Database Support for Interactive Computer Graphics

David L. Spooner
Mathematical Sciences Department
Rensselaer Polytechnic Institute
Troy, NY. 12181

Abstract: Software applications involving interactive computer graphics are made up of several components. One of these is a data modeling component. This data modeling component is often application dependent and therefore difficult to integrate with the data model of other related applications. One solution to this problem is to interface a DBMS with an interactive graphics system to reduce data model dependency and increase sharing of data. This also has the advantage of allowing novice users to use computer graphics by eliminating the need for the user to write complex programs invoking the graphics system.

This paper explores these ideas by discussing the implementation of a prototype interface between a relational DBMS and an interactive computer graphics system. It presents both the database structures used to manage the data and the techniques used to design the interface. It concludes by discussing an approach for making the interface portable.

1. Introduction

An application program using interactive computer graphics consists of three major components: a data modeling component, a graphics component, and an interface component which connects the data model to the graphics routines [Fole82]. A major attempt has been made to standardize the graphics component of such programs with the development of the CORE graphics system [GSPC77] and the Graphics Kernel System (GKS) [ISO82]. The only attempt to standardize the data modeling and interface components is the Initial Graphics Exchange Specification (IGES) [Liew82] which is designed for data exchange between databases. As a result, the data modeling components of many application programs using computer graphics are very application dependent. See Figure 1.

This lack of a standardized general purpose data model and interface for interactive graphics application programs creates some major software development problems. Sharing of data between two independent applications is difficult. Cooperation and integration of related applications may be severely limited due to incompatible data models. Maintenance of application programs is made difficult because of the lack of standardization.

The dependency of a graphics application on a particular data model is similar to the data model dependency problem that occurred among information processing applications several years ago. That problem was solved with the development of general-purpose database management systems (DBMSs). DBMSs can also be used to eliminate much of the data model dependency present in interactive graphics application programs.

This paper discusses a software interface between a DBMS and a standardized graphics system (e.g. CORE or GKS). This interface allows different graphics application programs to share data, and new applications to be more easily created and integrated with existing applications. The goal of the system is the management of the data used to draw pictures, not the management of digitized pictures as is done in image processing applications. The system is interactive so that changes to data in the DBMS produce appropriate changes in a displayed picture and vice versa.

The interface has another major advantage. It opens up the world of interactive computer graphics to the novice user by eliminating the need to write complex programs to use the graphics system. The interface handles all the details of actually displaying a picture.

A prototype interface has been implemented allowing data stored in the

relational DBMS RIM [Boei81] to be displayed using the DYNAGRAPHICS [RPI81] interactive graphics system in the Center for Interactive Computer Graphics at Rensselaer Polytechnic Institute. This paper presents the design of the prototype interface and discusses its extension to other types of database systems.

In the next section we discuss the techniques used for interfacing a DBMS and graphics system. Section 3 further describes the prototype interface. Section 4 briefly discusses techniques which can be used to create a portable interface for a variety of DBMSs and graphics systems.

2. Interfacing Issues

The long range goal of this work is to develop a general interface which will work with a variety of DBMSs and interactive graphics systems. We are not attempting to integrate a specific DBMS with a specific graphics system. Therefore flexibility and portability are major goals.

The motivation for designing an interface rather than attempting to integrate a DBMS with a graphics system is largely one of cost-effectiveness. Many organizations currently have a DBMS and an interactive graphics system in house. It is more cost-effective for such organizations to interface these two existing systems than it is for them to import a special-purpose integrated graphics database system.

Several researchers have suggested ways of integrating a DBMS with an interactive graphics system. The Picture Building System [Well76] is the first example of such a system. This system uses a relational database to store graphics commands for drawing a picture. Lorie [Lori82] developed a VLSI design system using the Picture Building System. The Spatial Data Management System [Hero80] integrates a DBMS with a graphics system, but stresses data retrieval and browsing of the database, rather than picture construction and display. Finally, Garrett [Garr80] developed a system with capablities similar to the Picture Building System, and uses the idea of "triggers" to interactively update both the data in the database as well as a displayed picture.

Since our goal is a portable interface rather than an integrated graphics database system, our system differs in design from each of these. It differs from the Picture Building System in that it makes heavy use of system-supported hierarchically organized data often referred to as "complex objects" in the literature [Lori82], and uses the "picture segmentation" facilities available in many interactive graphics systems. (A picture segment is a subpicture which can be displayed and manipulated independently from the rest of a displayed picture. It is the fundamental unit for picture composition in interactive graphics systems such as CORE and GKS.) As will be seen below, this combination of complex objects and picture segmentation simplifies the design of the interface. Our interface differs from Lorie's VLSI design system in that it extends the idea of complex objects to a general interface for graphics applications. It differs substantially from the Spatial Data Management System since it is concerned with picture manipulation rather than browsing through a database. It differs from Garrett's system primarily in implementation techniques. Garrett presents some interesting ideas for the design of an integrated DBMS and graphics system. However, these ideas do not seem to generalize nicely to meet our goals of portability and flexiblity for the data model.

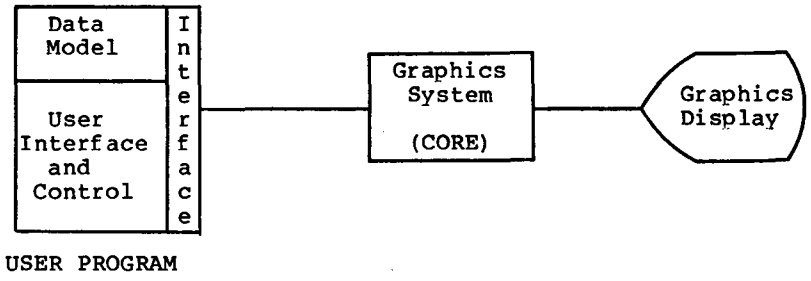
The major contribution of this work is the adaptation of selected features from each of these systems to develop a technique for interfacing existing DBMSs to existing graphics systems. The resultant technique has the advantage of being independent of any particular DBMS or graphics system. An application of this technique to a specific DBMS and graphics system is demonstrated by the prototype interface described in this paper, and the design approach is thereby validated. In Section 4, we discuss the way in which the prototype interface will be extended to be portable across different DBMSs and graphics systems.

2.1 Complex Objects

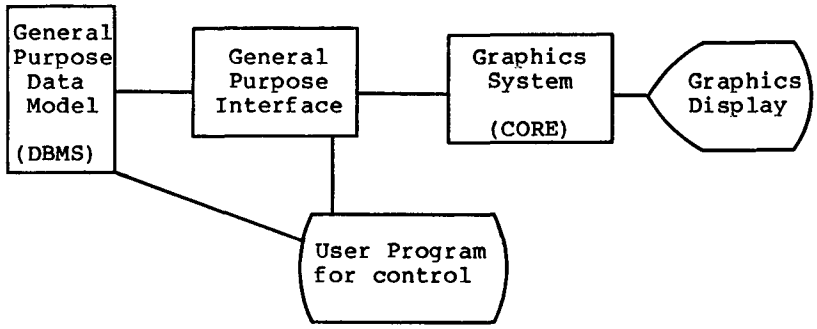
Lorie [Lori82] and others have argued the virtues of using complex objects to represent graphical data. A complex object can be thought of as a hierarchically organized collection of data describing an object. This idea works well for graphical data because most pictures are hierarchical in nature. For example, consider an organization for data from a blueprint or floor plan of a building. Such data may be organized into the hierarchy in Figure 2.

In this example, each building is composed of one or more floors. Each floor is composed of one or more rooms and/or hallways. Each room has windows, doors, and furniture, and so on.

When one applies the idea of complex objects to a relational DBMS, one gets a hierarchy of nested relations. If relation B is nested as a child of relation A, then every tuple of B must have a parent tuple in A. To represent this structure in the database, it is



a) Application dependent data model



b) Application independent data model

Figure 1: The Software Components of a Graphics Application

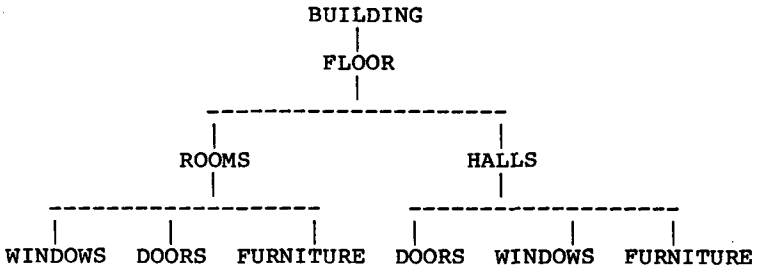


Figure 2: Hierarchical Organization of Floor Plan Data

necessary to introduce unique ID numbers for each tuple of each relation. Then each tuple in relation B, in addition to containing its own unique ID number, contains the ID number of its parent tuple in relation A (i.e. each tuple has a pointer to its parent in the complex object).

The prototype interface adopts the idea of complex objects. However, it is necessary to make one important extension to their definition. This extension stems from the fact that in many cases it is convenient to violate the strictly hierarchical requirement imposed on complex objects. In Figure 2, notice that windows, doors, and furniture occur in both rooms and hallways. It seems most logical to combine all windows into one WINDOW relation, all doors into one DOOR relation, and all furniture into one FURNITURE relation. This violates the hierarchical organization of the relations.

In addition, consider a door between a room and a hallway. This door is shared by both the room and the hallway, but logically there should be only one tuple in the DOOR relation for this door. This tuple must have two parents in the complex object, the tuple for the room containing the door, and the tuple for the hallway containing the door.

This type of violation of the hierarchical organization for complex objects must be allowed in the prototype interface. However, since a tuple may have many parents in this situation, a single parent pointer in each tuple is not sufficient for defining the structure of the complex object. To solve this problem, special link relations are introduced into the database for relations which contain tuples with more than one parent. There is one link relation for every possible parent relation in such situations. Each tuple of a link relation contains the unique ID number of a child tuple and the unique ID number of its parent from a particular parent relation. For example, there is one link relation to link tuples from relation DOOR to their parents in relation ROOM, and another link relation to link the same tuples in relation DOOR to their parents in relation HALL.

Figure 3 shows a database schema for the data in Figure 2 using this complex object organization. The last six relations are link relations to link ROOM and HALL to each of WINDOW, DOOR, and FURNITURE. Note that the first column of all relations, except the link relations, contains the unique ID numbers for the tuples.

2.2 Semantic Attributes

A general purpose interface between a DBMS and a graphics system must be told which data in the database has graphical meaning and which does not. It is undesirable to force the user to use specific columns of a relation or specific column names for this purpose. To handle this problem, Weller and Williams [Well76] introduce the idea of graphical semantic attributes for the columns of a relation. These attributes are part of the schema for the database and indicate to the interface not only which data has graphical meaning, but also the type of graphical meaning (e.g. x translation, y translation, x scale factor, etc.). The prototype interface adopts this idea and extends it to include the idea of complex objects. If the interface is to be able to interpret complex objects, it must be able to identify which column of a relation contains the unique ID numbers for the tuples of the relation and which column contains the ID numbers for the parent tuples. For link relations, it must identify which column contains the link to the parent tuple and which column contains the link to the child tuple. This information is defined for the interface using three structure semantic attributes.

The graphical and structural semantic attributes for the relations in a database must be stored where the interface can easily refer to them. A reasonable place to store them is in the database. Therefore the interface expects the database to contain a relation named SEMANTICS with the structure shown in Figure 4. This relation lists the data type, semantic attribute, and byte address within a tuple for each column of each relation in the database. If two or more applications wish to interpret the data in the database differently, they need only use different SEMANTICS relations.

2.3 Mapping to the Graphics System

When complex objects are used to model the data from which a picture is drawn, every tuple in the complex object represents a separate entity. Collectively, these entities form the picture. For example, each tuple from the complex object defined in Figure 3 represents a building, floor, room, door, window, or piece of furniture. More than this, the complex object also represents the nesting of these entities. Since ROOM is a child of FLOOR, rooms must be drawn internally to a floor. Since WINDOW, DOOR, and FURNITURE are children of ROOM, they must each be drawn internally to rooms.

This nesting of entities represented by the tuples of a complex object can

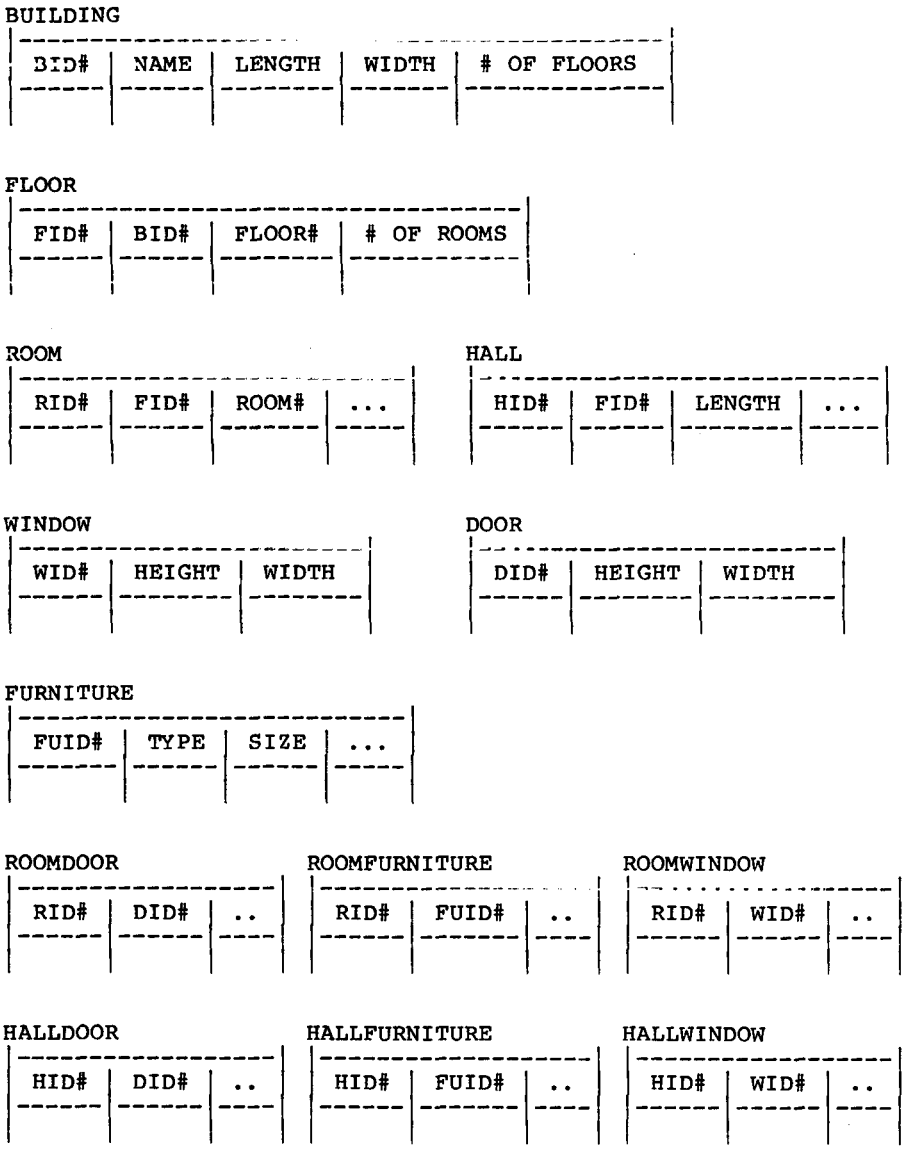


Figure 3: Relational Database of Floor Plan Data

easily be mapped into the interactive graphics system using the idea of segments and nested segments as discussed below. The mapping into the proposed hierarchical graphics standard, PHIGS [ANSI83], should be even easier since PHIGS organizes graphical data internally as a hierarchy. This has not been investigated, however.

Recall that a segment is a subpicture that may be displayed and manipulated independently from the rest of a displayed picture. Each segment may have one or more "display points" defined for it in the coordinate system of the graphics terminal. The subpicture contained in the segment is drawn at each of these display points. A segment may have other segments defined inside it, and displayed at specifed points within the subpicture of the outer segment. Thus, if each entity in a complex object is displayed in a separate segment, and if the segments are nested to match the nesting of entities in the complex object, then the internal data structure for the graphics system closely matches the structure of a complex object and can be created and manipulated very efficiently. The CORE, GKS, and DYNAGRAPHICS graphics systems all include this concept of segments. The GKS and DYNAGRAPHICS systems allow nesting of segments. Segment nesting must be simulated in CORE systems.

The use of nested segments by the graphics system also makes editing a picture easier. In a graphics system such as the DYNAGRAPHICS system, when a light pen or similar device is used to select part of a picture, the graphics system identifies the inner most nested segment included in that part of the picture as well as all segments which are ancestors of that segment. This simplifies the identification of the selected entity in the database because it defines the path from the root of the complex object down to the entity represented in the selected segment.

2.4 The POINTS Relation

As yet no mention has been made of how the geometry of an entity is stored in the database. This geometry is required to make drawing the entity possible. Since the prototype interface currently handles only two-dimensional pictures, it requires only the ability to store line segments and arcs describing a particular entity. The geometry of an entity, therefore, is a repeating group of line and arc segments. In relational database systems, a repeating group is usually represented as a separate relation where each tuple represents one member of the repeating group. Using this idea for the prototype interface, two representations are possible. Either the geometry for each entity can be represented by a separate relation, or all geometries can be represented in a single relation with all the tuples defining a particular geometry identified by a common key value. The latter approach is used in the prototype interface primarily because it is more easily implemented in the RIM DBMS. Hence the geometries of all the entities are stored in a single relation named POINTS. These geometries are associated with the entities represented by the tuples of a relation in a complex object via a special semantic attribute. This semantic attribute identifies the column in the relation containing the keys into the POINTS relation.

The POINTS relation is clearly somewhat limited in scope. Relations could be allowed to name entity drawing procedures like in the Picture Building System, but this mechanism violates the goal of simplifying the use of the graphics system, since it requires the user to write graphics programs. Much research is needed to identify more flexible mechanisms than the POINTS relation for representing the geometries of entities in the DBMS, while preserving the goal of simplifying the use of a graphics system. This is especially true for 3-dimensional pictures.

2.5 Drawing a Picture

To complete the discussion of interfacing issues, it is necessary to discuss the way in which the data in the database is used for drawing pictures. We begin with a brief summary of the major ideas from above, and then show how an interface can use these ideas to "interpret" the data in the database to construct a picture.

Every tuple of every relation represents an entity which may be drawn. Some of the attributes of each tuple represent graphical data, some represent structural data for complex objects, and some are simply descriptive data that are irrelevant to the interface. The SEMANTICS relation is used by the interface to distinguish the types of the various attributes in a tuple of a relation.

Every tuple must be associate with a geometry in the POINTS relation. The POINTS relation contains all the line or arc segments needed to draw the entity. A tuple may specify scaling factors which are used to scale all the segments in the geometry of the entity before it is drawn. In this way, different tuples may share the same geometry, but may scale it differently to adapt it to their individual requirements.

A tuple may also contain x and y translations and a rotation. In order for these to make sense, a reference point is needed. The lower lefthand corner of the

parent entity is used as the reference point for translating line and arc segments. Rotation is done about an entity's lower lefthand corner. All appropriate transformations for ancestors of an entity in a complex object must be done to the entity, in order from youngest to oldest ancestor, to place the pictoral representation of the entity into the displayed picture at the appropriate location.

For entities with more than one parent in a complex object, it does not make sense to put scale factors, translations, and rotations into the tuple representing the entity. The transformations may differ for the different parents. Therefore, these transformations are put in the link relations which are unique for each parent.

3. The Prototype System

The prototype interface consists of two parts: a graphics interpreter, and a graphics editor (See Figure 5). The graphics interpreter is responsible for interpreting the data in a complex object and drawing the picture of the complex object on the screen. It uses all of the techniques discussed above to accomplish these tasks. The graphics editor is responsible for providing editing functions which allow alteration of a displayed picture, and also for propagating these alterations back into the database. For more details on the graphics editor, see [Spoo83]. All software is written in FORTRAN 77.

3.1 Using the Prototype Interface

One of the major advantages of using a DBMS, such as RIM, as the data modeling component of a graphics application is that use can be made of the high-level query language of the DBMS. Many relational DBMSs allow a user to retrieve an arbitrary subset of the rows and columns of a relation simply by defining a Boolean predicate which describes the desired subset. Thus the user has great flexibility in selecting exactly what is to be included in a displayed picture by using the query language to select the relevant data to be given to the interface to the graphics system.

For example, using the query language QUEL [Ston76] and the database in Figure 3, if a user wants to display all of the rooms and halls on the third floor of building 10, he might enter the queries:

```
Range of f is FLOOR
Range of r is ROOM
Range of h is HALL

Retrieve (f.all) where f.FLOOR# = 3 and
                       f.BID# = 10
Retrieve (r.all) where r.FID# = f.FID# and
                       f.FLOOR# = 3 and
                       f.BID# = 10
Retrieve (h.all) where h.FID# = f.FID# and
                       f.FLOOR# = 3 and
                       f.BID# = 10
```

At first glance this appears more complicated than necessary, but it must be recognized that most of the complexity arises because QUEL is not equipped to handle complex objects directly. If the QUEL query language is modified so that it "understands" complex object structures, then this query might be simplified to:

```
Range of f is FLOOR
Range of r is ROOM
Range of h is HALL

Retrieve (f.all) where f.FLOOR# = 3
                   and f.BID# = 10
Retrieve Within Parent (r.all)
Retrieve Within Parent (h.all)
```

A query language similar to this which uses menus has been developed for the prototype interface.

The query language processor prompts the user to identify which complex object type he wishes to draw. It then allows the user to define "where" clauses, to select only the desired parts of the selected complex object type. The query processor also serves as the control program for the prototype interface. It queries the user and invokes the graphics interpreter and graphics editor (see [Spoo83]) repetitively until the user indicates he wishes to stop. Hence the user can draw many pictures by working at a high level with high level entities without dealing directly with the graphics system.

4. Extensions to Other Database Models

Ideally, an interface system is required that can support both a variety of DBMSs which use different data models and a variety of interactive graphics systems. Future work must build on what was learned from the prototype interface and generalize these ideas in a general-purpose interface.

The graphics side of the interface must be compatible with the proposed standards for interactive graphics systems, such as CORE and GKS. The database side is not so clear, however. No data models have as yet been accepted as standards. In fact, two different DBMSs supporting the same basic data model may be very different in capabilities and user interfaces. The relational data model is gaining wide acceptance for engineering applications, yet there are

SEMANTICS

RELATION NAME	COLUMN NAME	DATA TYPE	SEMANTIC ATTRIBUTE	START LOCATION
ROOM	RID#	INT	ID#	0
ROOM	FID#	INT	PID#	4
ROOM	ROOM#	INT		8
ROOM	PHONE	TEXT		12
ROOM	LENGTH	REAL	XS	20
ROOM	WIDTH	REAL	YS	24
ROOM	XLOC	REAL	XCOR	28
ROOM	YLOC	REAL	YCOR	32
.	.	.	.	.

Semantic Attributes:
ID# - unique ID number for the tuple
PID# - ID number for the parent of the tuple
XS, YS - x and y scale factors
XCOR, YCOR - x and y translation within parent

Figure 4: SEMANTICS Relation for the Database in Figure 3

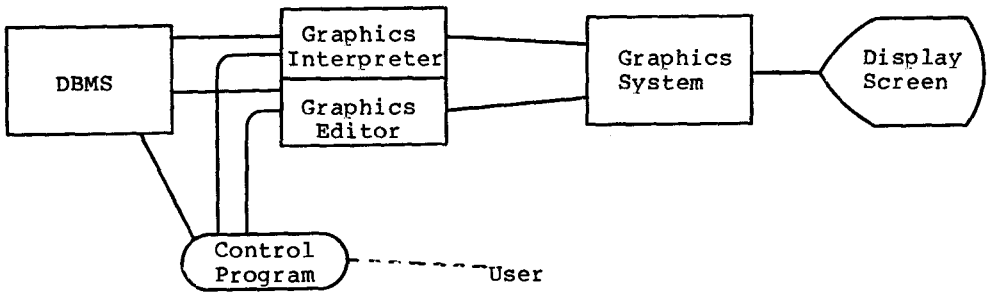


Figure 5: Design of the Prototype Interface

still many network and hierarchical DBMSs in use today. Thus techniques must be developed to deal with this wide variety of data models and DBMSs.

The only viable solution seems to be to define the interface using a very high-level data model. One likely candidate is a functional data model such as DAPLEX [Ship79]. Special routines can then be provided which will do the mapping between the high-level data model in the interface and the data model of a particular DBMS. This is similar to the approach used to handle different data models in the design of Multibase, a heterogeneous distributed database system [Smit81].

The reason for choosing a functional data model such as DAPLEX for the interface, is that it makes definition and manipulation of complex objects very straightforward and simple. A DAPLEX database consists of two constructs, entities and functions. Entities represent real world objects (e.g. floors, room, halls, etc.) and functions map entities to entities. To specify that rooms and halls are nested under floors in a complex object, one simply defines a function from floors to rooms and another function from floors to halls.

```
DECLARE  ROOMSOF(FLOOR) ==>> ROOM
DECLARE  HALLSOF(FLOOR) ==>> HALL
```

Given a particular floor entity, the ROOMSOF function returns all the entities for rooms on that floor. Similarly, the HALLSOF function returns the entities for all halls on the floor.

To specify that a door may belong to both a room and a hallway, one defines a function from room entities to door entities, and a function from hall entities to door entities.

```
DECLARE  DOORSOF(ROOM) ==>> DOOR
DECLARE  DOORSOF(HALL) ==>> DOOR
```

The instance of the door entity type for the shared door is then assigned to the range of both functions.

In addition, traversal of a complex object structure is facilitated using DAPLEX because of the use of function composition in the query language. For example, to retrieve all the doors on a particular floor, one might write:

```
DOORSOF( ROOMSOF (floor entity))  union
     DOORSOF( HALLSOF( floor entity))
```

The reader is invited to consult [Spoo83] for a more detailed example of the use of DAPLEX for the graphical interface.

5. Conclusions

In the first stage of this project we have developed a prototype interface between a relational DBMS and an interactive graphics system. The interface is capable of drawing 2-dimensional pictures from data stored in the database. It differs from other work in this area in that it is designed to be a portable interface between a DBMS and graphics system, rather than an integrated graphics database system. It is also unique in its coordinated use of complex object structures in the database, and picture segmentation facilities in the graphics system.

This prototype will serve as a test vehicle for a detailed study of the important issues in the design of a general-purpose interface. Future work will concentrate on generalizing these techniques for a variety of DBMS and graphics systems. This will involve designing the interface using a higher level data model than is supported by a typical DBMS. A high-level functional data model such as DAPLEX seems appropriate for this task. Mapping functions between this data model and any particular DBMS and graphics system can then be defined.

Acknowledgement: I would like to thank M.S. Krishnamoorthy and Moon-Jung Chung who have helped clarify ideas during many discussions. I would also like to thank Steven J. Shaer, Paowen Lee, and Carol Hallock who have done much of the implementation of the prototype interface.

References

[ANSI83] American National Standards Institute, Programmer's Hierarchical Interactive Graphics Standard. X3H3.1 Task Group, personal communication, 1983.

[Boei81] "User Guide: RIM 5.0 Prime PRIMOS." Boeing Commercial Airplane Company, Seattle, Washington, 1981.

[Fole82] Foley, J. D., and A. Van Dam, Fundamentals of Interactive Computer Graphics. Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[Garr80] Garrett, M., A Unified Non-procedural Environment for Designing and Implementing Graphical Interfaces to Relation Data Base Management Systems. Ph.D. Thesis, Dept of EE and CS, The George Washington University, Washington, D.C., 1980.

[GSPC77] "Status Report of the Graphics

Standards Planning Committee of ACM/SIGGRAPH." ACM SIGGRAPH Computer Graphics, Vol. 11, No. 3, Fall 1977.

[Hero80] Herot, C. F., "Spatial Management of Data." ACM Trans. on Database Systems, Vol. 5, No. 4, Dec. 1980, pp. 493-514.

[ISO82] International Standards Organization, "Graphics Kernel System (GKS) Functional Description." ISO TC97/SC5/WG2, X3H3/82-10.

[Liew82] Liewald, M., and P. Kennicott, "Intersystem Data Transfer via IGES." IEEE Computer Graphics and Applications, May, 1982, pp. 55-63.

[Lori82] Lorie, R., "Issues in Databases for Design Applications." File Structures and Data Bases for CAD, edited by J. Encarnacao and F.-L. Krause, North-Holland Publishing Company, Amsterdam, 1982, pp. 213-222.

[RPI81] "Center for Interactive Computer Graphics Users Manual." Rensselaer Polytechnic Institute, December, 1981.

[Ship79] Shipman, D., "The Functional Data Model DAPLEX." ACM SIGMOD International Conference Supplement, 1979, pp. 1-19, (also appeared in ACM Transactions on Database Systems, Vol. 6, No. 1, 1981).

[Smit81] Smith, J. M., et. al., "Multibase - Integrating Heterogeneous Distributed Database Systems." Proceedings National Computer Conference, 1981, pp. 487-499.

[Spoo83] Spooner, D., "Designing an Interface for a Database System and a Graphics System." In preparation.

[Ston76] Stonebraker, M., E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES." ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 189-222.

[Well76] Weller, D., and R. Williams, "Graphics and Database Support for Problem Solving." ACM SIGGRAPH Computer Graphics, Vol. 10, Summer 1976, pp. 183-189.