Delaunay: a Database Visualization System*

Isabel F. Cruz†
Database Visualization Research Group
Tufts University
ifc@cs.brown.edu

Michael Averbuch
Database Visualization Research Group
Tufts University
averbukh@cs.tufts.edu

Wendy T. Lucas
Database Visualization Research Group
Tufts University
wlucas@cs.tufts.edu

Melissa Radzyminski
Database Visualization Research Group
Tufts University
mradzymi@cs.tufts.edu

Kirby Zhang
Database Visualization Research Group
Tufts University
kzhan@cs.tufts.edu

Abstract

Visual query systems have traditionally supported a set of pre-defined visual displays. We describe the *Delaunay* system, which supports visualizations of object-oriented databases specified by the user with a visual constraint-based query language. The highlights of our approach are the expressiveness of the visual query language, the efficiency of the query engine, and the overall flexibility and extensibility of the framework. The user interface is implemented using Java and is available on the WWW.

1 Introduction

The *Delaunay*[1] system is an interactive system for the declarative querying and display of object-oriented databases. Delaunay is based on the visual database query language *DOODLE* (*D*raw an *O*bject-*O*riented *D*atabase Languag*E*) [2]. Users arrange graphical objects and graphical constraints to form a "picture" that specifies how to visualize objects belonging to a class. Taken together, the picture and class form a *user-defined term*, or *U-term*. The user interface is syntax-directed, therefore the U-terms are syntactically correct.

Figure 1 shows objects stored in an object-oriented database, such as O_2 [1], described using the F-logic syntax [8]. Figure 2 represents a DOODLE program for visualizing the database objects of Figure 1. The program consists of six U-terms, one for each class, and specifies that objects of class *module* are to be displayed as red boxes, objects of class *procedure* as purple circles, and objects of class *function* as blue circles. In addition, objects of class *calls* are to be displayed as arrows that go from the "caller" object to the "called" object (similarly, objects of class *contains* will be displayed as green arrows).

$oid_1 : contains\,[outer \rightarrow oid_5 : module;\ inner \rightarrow oid_6 : procedure\,]$
$oid_2 : calls\,[caller \rightarrow oid_7 : procedure;\ called \rightarrow oid_8 : procedure\,]$
$oid_3 : calls\,[caller \rightarrow oid_8 : procedure;\ called \rightarrow oid_9 : function\,]$
$oid_4 : set\,[members \rightarrow \{oid_1 : contains; oid_2, oid_3 : calls\}\,]$

Figure 1: The notation `<object identity>:<class name>` associates an object to the class to which it belongs. Class attributes represent mappings; e.g., *outer* maps oid_1 to the object oid_5 of class *module*. The attribute *members* is set-valued.

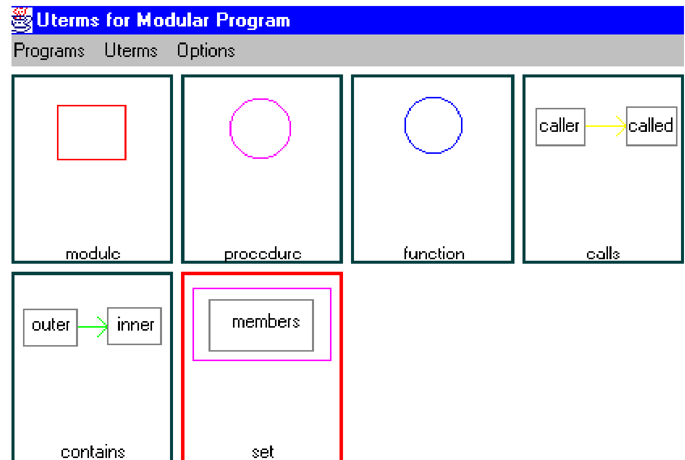

Figure 2: Simplified DOODLE program with six U-terms.

In the U-term for class *calls*, whose attributes refer to objects of class *procedure* or *function* between which the arrows will be drawn, the user does not have to draw the shape of the referred objects, but draws instead the key symbol *refbox* (for *reference box*), which shows the respective attribute name. Each refbox refers to a visual display defined elsewhere within the same program (or, in the case of a *labeled refbox*, in another program), which depends on the actual classes being represented. For example, the refbox *called* refers either to the display defined for the objects

*Research supported in part by the National Science Foundation under CAREER Award IRI–9625105, and by a CRA/NSF grant. WWW URL: http://www.cs.tufts.edu/~dbvis

†Work by this author performed in part at Brown University.

[1] Named after the cubist painter Sonia Delaunay (1885-1979) whose compositions include bright colored geometric figures.

of class *procedure* or of class *function*; this capability is just one of the forms of genericity that DOODLE supports [3]. Finally, the display of *set* is specified to be the aggregate of the displays of all its members.

In this paper we describe a prototype of the Delaunay system and the techniques that are used in our approach to tailorable visual database querying. The main characteristics of Delaunay are summarized here:

User interface The interface gives an overview of each DOODLE program (see for example Figure 2) and attempts to minimize the amount of information on the screen. It has access to the databases to prompt the user with their names, class names, and attribute names as appropriate, and provides access to other existing DOODLE programs through a simple pull-down menu. The input pad implements standard editing functions.The interface is implemented in Java and is available on the WWW using any Java-enabled browser.

System modularity Delaunay was designed for extensibility: we defined completely the languages between every two communicating software modules, so as to allow for future replacement or addition of other modules. A client-server architecture, based on standard Java streams and Internet sockets, enables the use of existing constraint resolution tools without reimplementing them in Java.

Expressiveness and efficiency The expressiveness of DOODLE stems from its capability to express recursion [2] and from the variety of U-terms that can be assembled using constraints [4]. Our constraints can express a wide variety of displays and can also be solved efficiently (in linear time) [5, 6].

The rest of the paper is organized as follows. In Section 2 we describe the two visual languages that are part of Delaunay. Sections 3 and 4 overview the graphical user interface and software architecture of the the current prototype, respectively. Examples of visualizations produced with Delaunay are given in Section 5. Finally, Section 6 discusses directions for future research.

2 Visual Languages in Delaunay

2.1 U-term Language

U-terms are elements of the U-term visual language. The main *graphical symbols* from which U-terms are composed fall into one of three categories: *prototypical symbols*, *constraint symbols*, and *key symbols*. Prototypical symbols include *box, text, circle, straight line, arrow*, and *double arrow*.

All prototypical symbols have pre-defined *landmarks* that serve as reference points on their boundaries. They include *center* (for all objects), *midnorth, mideast, midsouth*, and *midwest* (for polygons and circles), *head* and *tail* for (lines and arrows), and *arrowhead* and *arrowtail* for arrows (Figure 4 shows some landmarks). The user can also define reference points called *anchorpoints* anywhere on the border of a graphical symbol.

Spatial relationships between graphical symbols are specified using *constraints*. Two types of constraints are supported: *length* and *overlap*. A length constraint denotes the distance between two landmarks or anchorpoints. Its direction can be horizontal, vertical, or Euclidean. We use a class of basic constraints over a set of variables, where linear arithmetic expressions are combined with *min/max* operators (e.g., to set the height of a bounding box to the maximum height of the objects it contains). The variables may refer to other constraints in the same U-term, or to the value of a numerical database attribute (so that we can set a length to be proportional to that attribute value). An overlap constraint is used to specify when one graphical symbol is to be displayed on top of another.

Key symbols specify the interpretation of prototypical symbols and constraints. These symbols include: *defbox* (for *definition box*), *refbox*, and *grouping box*.

A defbox can contain any graphical symbol except for another defbox. Anchorpoints on its border can be associated, through the use of length constraints, with anchorpoints or landmarks on the objects within the defbox. The depiction of a database class, as defined by a defbox, can be referenced within another U-term by a refbox. A refbox can also have anchorpoints that coincide with those on the corresponding defbox. A grouping box can contain any symbol except for a defbox.

Other symbols like the *origin symbol*, which is represented by the coordinates (0,0) and by a a small blue box, are used to specify the coordinates of the prototypical symbols: a length constraint can be drawn between a landmark (or anchorpoint) and the origin.

2.2 DOODLE Programs

Each U-term in the DOODLE program of Figure 2 corresponds to a DOODLE rule of the form `<U-term>` $\leftarrow$ `<database-class>`, which is to be read right to left as "map objects of the database class to a set of visual objects as specified by the U-term".

In DOODLE programs, the order of the rules (and of the terms) is irrelevant. The meaning of a DOODLE program is defined formally as follows: each rule is translated into an F-logic rule [8], and the F-logic minimal model semantics is used [2, 3]. The execution of a DOODLE program creates a view of the database where visual objects are added to the query domain.

3 Interacting with Delaunay

The user starts by selecting a working database from a list of available databases, and then selects a class from that database. Next, the four main components of the Delaunay interface appear on the screen. These components are the tool window (Figure 3), the input pad, the U-term window, and the textual U-term window (Figure 4).

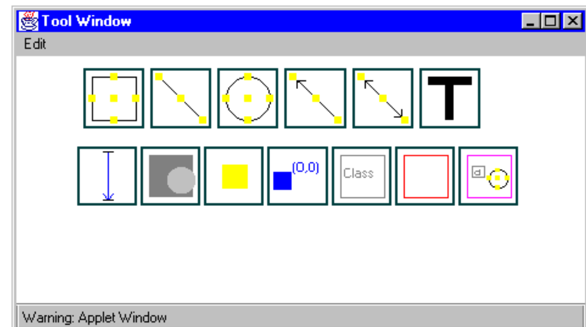

Figure 3: Tool window.

The tool window contains buttons for the graphical symbols. By clicking on a button and on a position on the input pad, the user places the corresponding symbol on that position. By double-clicking on any prototypical symbol, its

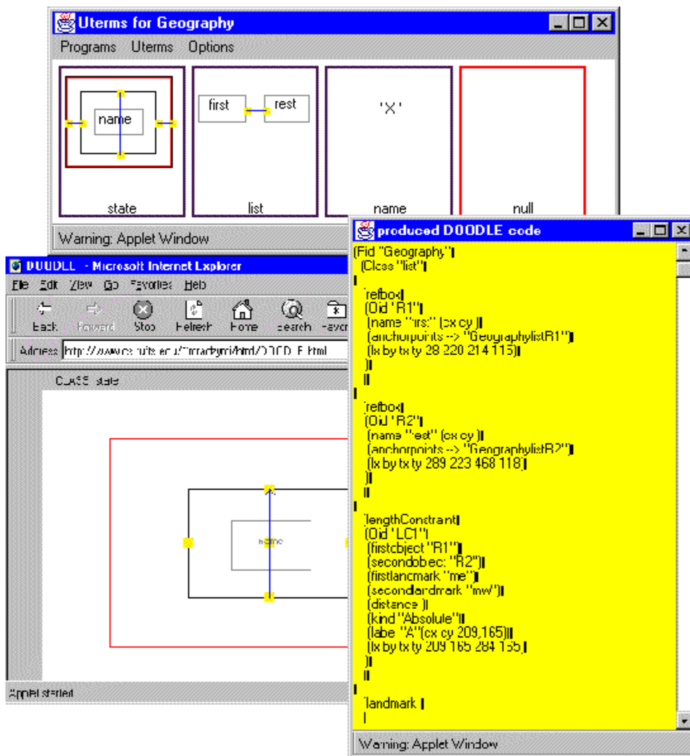

Figure 4: Input pad, U-term window, textual U-term window.

graphical attributes can be specified (e.g., color, line thickness). It is possible to assign *color* automatically: the user is prompted for the attribute on whose value the color depends. As a result, all objects with the same attribute value will be assigned the same color.

Landmarks appear automatically on all visual objects. A user can also specify anchorpoints by clicking on their button in the tool window and then positioning the resulting yellow box on the border of an object. The user is then prompted for an anchorpoint label.

The tool window also contains buttons for specifying constraints. After clicking on the length constraint button, the user selects the landmarks/anchorpoints for the start and end of the constraint. A window then pops up in which the user types in the equation associated with the constraint or, in the case of a proportional constraint, the name of a numerical attribute. The type of the length constraint (e.g., horizontal) is also selected here.

In the case of lines and arrows, an Euclidean constraint of length zero is assumed whenever one of the endpoints of the line or arrow is in direct contact with a landmark or anchorpoint on any other object. Attributes for this and all other constraints can be modified by double-clicking on the constraint itself.

To specify an overlap constraint, after selecting the corresponding button from the tool window, the user first clicks on the object to appear on the bottom, which becomes shaded dark gray. The user then clicks on the object to appear on the top, which becomes shaded light gray.

The last three buttons in the tool window are for adding key symbols to U-terms. After clicking on the refbox button, the user is prompted for an attribute of the class of the current U-term. The depiction of the class associated with that attribute will replace the refbox on the output pad. All of the objects drawn by the user on the input pad are assumed to be within a defbox, unless the user resizes it. The last button is used to draw a grouping box.

A miniature rendition of the current U-term appears in the U-term window. Each time the user adds a prototypical or key symbol to the input pad, the current U-term display is updated. In the case of constraints, the user can select an operation mode in which they are displayed or one in which they are not. This window also has a pull-down menu with an option for creating a new U-term.

A textual description the current U-term is also maintained by the interface as shown in Figure 4. The format is a list specifying the program in which the U-term appears, the class being depicted, and all the graphical objects [7].

4 Architecture

Delaunay uses a client-server architecture consisting of five distinct modules, as shown in Figure 5. In the current prototype, the User Interface runs on the client side and is implemented in Java, while the other modules reside on the server side.

The Input Pad is described in the preceding section. We now describe the Parser, the Database Front-End, the Query Engine, the Output Pad, and the data streams that connect them.

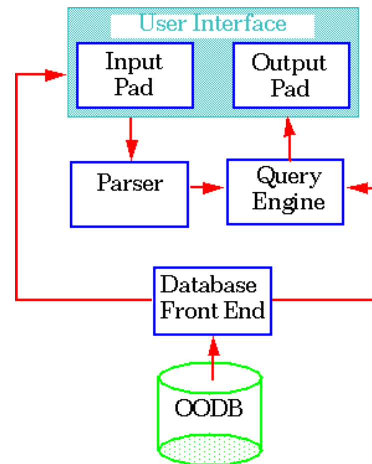

Figure 5: Block diagram of the Delaunay prototype implementation. The arrows represent the flow of data between modules.

The Parser The output stream from the User Interface is parsed by this module into a data structure that stores all symbols and associates them with the program and U-terms from which they originate. The data structure is then output to the Query Engine module.

The Database Front-End This modules retrieves the database objects, the bindings between object identities and classes (e.g., **bind**(oid_1,**binTree**)), and the class hierarchy (e.g., **subclass(binTree,tree)**). This information is accessed by the User Interface and by the Query Engine.

The Query Engine The Query Engine performs two functions: (i) it instantiates the U-terms with objects from the database, and (ii) it computes the coordinates of the instantiated prototypical objects by evaluating the system of instantiated constraints.

Instantiation To instantiate the prototypical objects and constraints of the U-terms, this module uses the data structure constructed by the Parser and the database information provided by the Database Front-End. The database objects are examined in two passes. In the first pass, we identify the database objects to be visualized, associate an U-term with each of them, and create a search structure for them. These objects are such that their class, or a superclass, has a U-term defined. We associate with each such object the U-term of its class or of the most specialized superclass. Also, we instantiate the prototypical graphical objects and variables for the coordinates of the landmarks and anchorpoints defined in the U-term. In the second pass, we instantiate the constraints over the previously instantiated variables. The constraints relate the coordinates of the landmarks and anchorpoints of the current object to those of other objects referred to via refboxes.

Evaluation The constraint solving strategy is as follows. We build a constraint graph where each vertex represents a constraint variable (i.e., a coordinate of a landmark or anchorpoint), and each edge represents a dependency between two variables established by a constraint. The edges are directed for a max or min constraint, and are undirected otherwise. If the edges of the constraint graph can be oriented such that the resulting digraph has no directed cycles, then we can solve the system of constraints in linear time [5, 6] by evaluating the variables in topological ordering. Otherwise, we resort to SkyBlue [9], a general constraint solver.

The Output Pad The input to the Output Pad contains the prototypical symbols (along with their visual attributes and their absolute coordinates) that correspond to the objects in the database and their drawing sequence. The Output Pad translates this data into its graphical representation, which is presented to the user in a Java applet. This applet shows the textual list of the objects being displayed and its graphical portion supports fish-eye views.

5 Examples

The DOODLE program of Figure 4 specifies a bar chart visualization of a geographical database storing population data for US states. The database has objects of class *state*, with attributes *name* and *population*, and of class *list*, with attributes *first* and *rest*. The class of the attribute values of *name* is *stateName*, of *population* is *real*, and of *first* and *rest* is *list*. The U-term for class *state* specifies that each

Figure 6: Visualization of the geographical database.

state is visualized by a rectangular bar bearing its name, as indicated by the refbox within it. The vertical constraint indicates that the height of the bar is proportional to the state's population. The color of the bar is assigned automatically. The U-term for class *list* specifies that the representation of the object associated with the *first* attribute should be placed immediately next to the representation of the object associated with the *rest* attribute. The U-term for class *stateName* specifies a textual representation, as depicted by the symbol 'X'. Finally, the representation for the class *null* is empty.

Figure 6 shows a bar chart visualization of the geographical database. Figure 7 shows another example, where the specified visualization is a binary tree displayed in an upward fashion. The second U-term specifies the display of the leaves.

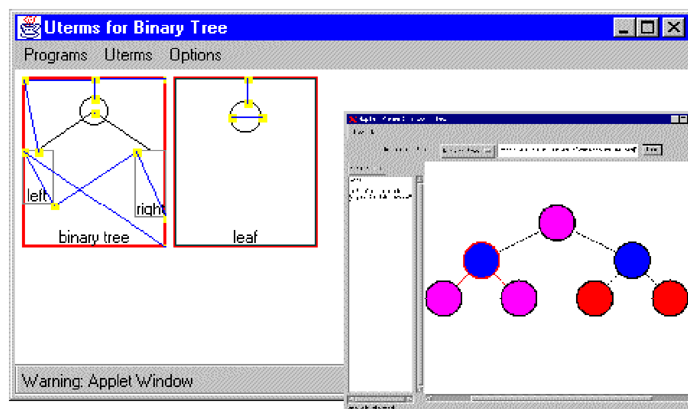

Figure 7: DOODLE program that specifies the binary tree representation and the output pad with the visualization.

6 Future Work

Future work will focus on extending the Delaunay prototype to support more complex DOODLE programs [3], polar coordinates, and macro constraints [4]. Other extensions include supporting the manipulation of DOODLE programs through inheritance and composition [2, 3].

References

[1] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System (The story of* O_2*)*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[2] I. F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 71–80, 1992.

[3] I. F. Cruz. User-defined Visual Query Languages. In *IEEE Symposium on Visual Languages (VL '94)*, pages 224–231, 1994.

[4] I. F. Cruz. Expressing Constraints for Data Display Specification: A Visual Approach. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, pages 443–468. The MIT Press, 1995.

[5] I. F. Cruz and A. Garg. Drawing Graphs by Example Efficiently: Trees and Planar Acyclic Digraphs. In *Graph Drawing '94*, number 894 in Lecture Notes in Computer Science, pages 404–415. Springer Verlag, 1995.

[6] I. F. Cruz, A. Garg, and R. Tamassia. Drawing Graphs Efficiently with Visual Constraints. Technical report, Department of Computer Science, Brown University, 1997.

[7] I. F. Cruz and W. T. Lucas. An Interchange Format for Declaratively Specified Visualizations with Constraints, 1996. Database Visualization Research Group Technical Report available at `http://www.cs.tufts.edu/~wlucas/vis3.ps`.

[8] M. Kifer, G. Lausen, and J. Wu. Logic Foundations of Object-Oriented and Frame-Based Languages. *J. ACM*, 42(4):741–843, July 1995.

[9] M. Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Computer Science Department, University of Washington, February 1992.