent the properties of the parts, such as part number, value, manufacturer and rating. To represent the topology of the circuit another relation would be used to store the connections between the parts. This relation would have one row for each connection and the attributes would be the parts being connected. In order to cross reference the connection relation and the parts relation we would need a unique identifier for each part.

While the relational model is well suited to business and a number of scientific problems, it does not do so well when it comes to representing designs and the design process. The application areas we are interested in are software, system and VLSI design. In these application areas a design is often developed incrementally over a period of time. When a new component is introduced to the design quite often all its properties are not known. This causes problems in the relational model since all the attributes of a tuple must be known when it is added to a relation. So in the previous example each time we add a part to the parts relation we need to know its part number and value, which is usually the case, but also its manufacturer and rating. The values of the last two attributes may not be known until the design is finished.

Another problem with the relational model is the way objects are represented. Each object in the data base corresponds to one row in a table. This has two significant implications in a design environment. First, in order to store all the properties of complex objects we must either use very wide relations (a large number of attributes) or several relations. Wide relations tend to cause storage problems in most relational systems so they are not a viable solution. On the other hand if we use several relations the data base system will not support operations on the whole object. The second implications is, if we want to store all the parts in one relation they must have a homogeneous structure. This means that all the parts must have the same attributes. In most design applications this is not the case so multiple relations are required to store all the parts. By using more than one relation to store conceptually similar objects we loose most of the simplicity and processing power of the relational model.

These disadvantages don´t mean that relational data bases cannot be used in graphics. Several successful systems based on the relational model have been produced, for example the work of Weller and Williams [Weller and Williams 1976], and Garret and Foley [Garret and Foley 1982]. There has also been some work on removing the above disadvantages from relational data base systems [Lorie and Plouffe 1982].

The second problem with most data base systems is the time required to access information. The information in the data base will be used for generating displays so the time required to retrieve stored information is critical to the success of the program. In most data base systems information retrieval is based on searching all or part of the data base for a record that satisfies a particular condition. In our case the records we are looking for are the ones that are to be displayed, and typically these records can be retrieved without searching. Therefore, search based retrieval is not neccessary and needlessly slows down the program.

In designing our data base system a number of design goals were kept in mind. These design goals are:

1) Retrieval Time - The success of most of our applications depended upon how fast displays could be generated. In order to be useful the data base system must be capable of retrieving all the information required to generate a display in less than two seconds.
2) Flexibility - The underlying data model must be flexible enough to accommodate the data bases produced in most design applications. This includes the ability to add arbitrary connections between objects while the application is running.
3) Implementable - We are not in the business of producing data base systems so we wanted to spend as little time as possible implementing the data base system. We also wanted the implementation to be well structured so it could be modified to handle different types of applications at a later time.
4) Usable - Since we are the major users of the data base system we wanted it to be relatively easy to use. Not only must the system be relatively easy to incorporate into programs, but it must also be easy to produce tools based on the data base system.

In view of the above design goals we decided to base our data base system on a combination of the network model and the frame systems that have been used in artificial intelligence [Barr and Feigenbaum 1981]. Since frames are the basic unit in our data bases we call our system FDB, short for Frame based Data Base system. This approach gets around the problems encountered with relational systems by assigning a unique frame identifier to each data object added to the data base. When an object is added to the data base none of its properties need be known. Its identifier can be used to reference it in other places in the data base. This identifier also makes it possible to retrieve information quickly since we no longer need to search for particular pieces of data. In order to decrease access time FDB also stores a portion of the data base in main memory.

In this paper we first present a description of the structure of FDB data bases. Then we describe how FDB is used in applications programs and the tools and techniques that have been developed to make it easier to use. In the fourth section of this paper we briefly look at some of the applications that use FDB.

2. The Structure of FDB Data Bases

The frame is the basic organizational unit in FDB data bases, each object is stored in a frame. A frame is made up of a collection of slots. Each slot is capable of storing one piece of data. A slot has four fields: name, visibility, type and value. The name field of a slot contains the name used to reference that slot. The same slot name can be used in a number of frames. The visibility field of a slot can be used either for security purposes or grouping together similar slots. For example, all slots dealing with graphical properties of objects could be given the same visibility level. Then any program traversing the data base will be able to easily separate the graphical from the nongraphical data. The type field of a slot specifies the type of data stored in the slot. The type of a slot can change over time. The value field contains the data value stored in the slot.

There are two types of frames in FDB data bases. Meta frames are used to represent the structure of the data base. A meta frame contains no application data, it describes the format of other frames in the data base. Ordinary frames are used for storing the applications data. If an ordinary frame has a meta frame then all the slots in the meta frame will also appear in the ordinary frame. Any slot values stored in the meta frame will be the default values for the corresponding slots in the ordinary frame.

There are three standard slots in all frames. The ´type´ slot indicates whether the frame is an ordinary frame of a meta frame. The ´isa´ slot is used to link ordinary frames to their meta frames. The value of the ´isa´ slot is the frame identifier of the corresponding meta frame. The ´owner´ slot stores the owner of the frame. The owner of a frame is the user who created the frame. This makes it possible to share a data base amongst several users and still keep track of who is responsible for what parts of it.

There are four types of data that can be stored in slots. Two of these types are for numeric data, integer and real. The third type is for character strings. The strings can be of arbitrary length, the length of a string is stored with it. The fourth data type is a pointer to a frame. This data types allows us to build up relationships between different objects.

There are two ways in which frames can be linked up to form networks. The first way is through the ´isa´ slot to the corresponding meta frame. This is referred to as inheritance since following this path gives us the default slots for the frame (ie. the ones it inherits from its parents). The isa network is automatically provided by FDB as long as the program fills in the ´isa´ slot.

The other way in which a network can be formed is through frame valued slots. There are two common linking structures that occur in FDB data bases. The first is a ring or list structure which is used to group together related frames of equal importance. To implement a structure of this form one slot in each frame is set aside as a link slot. The link slot of the first frame in the set points to the second frame in the set. Similarly the link slot of the second frame points to the third frame. If a ring structure is being constructed the link slot of the last frame in the set points to the first frame, otherwise it contains a null value. Sometimes a header frame is used to keep track of the first and last frames in a set. This structure is similar to the sets found in CODASYL DBTG data bases [Date 1981].

The other common linking structure is based on trees. In this case a frame has one or more frame valued slots. These slots point to the children of the frame. This makes it easy to model the hierarchies that occur in a number of design problems.

To illustrate how these techniques work we will show how a directed graph can be represented in an FDB data base. Directed graphs form the basis for a large number of notations for describing processes, such as Petri nets, PERT diagrams, transition diagrams and the data flow diagrams used in structured analysis. A directed graph consists of a set of nodes and a set of arcs which are used to connect the nodes. Each arc has a start node and an end node. A typical directed graph is shown in fig. 1. In this figure the nodes are represented by circles and the arcs by arrows.

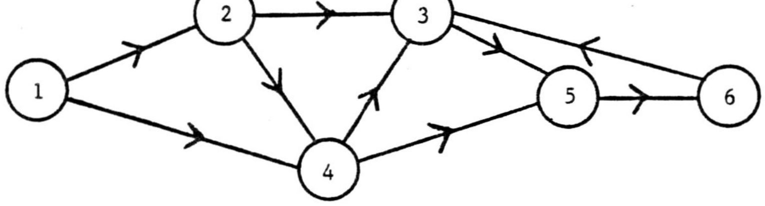


Fig. 1 A typical directed graph

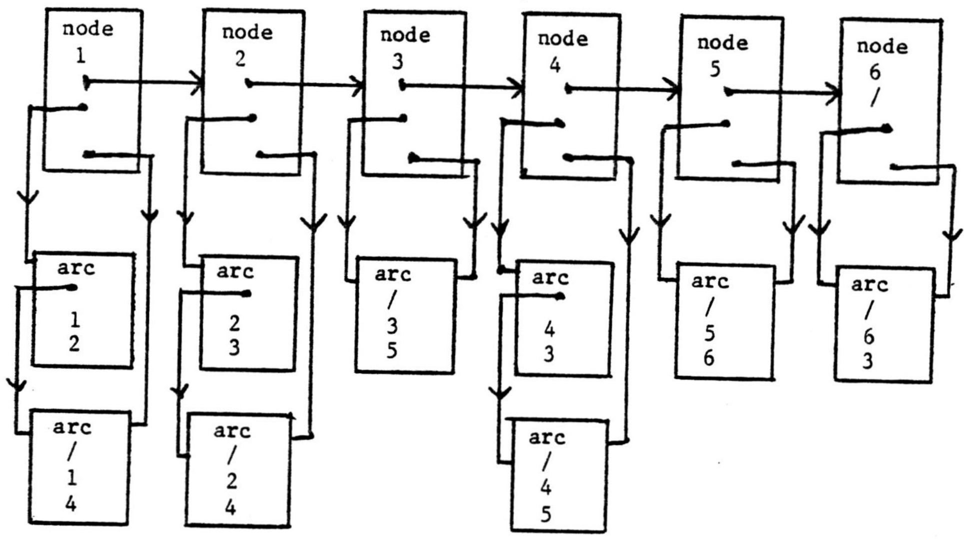


Fig. 4 FDB data base for directed graph

Since there are two basic types of objects in directed graphs we will need two meta frames to describe the structure of the data base. The node meta frame is shown in fig. 2. The ´type´, ´owner´ and ´isa´ slots will not be shown in this example since they are not part of the application. The ´fheader´ slot indicates that this frame is used to represent nodes. A header slot is included in a frame so we can tell in a traversal of a data base the purpose of the frames we encounter. The ´name´ slot is used to record the name of the node. The special value EMPTY is used to indicate that this slot currently has no value. The ´next_node´ slot is used as a link slot so all the node frames in the graph can be linked into a list. The arcs in the graph will also be linked into lists. There will be one arc list for each node in the graph. The arcs in the list are the ones leaving the node the list is attached to. The slots ´first_arc´ and ´last_arc´ point to the first and last arcs on the arc list for this node. The special frame pointer F0 is used as a null pointer, that is it does not point to a legal frame.

```
fheader   : node
name      : EMPTY
next_node : F0
first_arc : F0
last_arc  : F0
```

Fig. 2 Meta frame for nodes

The meta frame for arcs is shown in fig. 3. The ´fheader´ slot is again used to record the type of object represented by the frame. The ´next_arc´ slot is used to link together all the arcs that leave the same start node. The slots ´start_node´ and ´end_node´ point to the nodes at the start and end of the arc.

```
fheader    : arc
next_arc   : F0
start_node : F0
end_node   : F0
```

Fig. 3 Meta frame for arcs

Using these two meta frames as a basis we can construct the data base shown in fig. 4. for the directed graph in fig. 1. In this figure frames are represented by rectangles with their slot values written inside of them. Arrows are used to represent pointers between frames, except for the ´start_node´ and ´end_node´ slots of the arc frames where the node name are used to reduce clutter.

Previously we mentioned that the time required to access data was important to us. FDB solves this problem in two ways. First, it always keeps a portion of the data base in main memory. When the program opens the data base it can specify how much of the data base is to be kept in main memory. This portion of the data base is selected by a least recently used strategy. When a request is made for information not in main memory, the least recently used data is placed back on disk in order to make room for the requested data. The reasoning behind this strategy is, the user is more likely to request displayed information than information that is not displayed. In producing a display the required data must be transferred to main memory so it will form the main memory portion of the data base. In this way response to changes in the displayed information will usually be instantaneous.

The use of frame pointers also helps to reduce access time. FDB maintains an index on frame identifiers so when a request is received for a particular frame only an index search and a read are required to satisfy the request. As we will see in the next section most of the operations performed on FDB data bases involve frame pointer traversals, so efficient retrieval of particular frames is very important.

3. Using FDB Data Bases

Now that we have presented the structure of FDB data bases we will turn our attention to the operations that can be performed on them. From the graphics and interaction points of view there are several key operations that must be supported. These operations are: displaying all or part of the data base, detecting when a displayed object has been hit and retrieving information associated with displayed objects. We will now see how these operations can be performed in FDB. Traditional data base operations, such as the storage and retrieval of information, are also supported by FDB.

To illustate how the display operations work we will return to the directed graph example of section 2. If we want to display a directed graph the first thing we need to know is the positions of the nodes. There are several ways in which these positions could be determined. They could be calculated by a program that traverses the data base or they could be entered interactively by a graph editor. We will assume that the positions already exist and two slots, ´x´ and ´y´, are used to store them. The new meta frame for nodes is shown in fig. 5.

fheader	: node
name	: EMPTY
next_node	: F0
first_arc	: F0
last_arc	: F0
x	: 0
y	: 0

Fig. 5 Extended meta frame for nodes

With this information displaying the graph reduces to performing a traversal of the data base. This traversal consists of visiting each node frame in the data base and drawing a circle to represent the node. Then the arc list for the node is followed and an arrow is produced for each arc. The end points of the arrow can be found by accessing the node frames at each end of the arc. The outline of a procedure to perform this traversal is shown in fig. 6.

```
procedure draw_graph(start : frame);
  var
    current : frame;
    start_node, end_node : frame;
    x, y : integer;
    x1, y1, x2, y2 : integer;
  begin
    current := start;
    while current ≠ F0 do
      begin
        x = getvalue(current,x);
        y = getvalue(current,y);
        draw_circle(x,y,radius);
        arcs := getvalue(current,first_arc);
        while arcs ≠ F0 do
          begin
            start_node := getvalue(arcs,
                              start_node);
            end_node := getvalue(arcs,end_node);
            x1 := getvalue(start_node,x);
            y1 := getvalue(start_node,y);
            x2 := getvalue(end_node,x);
            y2 := getvalue(end_node,y);
            intersect(x1,y2,x2,y2,radius);
            intersect(x2,y2,x1,y1,radius);
            draw_arrow(x1,y1,x2,y2);
            arcs := getvalue(arcs,next_arc);
          end;
        current := getvalue(current,next_node);
      end;
  end;
```

Fig. 6 Procedure to draw directed graphs

In this procedure the function ´getvalue´ is used to retrieve the value of a slot within a frame. The first parameter to getvalue is the identifier of the frame and the second parameter is the name of the slot. The procedure ´intersect´ intersects a line with a circle and returns the point of intersection in the first two parameters. It is assumed that the circle is centered on the first end point of the line. This procedure is used to find where the arrow for the arc will intersect the circles representing the nodes so the resulting image will look better.

The draw_graph procedure illustrates the basic technique used to convert FDB data bases to images. In section 2 we showed that there are two basic linking schemes, sets and hierarchies. In displaying a data base we enumerate the elements of a set by following the next slot for the set. At each frame in the set we follow the hierarchy pointers to traverse the parts of the data base below it. In this way the parent node can set the display parameters for its children. This basic scheme is incorporated into the procedure ´traverse´ shown in fig. 7.

```
procedure traverse(start : frame; next : slot;
                   down : slot);
  var
    current : frame;
    below : frame;
  begin
    current := start;
    while current ≠ FO do
      begin
        draw(current);
        if gettype(current,down) = FRAME then
          begin
            set_parms(current);
            below := getvalue(current,down);
            traverse(below,next_slot(below),
                     down_slot(below));
            reset_params();
          end;
        current := getvalue(current,next);
      end;
  end;
```

Fig. 7 General display procedure

The traverse procedure has three parameters, the identifier of the frame where the traversal starts, and the names of the slots used for set and hierarchy pointers. In order to use this procedure the programmer must supply three procedures; draw, next_slot and down_slot. Given the identifier of a frame the draw procedure produces the image corresponding to that frame. The draw procedure does not produce the images of the frames pointed at by this frame. The next_slot and down_slot procedures return the names of the slots used for set and hierarchy pointers in the type of frame given by their parameter. All three of these procedures essentially consist of a case statement on the ´fheader´ slot and the code required to perform the operation for each type of frame in the data base. For our directed graph example the draw, next_slot and down_slot procedures are shown in fig. 8.

Each time a hierarchy pointer is followed the display parameters can be changed. This is the purpose of the calls to ´set_parms´ and ´reset_parms´ in the traverse procedure. The user can either supply his own version of these procedures or use the standard ones. An example of how this works is included in section 4.

```
function next_slot(f : frame) : slot;
  var
    ftype : integer;
  begin
    ftype := getvalue(f,fheader);
    case ftype of
      node : next_slot := next_node;
      arc :  next_slot := next_arc;
    end;
  end;

function down_slot(f : frame) : slot;
  var
    ftype : integer;
  begin
    ftype := getvalue(f,fheader);
    case ftype of
      node : down_slot := first_arc;
      arc  : down_slot := NULL;
    end;
  end;

procedure draw(f : frame);
  var
    ftype : integer;
    x, y : integer;
    x1, y1, x2, y2 : integer;
    start_node, end_node : frame;
  begin
    ftype := getvalue(f,fheader);
    case ftype of
      node : begin
               x := getvalue(f,x);
               y := getvalue(f,y);
               draw_circle(x,y,radius);
             end;
      arc :  begin
               start_node := getvalue(f,
                                 start_node);
               end_node := getvalue(f,end_node);
               x1 := getvalue(start_node,x);
               y1 := getvalue(start_node,y);
               x2 := getvalue(end_node,x);
               y2 := getvalue(end_node,y);
               intersect(x1,y1,x2,y2,radius);
               intersect(x2,y2,x1,y1,radius);
               draw_arrow(x1,y1,x2,y2);
             end;
    end;
  end;
```

Fig. 8 Next_slot, down_slot and draw procedures

In most interactive applications the user interacts with displayed data. The most common form of this interaction is called a hit, which occurs when the user points at an object on the screen and signals his selection, usually through the use of a button. In order to handle this type of interaction we must keep track of the position of each object on the display. The standard way of doing this is a correlation table or map [Foley and Van Dam 1982]. A correlation table has one entry for each user selectable object on the screen. This entry contains the screen area occupied by the object and the identity of the object. In our case the screen area is represented by the bounding rectangle for the object (the smallest rectangle that completely contains the object) and the object is identified by its frame identifier. The frame type is also stored in the correlation table.

When the user indicates he has selected something on the screen the correlation table is searched for an entry that contains the specified point. The result of this search is the frame corresponding to the selected object.

There are two problems that must be addressed when using this technique. The first problem is that bounding rectangles can overlap so the point specified by the user may correspond to several frames. There are several ways in which this problem can be solved. The first is to use the context of the action to determine the type of frame the user should be selecting. If only one frame of this type corresponds to the position pointed at, then it is the most likely choice. For example, if the operand to the command the user entered must be a node, then only frames representing nodes would be considered in a search of the correlation table. Another solution is to construct a precedence ordering on the frame types. A frame with higher precedence will be selected over one with lower precedence. A third solution is to report to the program that more than one frame has been selected and let the applications program decide which of the selected frames it will use. We have incorporated the first and third solutions into our correlation table routines. For each user hit the applications program can request the number of frames that satisfy the hit. If only one frame satisfies the hit there are no conflicts and the identifier and type of the frame are returned. If more than one frame satisfies the hit the number of frames satisfying the hit and the identifier and type of the first frame are returned. The applications program can then ask for the rest of the frames or make the request more specific by including a frame type.

The other problem with correlation tables is maintaining the table while the displayed information changes. Each time an object is removed from the screen it must also be removed from the correlation table. Similarly when a new object is displayed it must be added to the table. The obvious place to add entries to the table is the draw procedure that is called from traverse. At the end of each case in the case statement the procedure 'add_entry' is called to create a correlation table entry for the frame that was just drawn. The parameters to add_entry are two opposing corners of the bounding rectangle and the identifier for the frame. To remove an entry from the table the procedure 'remove_entry' is used. This procedure has one parameter, the identifier of the frame to be removed. Remove_entry must be called each time a frame is removed from the screen. In order to handle batch updates to the screen the procedure 'clear_table' can be used to clear the entire correlation table and 'remove_range' to remove a range of entries. The procedures 'begin_range' and 'end_range' are used to delimit the range of table entries defined by the intervening calls to add_entry.

Over the past few years a number of tools have been produced to aid in the construction and maintenance of FDB data bases. One useful tool is a program, called listdb, that produces a simple formatted listing of the contents of a data base. This listing is divided into two sections. The first section contains the meta frames and the second section the ordinary frames. In each section the frames are listed in frame identifier order. For each frame all the slots are listed along with their values. This program is used to check data bases that are produced or modified by applications programs.

Another useful tool is an interactive data base editor called fedit. Fedit allows the user to create a new data base or modify an existing one. This editor uses a standard ASCII terminal and a bit pad for selecting commands and arguments. A command menu is displayed at the bottom of the terminal screen. Above this menu is a work area containing either the contents of one frame or a list of the available slot names in the data base. To the right of the work area is a status area showing the size of the data base and the values of default arguments. All operations are performed by either selecting a command from the menu or pointing at parts of the frame shown in the work area. The menu includes commands for creating frames, adding slots, listing available slot names, removing frames, removing a slot from a frame and finding a particular frame. The slot values are changed by pointing (with the bit pad) at the slot to be changed and entering the new value on the keyboard. The syntax of the value determines its type. This simple command structure facilitates learning to use the editor. Fedit is mainly used to create the meta frames for a data base and enter the first few ordinary frames. It is rarely used to enter the entire data base.

Another useful tool is a set of programs that have been produced over the past few years to recover trashed data bases. Unlike commercial data base systems, FDB has no recovery mechanisms. When an application program crashes during testing it often leaves the data base in a bad state. Since most programmers assume their programs will work correctly the first time they rarely make a backup copy of their data base so recovery programs are a natural aid to program development. The current implementation of FDB does not overwrite deleted information on the disk. This means that old information can be recovered by adjusting some pointers and disk addresses. The recovery programs return the data base to a consistent state and recover as much data as possible. The programmer can then use fedit to restore the rest.

Most of our tools have been motivated by the applications we have tackled with FDB. As we look at new applications new tools and approaches suggest themselves.

4. Some Typical Applications

4.1. A Retrieval and Display System for well-defined Graphical Data Bases

We have noted that the FDB System is ideally suited for the representation of graphical and non-graphical information. We use this knowledge as a basis for the development of a display system for graphical / non-graphical data bases.

As this is one of the first attempts at applying the FDB System to the graphical medium some modification of the FDB System itself is a requirement for the success of the project. The modification deals only with extending the types of data that can be stored in frame slots. The complete set of types of data after this extension now include: the original types for integers, reals, character strings, and frame pointers, the types explicitly for graphical interpretation (point, line, and circle types), and the array types (for integer, real, and character string arrays). The value of having an explicit type to define, for example, an integer point or a line with real end-points or an array of all possible factory part names is obvious. The alternative is to painstakingly define and use many frame slots to fake what should otherwise be one logical data item.

Now that we have implemented these new data types in the FDB System we must consider how the user will enter the values corresponding to these types into a frame data base. This depends only on the agreed upon method of interaction between the user and the FDB System. If we are using FDB system calls either the value or a pointer to the value or a pointer to an array of pointers to values is passed as a parameter in the call. If an interaction tool such as "fedit" is used the value is entered in the same manner as it will appear in the screen representation of the slot and its value. We mention this simply as an assurance that the methods of interaction have been updated to be consistent with the extensions made to the types of data.

The input of large amounts of graphical and non-graphical data into the frame data base will have little meaning unless it is entered in some well-defined manner. The scheme used to enter, and then retrieve and display the acquired information forms the main scope of this chapter. However, first we should explain what we mean by "well-defined". A data base of one or more display items is well-defined if it has the following properties:

- A unique starting frame exists for every distinct display item in the data base.
- Only one starting frame is active at any one time (more on this later).
- Complicated display items are modelled using a tree based linking structure.

Thus the root of the tree corresponds to the start frame, the leaves hold the actual values, and each internal node has some organizational effect on its children. If we have such a data base then an interpreter can be implemented to scan the data base, retrieve a particular item, and display a visual representation of that item on some output device.

We have designed a scheme that is pertinent both to the user who is entering data into the frame data base and to the interpreter which retrieves and displays the data. Our scheme has two levels: an organizational level and an informational level. The organizational level concerns itself with the environment in which the constituent parts (figures) of the display item are found. Such factors as coordinate systems, colouring schemes, and orientation of figures are considered here. The informational level can be compared to a sub-tree of our tree, where the sub-tree describes some figure which cannot be broken down further without losing its meaning. The switch between the two levels is somewhat blurred in that each logically distinct figure of the display item is usually composed of more primitive building block figures and thus subject to some method of organization. For example, a 2 - dimensional box is a distinct informational figure which is composed of 4 line segments arranged in a certain fashion.

A core data base exists which is intended as an aid to the user in creating his frame data base. The core data base is a collection of meta frames which is used to format a well-defined graphical data base. The key meta frame is the organizational "worldframe". Its main function is to define a coordinate system for the figures that it identifies. The coordinate system specifies the area of the display in which the figures are located. The figures themselves are either primitive figures (the frame slot for the figure holds the actual value) or complex figures (the frame slot for the figure holds a frame pointer).

If a figure is complex the frame pointed to by the slot is modelled on one of two meta frame types. The new frame may be another "worldframe" or it may be a "pseudoprimitive" frame. If it is a "worldframe" then the coordinate system of that frame can also be specified. In this case the new coordinate system will be positioned relative to both the new system and the parent system. The figures in the new frame are also positioned accordingly. With this format we can define very complex figures quite quickly. We can also create "ord models" for complex figures. By this we refer to complex figures which will be used as logically discrete display values. These values may be used many times but with different orientation parameters and colouring schemes. This allows us to produce almost identical copies of a complex figure (which leads us to an interesting side-track: the application of moving pictures).

If the new frame is based on the "pseudoprimitive" frame type a very specific interaction between user and data base is required. The "pseudoprimitive" frame models a special purpose figure. Most of the frame slots are already designated with default values. The user only has to supply a minimal amount of information to complete the figure. Examples of such figures are the pie chart figure, the bar graph figure, and the business form figure (each requires a different "pseudoprimitive" meta model). It should be noted that the "pseudoprimitive" frame is both an informational and organizational frame. It is informational in that it represents one logically complete figure but it is also organizational in that complicated "pseudoprimitive" formats may include the use of "worldframe" types.

We now have a data base composed of organizational and informational frames which we want to display. The display interpreter is designed to have a specific action for each meta frame defined in the original core data base. If the user models his data base on the provided meta frames or if a library of "ord models" exist and are accessed by frame pointers the interpreter will display the data base item. If the user wants to add a new "pseudoprimitive" frame then he will also have to add a new action to the interpreter before the interpreter can display the data base item.

The interpreter begins by searching the data base for the active starting frame. The active starting frame is just a "worldframe" that has been labelled to identify it as the root starting frame for that display item. Having found the starting frame the interpreter initiates the "worldframe" action for that frame's slot values. The action is:

1) Define the coordinate system.
2) Set different colouring schemes.
3) Transform (scale, rotate, and translate) figures according to the coordinate system specifications (non-commutative transformations are required if more than one coordinate system level is specified).
4) Display the figures.

This is a recursive procedure where step 4) leads to further "worldframe" actions or "pseudoprimitive" frame actions. When the figures themselves are actual slot values (point, line, circle, character string, etc.) then the action is complete. The only terminal dependent operation made by the interpreter is in providing routines to display the final slot values.

Our system is still under development at this stage. But its design allows extensions to be made quite quickly. New "pseudoprimitive" frames require matching interpreter actions. New slot value types can be faked by creating "ord models" or by directly modifying the FDB System.

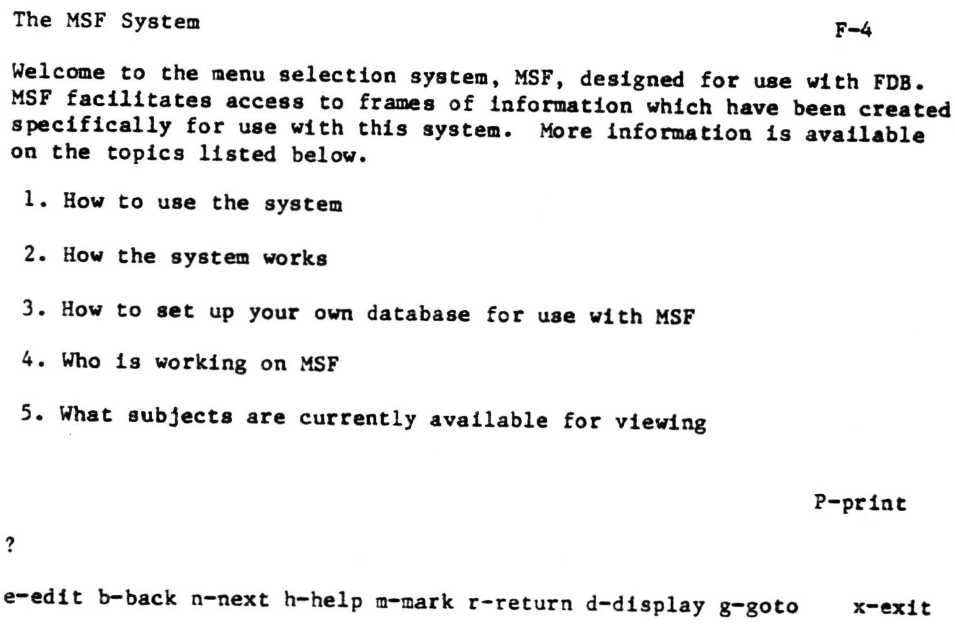
4.2. MSF : A Menu Selection System for use with FDB

MSF is a menu selection system, intended for use as a man-machine interface, which has been partially implemented on the PDP 11/23. The design of this system is based heavily on that of ZOG, a rapid response, large network, menu selection system that has been under study at Carnegie Mellon University for several years [Newell,1977] [Robertson,McCracken,and Newell,1980]. MSF facilitates access to and manipulation of frames stored in an FDB database. Although future plans for the system include its use to display graphical information and perform actions specified by the user and stored in frames, at the present time MSF is only capable of handling frames containing textual information.

While using the MSF system the user faces a terminal upon which is displayed a frame of information. Before leaving the terminal the user will probably view several such frames, called DISPLAY frames, all of the same general format. A typical MSF DISPLAY frame is shown in figure 9. Every frame contains a title at the top, possibly followed by a number of lines of text and/or options, and a line of global pads at the bottom. The upper right hand corner of the screen holds the frame number of the frame currently on display. There may also be a column of local pads along the right hand side of the screen, beside the options. As the user switches from one frame to another the format will remain the same but, with the exception of the global pads at the bottom, the content of the frame will change. Due to space constraints (a 24 line by 80 column screen is being used) the number of text and /or option lines is restricted to 17. The text is always displayed first followed by the options which are double spaced for easy assimilation and selection. Therefore, if there are no options, up to 17 lines of text can be displayed and, if there is no text, up to 9 options can be offered.

the keyboard to enter the character or digit associated with the desired item in response to a prompt in the lower left hand corner of the screen. The actions associated with the choice of a frame specific selection and a local pad are similar in that they are specified by the user who created the frame and/or localpads - MSF simply does what it is told. In contrast, the actions associated with the selection of a global pad are programmed into the MSF system and the user has no control over them.

Each option (or selection) of a DISPLAY frame has an action associated with it. This action is actually another frame number. When a frame specific selection is chosen by the user the MSF system accesses the associated frame number and examines the frame to see what should be done next. If this new frame is a DISPLAY frame then the old information disappears from the screen and the information in the new frame is displayed. As well as frame specific selections, any DISPLAY frame may have a LOCALPAD frame associated with it. This LOCALPAD frame contains information on what pads should be displayed and what actions should be taken if one is chosen.



```
The MSF System                                                  F-4

Welcome to the menu selection system, MSF, designed for use with FDB.
MSF facilitates access to frames of information which have been created
specifically for use with this system. More information is available
on the topics listed below.

 1. How to use the system

 2. How the system works

 3. How to set up your own database for use with MSF

 4. Who is working on MSF

 5. What subjects are currently available for viewing


                                                                 P-print

?

e-edit b-back n-next h-help m-mark r-return d-display g-goto    x-exit
```

Fig. 9 A typical MSF DISPLAY frame

The global and local pads and the frame options (called frame specific selections) all constitute items which the user can choose from. Selection is accomplished either by using a bit pad to position the cursor over the item of choice and depressing the z button or by using

Each local pad has an action (a frame number) associated with it and functions in exactly the same way as discussed for a frame specific selection. The purpose of having a separate LOCALPAD frame, instead of just including local pad information in a DISPLAY frame, is to allow several DISPLAY frames to share the same local pads. An examination of the meta structures of the DISPLAY and LOCALPAD frames (see Figure 10) will perhaps clarify this discussion.

```
       owner : 0
        type : META
         isa : 0
     fheader : DISPLAY
   textlines : EMPTY
  selections : EMPTY              owner : 0
       title : EMPTY               type : META
      text_0 : EMPTY                isa : 0
      text_1 : EMPTY            fheader : LOCALPAD
        .                       numpads : EMPTY
        .                        lpad_0 : EMPTY
     text_16 : EMPTY             lact_0 : EMPTY
    select_0 : EMPTY                .
    action_0 : EMPTY                .
    select_1 : EMPTY                .
    action_1 : EMPTY             lpad_3 : EMPTY
        .                        lact_3 : EMPTY
        .
    select_8 : EMPTY
    action_8 : EMPTY
   localpads : EMPTY
      marked : EMPTY
```

Fig. 10 The DISPLAY and LOCALPAD meta frames

The structure of the DISPLAY meta frame, as seen in figure 2, is fairly self explanatory. The slots for the title, text (text_0,...,text_16), and options (select_0,...,select_8) contain, in an ordinary frame of this type, the character strings which constitute the information display of the frame. The textlines and selections slots store the number of lines of text and options, respectively, that a frame contains. As can be seen, each select slot is accompanied by an action slot. Generally, this action slot is of type FRAME and contains the frame number of the action associated with the particular selection. Finally, the localpads slot holds the frame number of a LOCALPAD frame if that DISPLAY frame is to have localpads displayed with it. The meta structure of the LOCALPAD frame is similar to that of the DISPLAY frame. The lpad slots hold the character strings which are displayed as the pads. As for the select and action slots of the DISPLAY frame, each lpad slot has an lact slot associated with it which contains the number of the frame to be accessed if that pad is chosen. The numpads slot holds the number of pads, maximum four, which the LOCALPAD frame contains.

There are no restrictions on the frames associated with a local pad or frame specific selection aside from the fact that their meta type must be one recognised by the MSF system. (There will, however, generally be a logical connection between the information in the current frame and the contents of those accessible through selections from it.) In this way the user may establish a very complex network or directed graph of frames. Although at the present time the MSF system can only be used to display another frame as the response to a user defined selection, the groundwork has been laid for having a frame specific or localpad selection perform some other activity.

The global pads which appear at the bottom of the display screen offer the user a variety of options. Choosing the "help" pad will route the user to a network of frames which explain the MSF system and how to use it. As mentioned earlier, the network of frames a user may have access to can be very large and complex. Because of this several orientation aids have been provided. A user may "mark" frames which are important to him. As well as actually physically marking the frame by inserting the string "M-n" (for the nth marked frame) in the upper right hand corner of the display, this selection adds the frame to a list of frames which may be retrieved later. "Back" will return the user to the frame he was viewing prior to the one currently on display. When viewing a frame with several frame specific selections a user may decide he would like to go through all of them. In this case all he has to do is choose the first option and from there the global pad "next" can be used to switch to the second, third, etc.. The "return" option routes the user to a collection of frames which contain information on which frames he has recently seen and also those frames that he specifically marked. When "goto" is chosen the user is prompted for a frame number and if this frame exists and is a DISPLAY frame it will replace the frame currently on display. "Display" is useful for redisplaying the information in the current frame if for some reason the screen becomes garbled. The choice of "edit" puts the user into editor mode, a new set of global pads is displayed, and the user may add, remove, and alter frames as desired. When the user is finished with the MSF system, the choice of "exit" will return him to normal command mode.

5. FDB and a VLSI Design System

We are currently involved in the design and development of a system to support the computer aided design of VLSI circuits. Our approach to the design of such a system reflects the needs of current VLSI design systems [Mead and Conway 1980], [Losleben 1982], [Muroga 1982] and CAD systems in general [Goos and Hartmanis 1982].

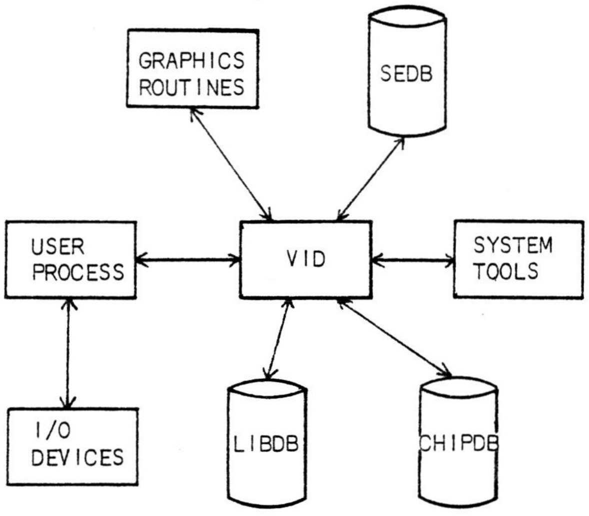


Fig. 11 Overview of VID

An overview of our VLSI interactive design system (VID) is shown in figure 11. We are implementing VID on a PDP 11/23 using an AED 512 raster graphics terminal. We are using a keyboard, a joy stick, and a bit pad as input devices. We have shown the system interacting with three FDB databases: SEDB, our system environment database representing a dynamic menu driven environment used to interface with the system, the chip database and various tools; CHIPDB, a database to store the various representations of the VLSI circuit during its design; and LIBDB, a library database to store references to other chips being designed, fabrication technology dependent information, and VID system data. We have grouped all possible tools under system tools. These include various tools for editing and viewing the chip, design rule checking, simulation, modifying the system environment and so on. This list is kept open to permit the addition of various tools as they are developed or acquired.

FDB plays an important role in the design of our system environment and the chip database. SEDB is important in storing the representation of different system interfaces. CHIPDB is important in storing the representation of the VLSI circuit through all phases of design. LIBDB provides a means of generality for system tools. We wish to present the design and implementation of the chip database to illustrate the application of FDB to VLSI design.

5.1. The VLSI Chip Database - CHIPDB

We are using a structured approach [Mead and Conway 1980] to VLSI design. This method is hierarchical in nature. The design proceeds in two stages: top-down and bottom-up. The initial design starts with a global view of the chip and subdivides this view until the basic primitives are to be defined. From here the design proceeds from the bottom up by implementation of the basic components of the cell. This lower level design propagates its effects through the higher levels of the hierarchy.

We will also base our model of the VLSI chip on a hierarchy. This structure has become popular in VLSI design and is described as a connective structure [Oakes 1979], a separate hierarchy [Trimberger 1981], and a deciduous tree [Daniel and Gwyn 1982]. We will refer to this structure as a nested cell model and will use it to model VLSI circuits.

Our nested cell structure will have three different types of nodes: the root node, representing the entire cell; composite nodes, representing information at the subcell level; and the leaf nodes which will hold all primitives necessary to describe the different representations of the chip.

This structure is basically a tree. There will exist only on root node per chip database. The root node will identify the fabrication process, describe its dimensions, and contain a list of subcells. A subcell contains the extent of the subcell, where it is to be placed in the parent cell, a reference to a cell definition, and a list of connectivity information. A cell definition can either be a composite or leaf node. A connectivity list displays the connections of a particular subcell to other subcells at the same level of the hierarchy. A composite node contains a description of its extent and a list of the subcells that comprise it. A leaf node contains its extent, placement information, and references to the different kinds of primitives used to describe the VLSI chips. This information will be graphical or textual in nature. We will represent symbolic or stick diagrams, mask layout geometry, and documentation at this level. Additional types of information can be added as needed. The root and composite node should contain at least one reference to a leaf subcell. This particular leaf subcell expresses the inter-connectivity of that particular node.

The root node meta frame is shown in figure 12. The NAME slot is used to give a name to the cell. The X_MAX and Y_MAX slots are used to give the dimensions, in microns, of the chip.

SCALE will contain the scale the chip was defined in. Technological information are stored in the TECH and LAMBDA slots. These are used to identify the technological dependent information that would be residing within the library data-base The SUBCELL# slot indicates the number of subcell references. A maximum of 20 subcells can be defined within this frame. However, an OVERFLOW slot which points to a frame is used if the number is exceeded. This frame contains a list of additional subcells. Each SUBCELL slot points to a frame containing a reference to a composite node or leaf node.

FHEADER	: ´root´
NAME	: ´CHIPXX´
X_MAX	: 4800.
Y_MAX	: 4800.
SCALE	: .01
TECH	: NMOS
LAMBDA	: 3.
SUBCELL#	: 0
SUBCELL0	:
:	:
SUBCELL19	:
OVERFLOW	: FO

Fig. 12 Root node meta frame

The subcell meta frame is shown in figure 13. Here the EXTENT slot describes the bounding box of the subcell. The TRANSFORM slot is used to store a transformation matrix. This matrix describes the placement of the subcell within its parent cell. The CELL_DEF slot is a pointer to either a leaf node frame or a composite node frame. The CONNECT_LIST slot is used to point to a list of frames. This list is comprised of subcells that interconnect with the subcell.

FHEADER	: ´subcell´
NAME	: ´CELLXX´
EXTENT	: a
TRANSFORM	: a
CELL_DEF	: FO
CONNECT_LIST	: FO

Fig. 13 Subcell meta frame

The composite node meta frame is set up similar to the root node. This frame will only contain an EXTENT slot and a list of SUBCELL slots. The SUBCELL slots are set up the same as those in the root node meta frame.

The leaf node meta frame is very simple. It has an EXTENT slot to describe the area it occupies and a list of slots which are set up to contain pointers to a particular representation. In our meta frame we have a STICKS slot fot a symbolic representation of a cell, a MASK slot for mask layout geometry, and a DOC slot for the cell´s documentation.

We will present one final meta frame to illustrate how we represent these primitives. Figure 14 shows the meta frame for the mask representation. Essentially, we have a list of slots for each layer in our technology. The slots shown are those used in NMOS which was declared as the fabrication technology in the root. Each of these slots points to a layer primitive frame that contains a list of all the shapes of a particular component defined within the boundaries of the leaf node.

FHEADER	: ´mask´
POLY1	: FO
POLY2	: FO
DIFF	: FO
METAL	: FO
CONTACT	: FO

Fig. 14 Mask meta frame

We have found it very natural to represent the nested cell model with FDB. The frames described describe the minimum structure necessary to implement this model within the system. One nice feature of FDB is the ability to modify or make additions to frames without any high overhead with respect to time or effort. This flexibility will allow the addition of other features we may wish to add to the cell or leaf descriptions. In addition, the use of the library makes our model more general in nature. This will allow designs of various technologies to be produced along with a general set of tools to operate on these designs.

6. Summary

In this paper we have shown how a data base system can be combined with graphics routines to provide a complete support system for data and display management. We have described the structure of the data bases we have been using and the precedures used to produce displays from these data bases. We have been using this graphical data base system for a number of years mainly in design oriented applications.

References

[Barr and Feigenbaum 1981] Barr A., E. Feigenbaum, The Handbook of Artificial Intelligence, Vol. 1, William Kaufmann Inc., Los

Altos, Ca., 1981.

[Daniel and Gwynn 1982] Daniel M.E., C.W. Gwyn, " CAD Systems for IC Design", IEEE Trans. CAD/ICAS, CAD-1, p.2, January, 1982.

[Date 1981] Date C.J., An Introduction to Database Systems, Addison-Wesley, Reading Mass., 1981.

[Foley and Van Dam 1982] Foley J.D., A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading Mass., 1982.

[Garret and Foley 1982] Garret M.T., J.D. Foley, "Graphics Programming Using a Database System with Dependency Declarations", ACM Transactions on Graphics, vol.1, no.2, p.109, 1982.

[Goos and Hartmanis] Goos G., J. Hartmanis, Computer Aided Design Modelling, Systems Engineering, CAD Systems, Springer Verlag, 1980.

[Lorie and Plouffe 1982] Lorie R., W. Plouffe, "Complex Objects and Their Use in Design Transactions", Research Report RJ 3706, IBM Research Laboratory, San Jose, Ca, 1982.

[Losleben 1982] "Losleben P., "Computer Aided Design for VLSI", in VLSI: Fundamentals and Applications, Springer-Verlag, 1982.

[Mead and Conway 1980] Mead C.A., L. Conway, Introduction to VLSI Systems, Addison-Welsesly Publishing Company, 1980.

[Maruga 1982] Maruga S., VLSI System Design, John Wiley and Sons, Inc. 1982.

[Newell 1977] Newell A., "Notes for a Model of Human Performance in ZOG, Carnegie-Mellon University Technical Report, 1977.

[Oakes 1979] Oakes M.F., "The Complete VLSI Design System", Proceedings, 16th Design Automation Conference, June 1979.

[Robertson, McCracken and Newell 1981] Robertson G., D. McCracken, A. Newell, "The ZOG Approach to Man-Machine Communication", International Journal of Man-Machine Studies, vol. 14, p.461-488.

[Trimberger 1981] Trimberger S., J.A. Rowson, C.R. Lang, J.P. Gray, "A Structural Design Methodology and Associated Software Tools", IEEE Trans. CAS-28, p.618, July 1981.

[Weller and Williams 1976] Weller D., R. Williams, "Graphic and Relational Data Base Support for Problem Solving", Computer Graphics, vol.10, no.2, p.183, 1976.