The Design and Implementation of a Database Environment for Vision Research[*]

Rex M. Jakobovits

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195

Abstract

The Database Environment for Vision Research (DEVR) is an entity-oriented scientific database system based on a hierarchical relational data model (HRS). This paper describes the design and implementation of the data definition language, the application programmer's interface, and the query mechanism of the DEVR system.

DEVR provides a dynamic data definition language for modeling image and vision data, which can be integrated with existing image processing and vision applications. Schema definitions can be fully interleaved with data manipulation, without requiring recompilation. In addition, DEVR provides a powerful application programmer's interface that regulates data access and schema definition, maintains indexes, and enforces type safety and data integrity.

The system supports multi-level queries based on recursive constraint trees. A set of HRS entities of a given type is filtered through a network of constraints corresponding to the parts, properties, and relations of that type. Queries can be constructed interactively with a menu-driven interface, or they can be dynamically generated within a vision application using the programmer's interface. Query objects are persistent and reusable. Users may keep libraries of query templates, which can be built incrementally, tested separately, cloned, and linked together to form more complex queries.

[*]Submitted in fulfillment of the Ph.D. Qualifying Project requirement. This research was sponsored by the National Science Foundation under grant IRI-9116809, and by the Washington Technology Center. A shorter version of this paper to appear in *IS&T/SPIE Symposium on Electronic Imaging: Science and Technology, February 1995.*

Contents

1 INTRODUCTION

Experimentation with image-related data often involves a number of disjoint applications, each with their own internal data representation, communicating via flat files. This approach leads to redundant processing, as data is marshalled in and out of the various structures and the file system. Furthermore, it tends to result in an unwieldy collection of cryptic disk files that the researcher must manage, making it difficult to browse and correlate the intermediate data.

The Database Environment for Vision Research (DEVR) is an entity-oriented scientific database system designed to facilitate experimentation with image-related data[8]. The system provides a framework in which computer vision researchers may structure their internal data to promote interoperability between applications. DEVR frees the researcher from having to manage data at the file system level, and it enables the user to formulate sophisticated queries across all aspects of the experiment process.

I implemented DEVR on top of the Object Database Environment (ODE)[1], a persistent C++ system. Objects are saved to disk automatically, and retrieved into memory whenever they are referenced within an application. The DEVR library consists of over 5000 lines of O++ code[3], including the application programmer's interface and the textual user interface.

DEVR offers a dynamic data definition language for modeling image and vision data. I have designed and implemented a powerful application programmer's interface, which allows users to integrate the database with existing image processing and vision applications. I have also implemented a textual user interface, which supports interactive browsing and template-driven query construction. DEVR's query processor supports a wide range of multi-level queries on complex user-defined types.

2 THE HRS DATA MODEL

2.1 Components of the HRS

DEVR is based on a hierarchical relational data model (HRS) which has evolved from the Relational Data Structure (RDS) of Shapiro and Haralick[9]. Every entity in the system (images, regions, edges, etc.) is described by a schema consisting of three components: *properties*, *parts*, and *relations*.

The properties component of a schema is a table of attribute definitions, where each entry specifies an attribute label and declares its type. Properties may be either atomic (i.e. float, integer, string, etc.), or complex types. Instances of the HRS defined by a schema will have attribute *values* that correspond to the entries in the schema's property table. These values record global information about the entity, such as the number of rows in an image, or the slope of an edge.

The parts component consists of any number of part sets, which are collections of pointers to other entities in the system. This allows the user to represent the natural decomposition of spatial and image data in an organized hierarchy. For example, a **View_Class** HRS may be defined to contain an **Images** part set, which in turn may contain an **Edges** part set, etc.

The relations component consists of attributed relations over the parts of that entity. Each relation is made up of a set of tuples. A tuple consists of an ordered list of pointers to entities in the parts sets, and an optional list of attributes. For example, the **Image** HRS may contain a **proximity** relation, whose tuples consist of pairs of pointers to edges and a numeric attribute describing the distance between them.

2.2 Example Application: TRIBORS

The HRS data model has been successfully used to support a number of vision applications. Figure 1 shows some of the data types used in the *Triplet-Based Object Recognition System* (TRIBORS)[7], an application that uses synthetic images to create probability models for use in 3D object recognition. TRIBORS was originally implemented without the HRS model or the DEVR system, using arbitrary data structures and ASCII file dumps to maintain data between executions. The input image files were scattered in various directories maintained by the system's designer. The HRS model easily supported the TRIBORS data types, and DEVR's application programmer's interface was used to link TRIBORS with the database. DEVR maintained the input images, synthetic images, and internal data structures (such as extracted edges).

Each 3D object to be processed by TRIBORS is represented by a **Tribors_Object** HRS, which consists of a 3D model and a set of view classes. The **Model_3D** HRS is decomposed into faces, edges, and points. Each **View_Class** HRS contains a number of real images and synthetic images, which in turn reference viewable gray scale images. TRIBORS generates a probability model for each view class, which is stored as a relation in the HRS for that view class. Each tuple of the probability relation consists of a triple of edges from the model, with attributes describing the orientation of the segments, the frequency of the triple's occurrence within the images for that view class, etc.

Actual CAD models from TRIBORS experiments has been successfully imported into DEVR, including multiple view classes consisting of over a hundred images and their corresponding spatial entities.

3 IMPLEMENTING THE HRS

The HRS model is implemented as a hierarchy of C++ classes [10], as shown in figure 2. All objects stored in the database descend from the **pBase** class, which defines generic methods for printing and identification. Because of the object-oriented design of the type system, DEVR can be cleanly extended to incorporate new atomic types and functionality, making full use of the automatic dispatching and type checking functionality of C++.

3.1 The Schema Class

One subclass of pBase is the **Schema** class, whose instances represent the HRS types defined in the database. Each Schema object is assigned a unique name when it is created. The system provides direct, efficient retrieval of a Schema object via its name.

Each Schema object stores the labels and type information that denote the properties, parts, and relations of an HRS type. In addition, it maintains the *type extent*, a collection of pointers to every instance of that type. Whenever the user constructs a new entity, the system inserts a pointer to it in the type extent of its schema. Conversely, when an entity is destroyed, the system automatically removes the pointer from its schema's type extent. The extent provides an efficient means for navigating over all HRS entities of a particular type.

Tribors_Object

PROPERTIES

model	*Model_3D*

PARTS

view_classes	*Set of View_Class*

Face_3D

PARTS

border	*List of Edge_3D*

Edge_3D

PROPERTIES

startp	*Point_3D*
endp	*Point_3D*

Point_3D

PROPERTIES

x_coord	*Real*
y_coord	*Real*
z_coord	*Real*

Real_Image

PROPERTIES

the_object	*Tribors_Object*
origin_viewpoint	*Point_3D*
distance	*Real*
sequence_num	*Integer*
image	*Gray_Scale_Image*

Model_3D

PROPERTIES

the_object	*Tribors_Object*

PARTS

faces	*Set of Face_3D*
edges	*Set of Edge_3D*
points	*Set of Point_3D*

View_Class

PROPERTIES

the_object	*Tribors_Object*
min_lat	*Real*
max_lat	*Real*
min_long	*Real*
max_long	*Real*

PARTS

model_edges	*Set of Edge_3D*
real_images	*Set of Real_Image*
synth_images	*Set of Synth_Image*

RELATIONS

real_probability_model

edge1	edge2	edge3
Edge_3D	*Edge_3D*	*Edge_3D*

ATTRIBUTES

frequency	*Real*
view_length	*Stat_Record*
orientation	*Stat_Record*
distance	*Stat_Record*
theta	*Stat_Record*

synth_probability_model

(similar to real_probability_model)

Figure 1: Schemas from the TRIBORS Application.

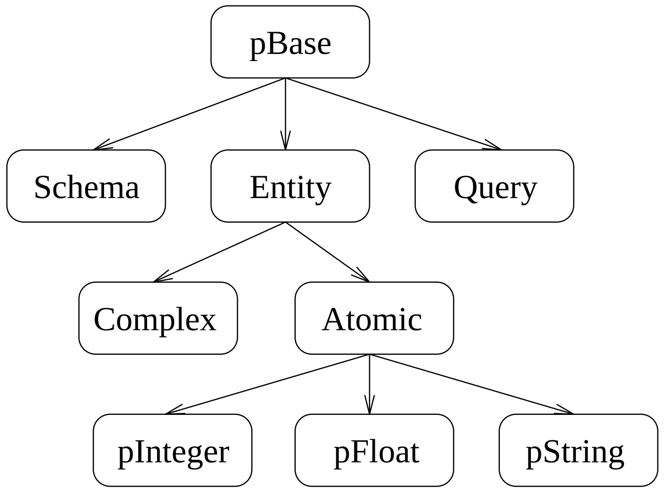


Figure 2: The class hierarchy in DEVR.

New data types are recursively constructed from existing types. DEVR supports self-referencing schemas and circular type relationships. For example, consider the TRIBORS system described in figure 1. An **Object_3D** references a **Model_3D**, while the **Model_3D** refers back to the **Object_3D** which it models. When defining components for a new type, the user may refer to partially completed types.

3.2 The Entity Class

All data instances are derived from the Entity class. They can be either atomic (floats, integers, strings, etc.), or complex entities (user-defined types). The components of a complex entity may contain pointers to other entities, so each entity is a network of sub-entities.

Entities are comprised completely of raw data (i.e. arrays of pointers to other entities); all type-specific information (such as field labels and types) is relegated to the schema object, thereby relieving complex entities from having to store any redundant information. This allows entities to be as compact as possible, which is especially important in vision applications that may involve thousands of entities.

All complex entities are implemented using a single C++ class, regardless of their specific type. Complex types are available immediately; entities may be generated as soon as a type definition is completed, without requiring recompilation of the system. By enabling fine-grained interleaving of type definition and data entry, DEVR promotes flexible user interaction and allows versatile real-time transactions. Furthermore, by treating data definition and manipulation uniformly, application development is simplified.

Each complex entity contains a pointer to the schema identifying its type. In fact, every type declaration is actually implemented as a pointer to a specific schema. For example, the property table of the **Edge_3D** schema contains an entry called **startp** whose type is indicated by a pointer to the **Point_3D** schema. This approach incurs minimal overhead during type checking. Consider the case where a user attempts to assign the **startp** property of an **Edge_3D** entity **E1** to an entity **E2** which is not a **Point_3D**. The system can detect this by simply comparing the schema pointer of **E2** against the pointer in the **startp** entry of the **Edge_3D** property table.

4 THE PROGRAMMER'S INTERFACE

The DEVR system provides a powerful application programmer's interface, called the *MetaMaster*, which can be linked with C++ applications. By regulating schema definition and data access, the MetaMaster enforces type-safety, maintains indexes, and guarantees data integrity. For example, the MetaMaster does not allow a user to destroy a schema unless all instances of that Schema have been deleted.

In addition, the MetaMaster provides a convenient template-driven interface for constructing new types and cloning entities. A new schema may be created by passing an existing schema as a parameter to the MetaMaster method **create_new_schema()**. The new schema starts with all the components of the template schema, and can then be edited as necessary. Similarly, the MetaMaster has methods to create an identical copy of an entity.

Unlike traditional query languages which tend to impose restrictions on the generality of data processing[2], the MetaMaster interface is a self-contained C++ object that can interact seamlessly in any C++ program. It provides all the flexibility of a general programming language, thereby avoiding the impedance mismatch problem associated with embedded query languages. Users who prefer a declarative external query language may invoke the *Transaction Execution Manager* for handling batch transactions.

The MetaMaster interface provides methods for building new data types. For example, the following statements define schemas for a **Point_2D** and an **Edge_2D**:

```
persistent Schema *point_schema, *edge_schema;

point_schema = meta.new_schema("Point_2D");
meta.add_prop(point_schema, "x-coord", pFloat);
meta.add_prop(point_schema, "y-coord", pFloat);

edge_schema = meta.new_schema("Edge_2D");
```

```
meta.add_prop(edge_schema, "startp", point_schema);
meta.add_prop(edge_schema, "endp", point_schema);
```

The new **Point_2D** schema can be used immediately to create entities. The following example shows a function **newpoint**, which constructs a new **Point_2D** entity and returns its address:

```
persistent Complex *newpoint(float x, float y)
{
  persistent Complex *c = meta.new_complex("Point_2D");
  meta.set_prop(c, 0, x);
  meta.set_prop(c, 1, y);
  return c;
}
```

The newpoint function may now be used with the MetaMaster to create an **Edge_2D** entity and assign it start and end points:

```
persistent Complex *edge;

edge = meta.new_complex("Edge_2D");
meta.set_prop(edge, 0, newpoint(5, 10));
meta.set_prop(edge, 1, newpoint(7, 15));
```

The MetaMaster interface was successfully used to develop a number of applications, including the *Transaction Execution Manager* (a batch processor providing database access from the Unix command-line) and the *Textual User Interface* (a menu-driven system for interactive query sessions). In addition, the MetaMaster was used to integrate pre-existing vision applications (e.g. TRIBORS) with the DEVR system.

5 THE QUERY SYSTEM

5.1 Query Model: Constraint Trees

The query model used in DEVR is that of a recursive constraint tree. Every query has an associated *return type* that corresponds to a particular schema defined in the system, either atomic or complex. The query object consists of a root, internal nodes, and leaves. The root is a template resembling an HRS entity of the return type, having *constraint slots* for each of the properties, parts, and relations defined in the schema. A constraint slot may be empty (implying no constraint), or it may contain a pointer to a *sub-query* of the type associated with that slot. Constraint slots may be either atomic or complex, depending on the type of the corresponding component in the schema. For atomic slots, the sub-query will be a leaf in the constraint tree, consisting of a local boolean expression over the value of that atomic type. The expression may contain an arbitrary number of AND/OR/NOT clauses involving comparison operators (>, <, ==, etc.) and constants.

For complex slots, the sub-query will itself be a constraint tree, whose root is a template of the return type for that slot. For example, consider the query *"find all edges whose* **startp** *is in the upper left quadrant and whose* **endp** *is in the lower left quadrant."*. For a 512×512 image, the query is depicted in figure 3. The root of the constraint tree for this query is of type **Edge_2D**, with two children nodes of type **Point_2D**, occupying the **startp** and **endp** slots.

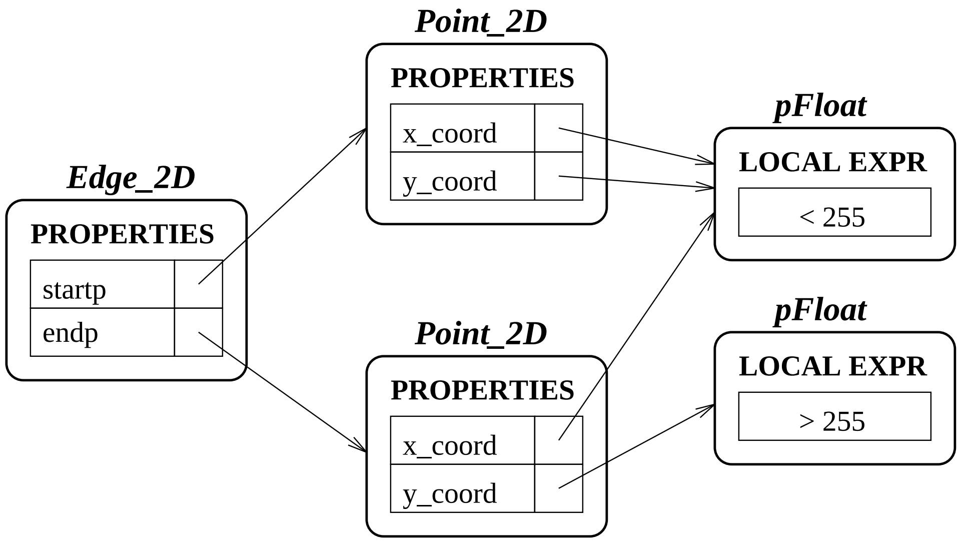


Figure 3: Query on a set of edge entities.

The boolean expression at each leaf only allows local constraints on the atomic value for that component. To represent constraints involving more than one component, each internal node may contain a *regional* constraint expression, whose operands can be any leaves of the sub-tree rooted at that node. Any component name used in a regional expression must be a fully specified path to a leaf, to distinguish it from other possible components. For example, consider the query *"find all horizontal edges"*, which can be modeled as a single node of type **Edge_2D** containing the regional expression **"startp.y-coord == endp.y-coord"**.

Some components of an HRS may be collections of objects (i.e. lists or sets). For example, the **Tribors_Object** HRS defined in 1 contains a **view_classes** part, which is a set of pointers to **View_Class** entities. Queries over these aggregate components are of the same form as a queries over a single entity of the component type, with an additional *cardinality requirement.* This requirement imposes a quantitative threshold that is used to determine if a candidate collection satisfies the query. Cardinality requirements can be of the form *"at least N items"* or *"at least N% of items"*.

To illustrate the expressive power of the DEVR query model, figure 4 shows the following query that was implemented on the TRIBORS application: *"For all objects whose models have at least 20 surfaces, find the view classes whose latitude falls between 45 and 60."*

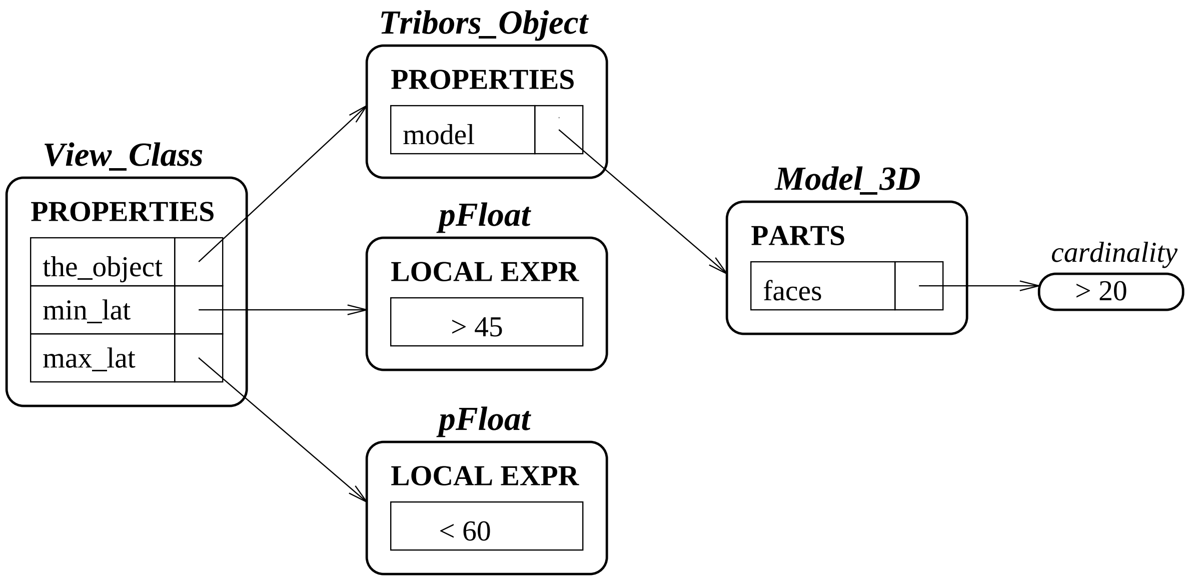


Figure 4: Query on a set of view classes in TRIBORS.

5.2 Implementation of the Query Model

The Query model is implemented as the Query class, a subclass of pBase. Each query constructed by the user is represented as an instance of the Query class, and can be stored in the database. The Query class contains methods for editing query components, parsing boolean expression, and determining whether a given candidate HRS satisfies a set of constraints. As with the Entity class, each Query has a pointer to a particular Schema that identifies its root type. Each Query contains a *template entity* whose component slots do not hold data, but instead hold pointers to other Query objects. To add a constraint on a particular component, a separate Query of that type is constructed and referenced from the component slot of the parent Query.

Users may keep libraries of reusable query objects, which can be built incrementally, tested separately, cloned, and linked together to form more complex queries.

The query object acts as a filter on a *candidate set* of HRS entities of the return type, yielding a *result set* which is the subset of those candidates satisfying every constraint in the query. The system provides a Set class which enables the user to store the results of queries for further processing and browsing. The Set class includes facilities for iterating over its members and maintaining local indexes. In addition, the system provides operators for standard set manipulation, such as union, intersection, and difference.

To test whether a candidate entity satisfies a constraint, DEVR performs a depth-first, recursive traversal of the constraint tree. Each constraint in the tree is applied to the corresponding node of the candidate entity, whose components must satisfy the boolean expression of that constraint. If any regional or local expression evaluates to FALSE, the candidate entity is rejected. If all nodes of the constraint tree are satisfied, a pointer to the candidate entity is inserted into the result set.

Users may construct queries interactively via the menu-driven *Query Specification Interface*, which prompts for boolean constraint expressions and subquery links. In addition, a graphical interface is being developed, in which queries will be visualized as a network of icons that can be manipulated with a pointing device.

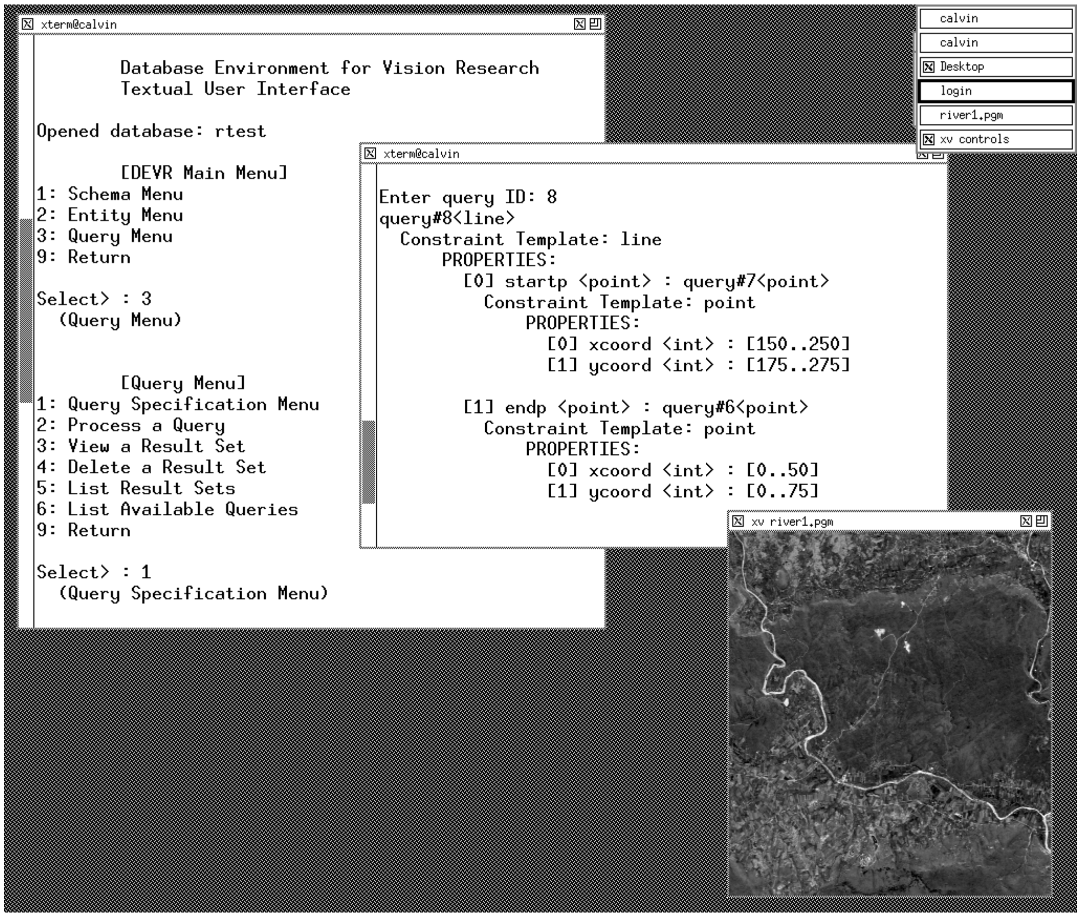
The Query class includes methods to parse boolean constraint expressions. When an expression is added to a query object, the expression is checked for validity, and then a **Filter** object is constructed and attached to the query. During query processing, the Query object passes each candidate value through the Filter object, which efficiently determines whether the constraint is satisfied.

Once queries have been constructed, the *Set Processing Interface* may be used to filter candidate sets. The user chooses a query object and declares an input set, and the system generates a result set. In addition to query processing, the interface allows users to browse sets, create indexes, and employ the standard set operations.

The interactive query interfaces are useful for vision researchers who want to pose high-level queries over their experiment data. However, for vision applications to be integrated with the database, an application programmer's interface is needed. The MetaMaster interface provides methods for constructing and executing queries dynamically from within a C++ application. A wide range of vision applications can be supported using automatic query generation, such as query-by-example.

6 THE TEXTUAL USER INTERFACE

The Application Programmer's Interface is an important tool for integrating DEVR with software applications, but it is not well suited for interactive experimentation or browsing data, since it requires the user to write and compile programs. The Textual User Interface (TUI) is an interactive menu-driven system that allows users to define new data types, create and edit entities, view textual data and images, construct queries, and process sets of data.



```
        Database Environment for Vision Research
        Textual User Interface

Opened database: rtest

        [DEVR Main Menu]
1: Schema Menu
2: Entity Menu
3: Query Menu
9: Return

Select> : 3
  (Query Menu)


        [Query Menu]
1: Query Specification Menu
2: Process a Query
3: View a Result Set
4: Delete a Result Set
5: List Result Sets
6: List Available Queries
9: Return

Select> : 1
  (Query Specification Menu)
```

```
Enter query ID: 8
query#8<line>
  Constraint Template: line
      PROPERTIES:
        [0] startp <point> : query#7<point>
          Constraint Template: point
              PROPERTIES:
                [0] xcoord <int> : [150..250]
                [1] ycoord <int> : [175..275]

        [1] endp <point> : query#6<point>
          Constraint Template: point
              PROPERTIES:
                [0] xcoord <int> : [0..50]
                [1] ycoord <int> : [0..75]
```

Figure 5: The DEVR Textual User Interface.

Existing schemas can be used as templates for constructing new schemas. Properties, parts, and relations may be added or deleted using simple menu commands. Users may also browse existing schema types and view the entities associated with any schema.

Users may construct constraint trees via the Query Specification Menu. After prompting for the root type of the constraint tree, the system displays a template of constraint slots corresponding to the components of that type. The user may constrain each slot with a boolean expression or a pointer to a sub-query. Query objects constructed in this manner are saved in the database and available for later use. Users may construct simple queries, test them separately, then incrementally build more complex constraint trees.

Once a query object has been defined, the user may return to the main Query Menu to process the query on an input set. Input sets are collections of pointers to entities, and can be either the entire extent of a given type, or a specific subset maintained by the user. Sets themselves are persistent objects that can be stored, browsed, and manipulated.

Appendix A demonstrates TUI being used to construct the following queries for the TRIBORS system:

- Find all objects with 2 or more view classes, whose CAD models have at least 40 edges and between 10 and 20 faces.
- Find all view classes of the object named "Cube3Cut" that have at least 15 real images, a minimum latitude of 50, and a maximum longitude of 100.

7 CONCLUSION

The DEVR system has demonstrated that the HRS model is effective in modeling image processing and vision applications. The MetaMaster interface has been used successfully to convert existing vision software to the HRS model (e.g. TRIBORS), and has been tested on experiments involving over one hundred images and their extracted spatial entities. In addition, it has proven to be expedient in developing new applications. The Textual User Interface itself was implemented using the MetaMaster interface.

A graphical interface is being developed[6], which will enhance the system's querying and browsing facilities, and enable the user to design types and construct queries visually.

Future work will involve testing the system on large volumes of data (e.g. satellite images), and analyzing the performance of the query processor. DEVR provides a good environment for research in query optimization because it is a hybrid of relational and object-oriented systems. Queries over the relations component of an HRS could rely on traditional relational query processing techniques[5]. Since the tuple fields are restricted to the members of the part sets of a given entity, joins could be easily optimized and local indexes could be maintained with little overhead. Queries that involve more than just the relations of an entity would require the navigation of paths of pointers across the properties and parts of candidate entities. To optimize the query paths, indexes can be maintained on key components.

DEVR's query processing facilities provide a rich environment for organizing and browsing experiment data. The HRS data model promotes interoperability between applications, and provides a practical framework in which data may be shared among researchers in the computer vision community.

A A Session with TUI

To demonstrate query specification and processing, an annotated transcript of a session with the Textual User Interface is included. Some of the output is edited for brevity.

First, the TUI executable is invoked on a database called "tribtest" which contains data imported from the TRIBORS application.

```
% tui tribtest

        Database Environment for Vision Research
        Textual User Interface

Opened database: tribtest

        [DEVR Main Menu]
1: Schema Menu
2: Entity Menu
3: Query Menu
9: Return
```

A.1 Browsing Schemas

From the Schema menu, the user may see a list of all the data types that are currently defined.

```
        [Schema Menu]
1: View Schema
2: Create New Schema
3: Edit Existing Schema
4: Delete Schema
5: List Schemata
9: Return

Select> : 5
  (List Schemata)

Schemata defined in the database tribtest:

integer     real         fixed_string  var_string
Object_3D   Model_3D     Face_3D       Edge_3D
```

```
Point_3D    Edge_2D      Point_2D      View_Class
Stat_Record Real_Image   Synth_Image   Grey_Scale_Image
Edge_Image  Image_Header Experiment
```

The user may now view the components of any schema by name.

```
Select> : 1
  (View Schema)

Enter type: Object_3D

SCHEMA: Object_3D
NUMBER OF INSTANCES: 2
PROPERTIES:
  [0] name <fixed_string>
  [1] model <Model_3D>
PARTS:
  [0] views <View_Class>
  [1] experiments <Experiment>
NO RELATIONS

Select> : 1
  (View Schema)

Enter type: Model_3D

SCHEMA: Model_3D
NUMBER OF INSTANCES: 2
PROPERTIES:
  [0] the_object <Object_3D>
PARTS:
  [0] faces <Face_3D>
  [1] edges <Edge_3D>
  [2] points <Point_3D>
NO RELATIONS
```

A.2 A Query over Object_3D Entities

The user now wishes to find all **Object_3D** entities with 2 or more view classes, whose CAD model has at least 40 edges and between 10 and 20 faces. Using the Query Specification Menu, the constraint tree may be formulated from the bottom up. First, a query over **Model_3D** entities is constructed, which will later be used to constrain the *model* property slot of a **Object_3D** query.

```
        [Query Specification Menu]
```

```
1: View Query
2: Define New Query
3: Edit Query
4: Delete Query
5: List Available Queries
9: Return

Select> : 2
  (Define New Query)

Enter type: Model_3D

Created query#1<Model_3D>.
Edit this query [y/n]: y

The query currently selected for editing is:
query#1<Model_3D>
    Constraints on Properties:
      [0] the_object <Object_3D> : NULL
    Constraints on Parts:
      [0] faces <Face_3D> : unconstrained.
      [1] edges <Edge_3D> : unconstrained.
      [2] points <Point_3D> : unconstrained.

        [Edit Query]
1: Select New Query to Edit
2: View Selected Query
3: Edit Property Constraint
4: Edit Parts Constraint
5: Edit Relation Constraint
6: Edit Constraint Expression
9: Return

Select> : 4
  (Edit Parts Constraint)

Enter slot number of part to constrain: 0
Enter constraint on cardinality of faces: (>= 10) AND (<= 20)
Slot 0 has been updated.

Enter slot number of part to constrain: 1
Enter constraint on cardinality of edges: >= 40
Slot 1 has been updated.
```

The **Model_3D** query has been stored in the database, and assigned query ID #1 for future reference. Next, the user creates an **Object_3D** query, adding constraints to the *model* property and the *view_classes* part slot.

```
Select> : 2
  (Define New Query)


Enter type: Object_3D
Created query#2<Object_3D>.

Edit this query [y/n]: y

The query currently selected for editing is:

query#2<Object_3D>
  Constraints on Properties:
    [0] name <fixed_string> : NULL
    [1] model <Model_3D> : NULL
  Constraints on Parts:
    [0] views <View_Class> : unconstrained.
    [1] experiments <Experiment> : unconstrained.

        [Edit Query]
1: Select New Query to Edit
2: View Selected Query
3: Edit Property Constraint
4: Edit Parts Constraint
5: Edit Relation Constraint
6: Edit Constraint Expression
9: Return

Select> : 3
  (Edit Property Constraint)

Enter slot number of property to constrain: 1
Enter query ID of constraint for model<Model_3D>: 1
Slot 1 has been updated.

Select> : 4
  (Edit Parts Constraint)

Enter slot number of part to constrain: 0
Enter constraint on cardinality of views: >= 2
```

```
Slot 0 has been updated.

Select> : 2
  (View Selected Query)

The currently selected query:

query#2<Object_3D>

Constraints on Properties:
  [0] name <fixed_string> : NULL
  [1] model <Model_3D> : query#1<Model_3D>
    Constraints on Properties:
      [0] the_object <Object_3D> : NULL
    Constraints on Parts:
      [0] faces <Face_3D> : (((>= 10)) AND ((<= 20)))
      [1] edges <Edge_3D> : (((>= 40)))
      [2] points <Point_3D> : unconstrained.
Constraints on Parts:
  [0] views <View_Class> : (((>= 2)))
  [1] experiments <Experiment> : unconstrained.
```

Now the completed query is used as a filter over all entities of type **Object_3D**, creating an output set of pointers to those entities that satisfy the constraints.

```
        [Query Menu]
1: Query Specification Menu
2: Process a Query
3: View a Set
4: Delete a Set
5: List Sets
6: List Available Queries
9: Return

Select> : 2
  (Process a Query)


Enter query ID: 2
Query is of type query#2<Object_3D>.

Query over all entities of this type [y/n]: y
```

```
input set: set of 2 items.
  { Object_3D#1, Object_3D#2 }

** testing an item: Object_3D#1 **
PROPERTIES:
  [0] name <fixed_string> : Fork
  [1] model <Model_3D> : Model_3D#1
    PROPERTIES:
      [0] the_object <Object_3D> : Object_3D#1
    PARTS:
      [0] faces <Face_3D> : set of 14 items.
      [1] edges <Edge_3D> : set of 36 items.
      [2] points <Point_3D> : set of 24 items.
PARTS:
  [0] views <View_Class> : set of 2 items.
  [1] experiments <Experiment> : set of 0 items.

** Object_3D#1 did not satisfy **

** testing an item: Object_3D#2 **
PROPERTIES:
  [0] name <fixed_string> : Cube3Cut
  [1] model <Model_3D> : Model_3D#2
    PROPERTIES:
      [0] the_object <Object_3D> : Object_3D#2
    PARTS:
      [0] faces <Face_3D> : set of 18 items.
      [1] edges <Edge_3D> : set of 48 items.
      [2] points <Point_3D> : set of 32 items.
PARTS:
  [0] views <View_Class> : set of 3 items.
  [1] experiments <Experiment> : set of 0 items.

** Object_3D#2 satisfies **

set#183 created with 1 elements.
{ Object_3D#2 }
```

A.3 A Query over View Classes

Only the **Object_3D** named "Cube3Cut" satisfied the constraints of query #1. Cube3Cut's **Model_3D** contains 3 distinct view classes. The user now wishes to find out which of those view classes contain at least 15 real images, have a minimum latitude greater than 50, and a maximum longitude less than 100.

Once again, the query is constructed bottom-up. First, the user constructs the query "Find all 3D objects named Cube3Cut".

```
  (Define New Query)

Enter type: Object_3D
Created query#3<Object_3D>.

Edit this query [y/n]: y

The query currently selected for editing is:

query#3<Object_3D>
  Constraints on Properties:
    [0] name <fixed_string> : NULL
    [1] model <Model_3D> : NULL
  Constraints on Parts:
    [0] views <View_Class> : unconstrained.
    [1] experiments <Experiment> : unconstrained.

Select> : 3
  (Edit Property Constraint)

Enter slot number of property to constrain: 0
Enter constraint for name<fixed_string>: == Cube3Cut
Slot 0 has been updated.
```

Next, the root level query is constructed, of type **View_Class**. The property slot *the_object* is constrained with the previously constructed query. Also, the *min_lat* and *max_long* properties are assigned integer constraints. Finally, the cardinality of the *real_images* part set is constrained to be >= 15.

```
  (Define New Query)

Enter type: View_Class
Created query#4<View_Class>.

Edit this query [y/n]: y

Select> : 3
  (Edit Property Constraint)

Enter slot number of property to constrain: 0
Enter query ID of constraint for the_object<Object_3D>: 3
Slot 0 has been updated.
```

```
Enter slot number of property to constrain: 1
Enter constraint for min_lat<integer>: > 50
Slot 1 has been updated.

Enter slot number of property to constrain: 4
Enter constraint for max_long<integer>: < 100
Slot 4 has been updated.

Select> : 4
  (Edit Parts Constraint)

Enter slot number of part to constrain: 1
Enter constraint on cardinality of real_images: >= 15
Slot 1 has been updated.

Select> : 2
  (View Selected Query)

The currently selected query:

query#4<View_Class>
  Constraints on Properties:
    [0] the_object <Object_3D> : query#3<Object_3D>
        Constraints on Properties:
          [0] name <fixed_string> : (((== Cube3Cut)))
    [1] min_lat <integer> : (((> 50)))
    [2] max_lat <integer> : NULL
    [3] min_long <integer> : NULL
    [4] max_long <integer> : (((< 100)))
  Constraints on Parts:
    [0] model_edges <Edge_3D> : unconstrained.
    [1] real_images <Real_Image> : (((>= 15)))
    [2] synth_images <Synth_Image> : unconstrained.
```

Now the query is used to filter all view classes of the system. The first two view classes fail because they belong to the Fork object instead of the Cube3Cut object:

```
        [Query Menu]
1: Query Specification Menu
2: Process a Query
3: View a Set
```

```
4: Delete a Set
5: List Sets
6: List Available Queries
9: Return

Select> : 2
  (Process a Query)


Enter query ID: 4
Query is of type query#4<View_Class>.

Query over all entities of this type [y/n]: y

input set: set of 5 items.
  { View_Class#1, View_Class#2, View_Class#3, View_Class#4, View_Class#5 }

** testing an item: View_Class#1 **
  PROPERTIES:
    [0] the_object <Object_3D> : Object_3D#1
        PROPERTIES:
          [0] name <fixed_string> : Fork

** did not satisfy. **

** testing an item: View_Class#2 **
  PROPERTIES:
    [0] the_object <Object_3D> : Object_3D#1
        PROPERTIES:
          [0] name <fixed_string> : Fork

** did not satisfy. **
```

Of the remaining three view classes, two satisfy the entire constraint tree, while one is rejected because the *max_long* property exceeds 100.

```
** testing an item: View_Class#3 **
  PROPERTIES:
    [0] the_object <Object_3D> : Object_3D#2
        PROPERTIES:
          [0] name <fixed_string> : Cube3Cut
    [1] min_lat <integer> : 62
    [4] max_long <integer> : 55
  PARTS:
    [1] real_images <Real_Image> : set of 20 items.

** satisfies! **

** testing an item: View_Class#4 **
  PROPERTIES:
    [0] the_object <Object_3D> : Object_3D#2
        PROPERTIES:
          [0] name <fixed_string> : Cube3Cut
    [1] min_lat <integer> : 62
    [4] max_long <integer> : 180
  PARTS:
    [1] real_images <Real_Image> : set of 20 items.

** did not satisfy. **

** testing an item: View_Class#5 **
  PROPERTIES:
    [0] the_object <Object_3D> : Object_3D#2
        PROPERTIES:
          [0] name <fixed_string> : Cube3Cut
    [1] min_lat <integer> : 62
    [4] max_long <integer> : 30

** satisfies! **

set#189 created with 2 elements.
```

References

[1] R. Agrawal and N. Gehani. Ode: Object database & environment. *SIGMOD*, 1989.

[2] F. Bancilhon and P. Buneman. *Advances in Database Programing Languages.* ACM Press, New York, 1990.

[3] A. Biliris, N. Gehani, et al. *Ode 3.0.3 User Manual.* AT&T Bell Laboratories, Murray Hill, New Jersey.

[4] R. Jakobovits, L. Shapiro, and S. Tanimoto. Implementing multi-level queries in a database environment for vision research. In *IS&T/SPIE Symposium on Electronic Imaging: Science & Technology*, February 1995.

[5] H. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, New York, 1991.

[6] L. Lewis, L. Shapiro, and S. Tanimoto. Flexible data organization with visualization support for a visual database system. In *IS&T/SPIE Symposium on Electronic Imaging: Science & Technology*, February 1995.

[7] K. Pulli. Tribors: A triplet-based object recognition system. Technical Report 95-01-01, Deptartment of Computer Science and Engineering, University of Washington, Seattle, WA, January 1995.

[8] L. Shapiro, S. Tanimoto, J. Brinkley, J. Ahrens, R. Jakobovits, and L. Lewis. A visual database system for data and experiment management in model-based computer vision. In *Proceedings of the Second CAD-Based Vision Workshop*, pages 64–72, February 1994.

[9] L. G. Shapiro and R. M. Haralick. A spatial data structure. In *Geo-Processing 1*, pages 313–337, 1980.

[10] B. Stroustrup. *The C++ Programming Language, 2nd Ed.* AT&T Bell Labs, Inc., Murray Hill, NJ, 1993.