Kaleidoquery: A Visual Query Language for Object Databases

Norman Murray, Norman Paton and Carole Goble

Department of Computer Science
University of Manchester
Oxford Road, Manchester, M13 9PL, UK
E-mail:⟨murrayn, norm, carole⟩@cs.man.ac.uk

ABSTRACT

In this paper we describe Kaleidoquery, a visual query language for object databases with the same expressive power as OQL. We will describe the design philosophy behind the filter flow nature of Kaleidoquery and present each of the language's constructs, giving examples and relating them to OQL. The Kaleidoquery language is described independent of any implementation details, but a brief description of a 3D interface currently under construction for Kaleidoquery is presented. The queries in this implementation of the language are translated into OQL and then passed to the object database O_2 for evaluation.

KEYWORDS: Visual query language, OQL, object databases, three-dimensional interface.

INTRODUCTION

The lack of a generally accepted and widely supported query language has probably had a significant effect in slowing the uptake of early commercial object-oriented databases. However, the emergence of the Object Query Language (OQL) which is being standardised by the Object Database Management Group (ODMG) and being supported by a growing number of object database vendors promises to address this limitation. However, textual query languages exhibit several problems to the database user including the need to know the databases classes, attributes and relationship structure before writing a query, and also the problems of semantic and syntactic errors.

Visual query languages attempt to bridge the gap of usability for users, and this paper presents a new visual query language for object databases that depicts the query as a filter flow. The visual queries produced can be translated into the ODMG standard OQL. In this way the language can be utilised in any ODMG compliant database which supports OQL. The version of Kaleidoquery presented within this paper is for OQL from version 2.0 of the ODMG standard [7].

This paper is organised as follows. In the next section we discuss the design philosophy of Kaleidoquery, and specify the language through examples of the constructs and the corresponding OQL queries. This is followed by a brief introduction to one of the query environments that has been implemented for the language, concluding with a discussion of related work and a summary.

LANGUAGE DESCRIPTION

Graph based query languages tend to use a representation of the database schema as the starting point for querying the database. The user selects parts of the schema and constructs a query from these by combining them with query operators. A problem with graph based queries is that they depict the database schema using entity-relationship diagrams or some other notation that, while readable for a database designer, exploits an abstract symbolic notation that may at first prove cumbersome to the casual user of a database and require some learning. We wanted to depict the query as the user would visualise it in operation. When performing a query we start with a mass of information. To this we apply filters or constraints. Our mass of information passes through these filters letting through only information that we are interested in. We wanted to depict this flow and refinement of information through the query which is not clearly seen in graph based queries. An early example of this form of filter flow was proposed by Shneiderman [19]. Shneiderman used the metaphor of water flowing through a series of pipes and filters, where each filter would let through only the appropriate items and the layout of the pipes indicating the relationships of *and* and *or*.

This filter flow model may prove to be easier to understand over a graph based data model as the information passes along a route defined by the model as the query filters the information. This is in contrast with other graph-based languages which obtain a obtain a view of the database schema and apply queries to sections of the schema, where the ordering of the query, if there is any, may be hard to distinguish.

Our queries follow this filter flow model with the input to the queries composed of class instances (extents) entering the query and flowing through, being filtered by the constraints placed on the attributes of the class. The output of the query can then be examined, or it can flow into other queries to be further refined. A query is therefore composed from the extents of the database that form the primary inputs to the query, the classes of the database schema and additional constructs that together form the query visualisation.

In designing the language the premise that has been adopted is that the textual equivalent of the language is to be retained, but with this an iconic or abstract representation may be attached, as studies have shown that icons with a textual description give better comprehension than textual or pictorial icons [2, 10, 13]. As the participants become familiar with the interface they will associate the icon/abstract representation with the text and in this way will not have to read the full text to comprehend the query.

We shall now examine the visualisation of the classes and extents before moving on to the visualisation of the query constructs.

The Database Schema and Extents

For the examples in this paper we will use the classes and extents shown in Figure 1. This consists of the *Person* class that has the properties name, age, children, parent, employer and salary, and the *Company* class having the properties name, location and employees. The only visible information on the exterior of the class is the iconic representation of the class and its name. All other information pertaining to the class is accessible when the participant selects the class and then chooses to view its attributes. The ODL (Object Definition Language) for the two classes is:

```
class Person
(   extent People)
{
    attribute string name;
    attribute int age;
    relationship set<Person> children
        inverse Person::parents;
    relationship list<Person> parents
        inverse Person::children;
    relationship Company employer
        inverse Company::employees;
    attribute real salary;
};

class Company
(   extent Companies)
{
    attribute string name;
    attribute string location;
    relationship set<Person> employees
        inverse Person::employer;
};
```

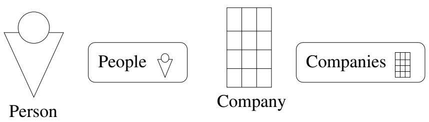

Figure 1: Example Schema

The database extents can consist of all the instances, or a subset of the instances of a particular class. The visualisation of the extents, here shown to the right of their respective classes in Figure 1, consists of the extent name and the icon associated with the class of the extent, surrounded by an oval box. In this database there are two extents, one for each class in the schema. The extent allows the participant access to the group of instances, which may be viewed by selecting the extent. As queries are constructed on the schema, the extent group chosen as input to the query will *flow* through the query and be filtered according to the constraints placed on it. Results from a query or a sub query have the new extents, representing the results, placed after the class constraints, and these extents can be used to provide access to the results or as input to other sections of the query.

Results Selection

The simplest form of query allows us to selectively view certain attributes of interest of an instance rather than obtaining the complete instance details.

With Kaleidoquery the initial type of the data will be defined by the extent that flows into the query. It is this type that will be passed along the filter flow pipes, visualised as upward pointing arrows, ↑, to the results. The type of the filter flow is not altered by the query constraints – the only time the filter flow can be altered is when it flows into a results box. Along the filter flow it is the instances of the type that are filtered by constraints placed along the filter flow. We shall see how constraints are visualised later.

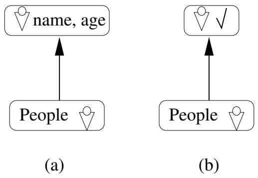

Figure 2: Selecting attributes

In Figure 2 we see two versions of the visualisation of the filter flow type where certain attributes of the class Person are selected for output. In Figure 2 (a) the class has the attributes that have been selected for output in the results placed to the right of the class icon. The instances from the *People* extent flow towards the new results extent but only the attributes shown selected in the results box are allowed to enter the results, here consisting of the *name* and *age* attributes of the Person class. This query is the same as the *select* statement of OQL, the OQL for this query being:

```
select tuple(name:p.name, age:p.age)
from p in People
```

If many attributes of the class have been selected to appear in the results the user may choose not to view them as part of the query but show that some of the attributes have been selected by having a tick placed next to the class, as shown in Figure 2 (b). The attributes that have been selected for inclusion can be examined by selecting the results and choosing to see the result's attributes. If a query is placed on the class and none of the attributes have been selected for inclusion then the default is to view all the attributes of the class, as can be seen in our next query in Figure 3. Towards the end of this section we shall see how we can structure, order and group the query results.

Simple Query Constructs

The query shown in Figure 3 includes a simple constraint that restricts the results to include only persons who are aged less than 20. In this case no attributes in the person class have been selected for viewing so by default the results consist of the group of people satisfying the query. Any of the operators =,>,>=,<,<= and *like* can be used in these queries. The OQL for this query is:

```
select p
from p in People
where p.age < 20
```

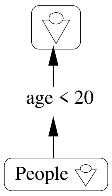

Figure 3: A simple query

We shall now look at how simple queries can be combined using boolean expressions.

And,Or and Not

The filter flow model described by Shneiderman [19] was chosen to visualise the *and* and *or* relations of the query as it has been previously shown that boolean logic has proven difficult for large sections of the population. Errors range from improper use of *and* and *or* between the English and the boolean logic interpretations of their operation, to the difficulties of using parenthesis to express the rules of precedence on the boolean operators, especially when building complex queries, [3, 11, 15]. Preliminary results have shown that the filter flow representation has proven to be more favourable than a text only SQL interface [22]. We shall now examine how *and* and *or* can be specified in a query. Figure 4 shows four different query visualisations. Figure 4 (a) shows the *and*-ed visualisation of the OQL query:

```
select p
from p in People
where p.age < 20 and p.name = "Smith"
```

The instances from the input extent flow through the query and are filtered through each constraint in turn. Figure 4 (b) shows how an *or* is visualised by the extent instances flowing into each part of the *or* query and combining at the top of the

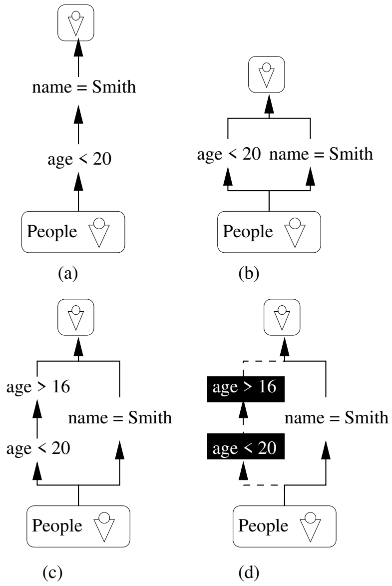

Figure 4: Visualising *and*,*or* and *not*

query to produce one group of results. The OQL of the query is the same as the above example except that the constraints have been *or*-ed. In Figure 4 (c) we see a more complex query using both *and* and *or* expressions. The OQL for this is as follows:

```
select p
from p in People
where (p.age < 20 and p.age > 16) or (p.name = "Smith")
```

With *not* we can either select an individual constraint or we can select a group of constraints that we wish to apply the negation operation to. If a single constraint is chosen then this constraint is highlighted by darkening or inverse videoing the constraint. If whole branches of the query have been selected for the Not operation then in addition to the constraints of that branch being shown in inverse colour, the flows linking the constraints can be highlighted also, in this case shown as dashed lines, see Figure 4 (d).

Joins

In Figure 5 (a) we see a query utilising more than one extent and comparing attributes of the class of each extent. Here we have introduced another extent called *NewPeople*. The query obtains the names of the people in the extent *People* that have the same salary as people in *NewPeople*.

```
select p.name
from p in People, q in NewPeople
where p.salary = q.salary
```

A self join can be performed by using the same extent as input to the two flows, as seen in Figure 5 (b). The OQL for this query is:

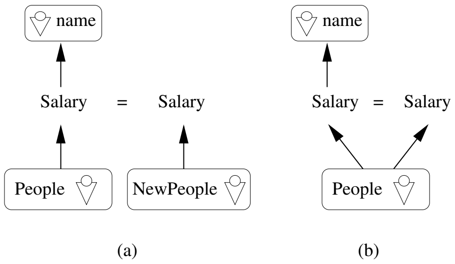

Figure 5: A join and a self join

```
select p.name
from p in People, q in People
where p.salary = q.salary
```

Aggregates

Aggregate functions, such as count, sum, avg, etc, can be applied to collection results or collection extents as shown in Figure 6. The extent to be aggregated is selected and then the aggregate function is applied to the extent. The aggregate function is then surrounded by another results box. The query in Figure 6 computes the number of persons in the extent People. The corresponding OQL for the visual query is:

```
count(People)
```

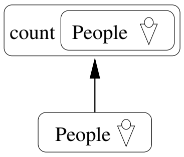

Figure 6: Evaluating the size of the extent People

Aggregates can also be applied to derived collections and collection attributes of a class. The collection attributes of a class are selected and the aggregate operators applied to them. For example, Figure 12 finds the maximum salary in a set of people's salaries.

Arithmetic

Figure 7 shows an arithmetic expression. The participant can select attributes in a class and apply arithmetic operators to them. In Figure 7 the attribute weight of person has been multiplied by 2.205 to change it from kg to lbs. The participant also has the option of naming their expression, and as can be seen in Figure 7 (b), the new attribute lbs has been selected for inclusion in the results.

The OQL representations for the queries in Figures 7 (a) and (b) are as follows:

```
select (p.weight*2.205)
from p in People

select struct(lbs:p.weight*2.205)
from p in People
```

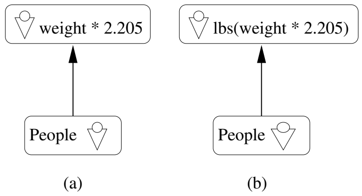

Figure 7: Arithmetic

More complex arithmetic expressions could be built using attributes from differing classes. This would be accomplished using paths which are discussed in the following section.

Path Expressions

In Figure 8 we see how a query that has been formed through navigation from one class to another related class can be visualised.

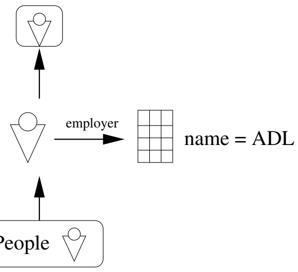

Figure 8: Path expressions

To place a constraint on a person's company, the user has to navigate from the Person class to the Company class. The navigation along the relation is visualised as a horizontal arrow with the relation name located above the arrow. After the arrow the related class is displayed. From the Company class we can select the attribute name and place the constraint that it be equal to "ADL", with the equivalent OQL for this query being:

```
select p
from p in People,
where p.employer.name = "ADL"
```

Here we have shown how a one to one relationship is visualised. In the next section we will see how a one to many relationship is visualised as well as seeing how multiple constraints on a related class are treated.

Paths with Multiple Constraints

In Figure 8 we only added a single constraint to the related class's attribute. In Figure 9 we see how multiple constraints over related attributes are visualised in the same way as was done previously in the section on boolean operators. We have navigated from the *Company* class to the set of *employees* of that company. As before this relation is visualised as an arrow going to the left with the relation name placed above the attribute. As this is a one to many relationship, in that many employees work for a company, the Person icon is surrounded by an oval box to show that there are a group of Persons rather than a single entity. To the members of this set we apply constraints to find the employees that are aged over 60 *or* who have salaries of equal or greater value than 25000. If a company does have an employee that satisfies these constraints then the name of the company is retrieved. From this example we can see how queries using *and* or combinations of *and* and *or* would be handled. The OQL follows:

```
select c.name
from c in Companies, e in c.employees
where e.age > 60 or e.salary >= 25000
```

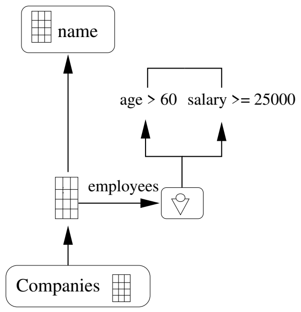

Figure 9: Path expression utilising *or*

Membership Testing and Universal and Existential Quantification

For membership testing and universal and existential quantification we have introduced a new arrow, , that links the function with the two operators of the expressions. As subqueries are evaluated, the results of these queries can be used as input to other queries. Figure 10 shows this in action. The query finds all employees that work for companies located in England. A membership test is performed on the results of the query "what companies are located in England".

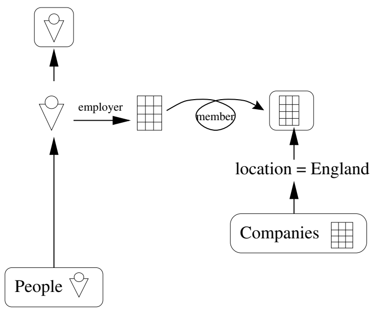

Figure 10: Membership testing

The OQL for the query shown in Figure 10 is as follows:

```
select p
from p in People
where p.employer in
    (select c
    from c in Companies
    where c.location = "England"))
```

Figure 11 depicts two queries using the *for all* and *exists* operators of OQL. These queries are very similar to the one shown in Figure 9, except that the filter flow arrow that leaves the employee set is annotated with the *all* or *exists* operation. To highlight that these functions have been applied to the filter flows we could alter the colour of the filter flow.

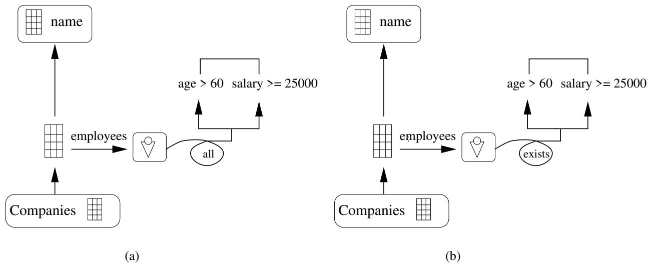

Figure 11: Universal and existential quantification

The OQL queries that are equivalent to Figures 11(a) and 11(b) respectively are as follows:

```
select c.name
from c in Companies
where for all e in c.employees:
    e.age>60 or e.salary>=25000

select c.name
from c in Companies
where exists e in c.employees:
    e.age>60 or e.salary>=25000
```

Combining operators

A more complex query is shown in Figure 12 that uses a combination of the operators that have been introduced so far. This query finds the names of companies that have employees older than 60 or that receive salaries greater than the maximum salary obtained by the Smiths that work for companies located in England. In this query we combine related classes, *and* and *or*, aggregates, and membership testing to create the visual query. The OQL that equates to the query in Figure 12 is:

```
select c.name
from c in Companies, e in c.employees
where e.age>60 or e.salary>=
    max(select p.salary
    from p in People
    where p.name = "Smith"
        and p.employer in
        (select c
        from c in Companies
        where c.location = "England")))
```

Selecting the results that we wish to view was dealt with previously. In the next three sections we shall see how we can manipulate the structure, ordering and the grouping of the results.

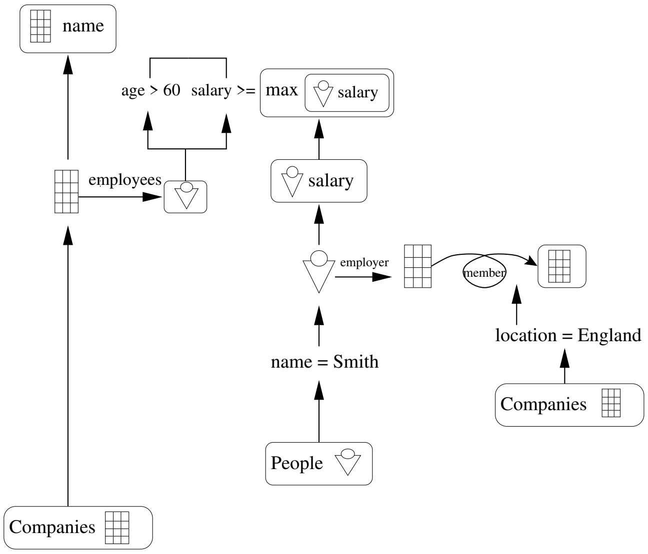

Figure 12: Complex Query

Structuring Results

One of the problems with writing an OQL query is that as you write your query, you must also concentrate on how you want the results to be structured. In Kaleidoquery we have tried to remove this complication by separating the two tasks. The participant first builds their query, with the results being visualised above the query. The participant can then select, from the available options, how the results are to be structured. When satisfied they can then view the results by examining the output.

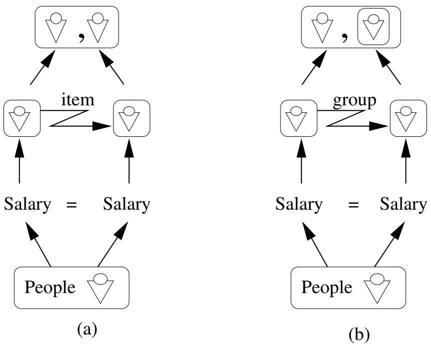

Figure 13: Structuring the results

With the queries in Figure 5 we may wish to view the results as pairs of related people with the same salary or as individuals listed with a group of people with the same salary. In OQL this is possible with the operators *struct* or *tuple* (both of which have the same behaviour) and correct nesting of the query. The OQL for obtaining pairs of related people is as follows:

```
select tuple(pPeople:p, qPeople:q)
from p in People, q in People
where p.salary = q.salary
```

The visual equivalent of this OQL query is shown in Figure 13 (a). Here we see that we wish to link each instance on the left of the query with each corresponding instance on the right that have equivalent salaries, i.e. we want to link them on a *per item* basis. This operation has been visualised with a structure linkage arrow, , with the style of structuring that we require, in this case *item* placed over the structure linkage arrow. At the top of Figure 13 we see a visualisation of the final structure of the results. Figure 13 (b) presents the visualisation for generating people with groups of people with equivalent salaries. The OQL for this query is:

```
select struct(thePerson:p,
    thePeople:(select q
    from q in People
    where p.salary = q.salary))
from p in People
```

By comparing each of the OQL queries we can see how different in nature they are. If we wish to obtain nested results then we can see that in OQL the queries require to be nested. With Kaleidoquery we can see that to change between the two queries is simply a matter of changing the structure operation; no other changes need to be made to the query as we have separated the two tasks of writing the query and structuring the results. This will allow for easier evolution of queries than OQL, where the user needs to reorganise the complete query to alter the structure of the results.

Ordering the results

In OQL, the results of a query can be sorted via the OQL operation *order by*. Visually, this can be performed by selecting the order that the results are to be presented by annotating the results with an order. If the results are nested then the ordering inside the nested results is independent of the ordering outside the results. In Figure 14 we see how the sort operator has been applied to the results of a query obtaining the name and location of companies with a list of their employee names and ages. The results of the query can then be annotated with numbers showing the order in which they are to be sorted. Along with this number is the ordering associated with the attribute. At present in OQL, results can be ordered in ascending, *asc*, or descending, *desc* order. These are visualised by an upward pointing arrow, ↑, for ascending and a downward pointing arrow, ↓, for descending. An advantage of this method of sorting over OQL is that the user can perform all the sorting at the final stage. With OQL, nested results need to be sorted before they are used in a join, as can be seen in the following OQL for the query of Figure 14.

```
select struct(name:c.name,location:c.location,
  employee:(select struct(name:p.name, age:p.age)
            from p in c.employees)
            order by employee.age desc)
from c in Companies
order by company.name asc,company.location asc
```

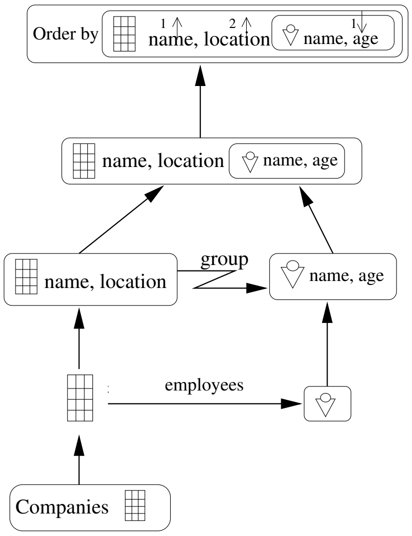

Figure 14: Ordering the results

Grouping Results

The results of a query can be partitioned into groups by application of conditions on the results. The results set is then split into subsets that satisfy the relevant partitioning conditions. Results can overlap between partitions, and there can also be results that satisfy none of the partition queries.

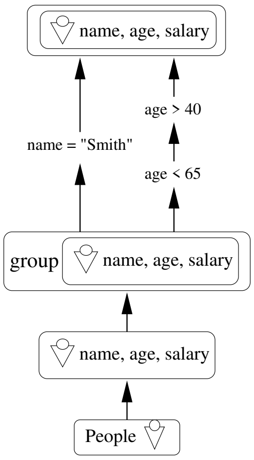

Figure 15: Grouping the results

Figure 15 shows a query where the name, age and salary of company employees are selected. We can then apply the *group by* operation to the results. This allows us to separate the results into distinct groups depending on the conditions that they satisfy. With two queries this means that the results could be partitioned into a minimum of one partition and a maximum of four partitions depending on whether the individual results satisfy none, one or both of the conditions.

The two partitionings cause the results set to become a set of results sets, as seen at the top of Figure 15.

```
select struct(name:p.name, age:p.age, salary:p.salary)
from p in People
group by
  part1:x.name = "Smith",
  part2:x.age > 40 and x.age < 65
```

After the results have been converted into partitions, a *having* clause can be used to filter the results using aggregation functions that operate on each of the partitions. In the following query (the same as the above except for the last line) the *having* clause calculates the average age of each of the partitions and only where this value is less than 50 will the partitions be present in the results. The visualisation of the *having* clause of the query can be seen at the top of Figure 16.

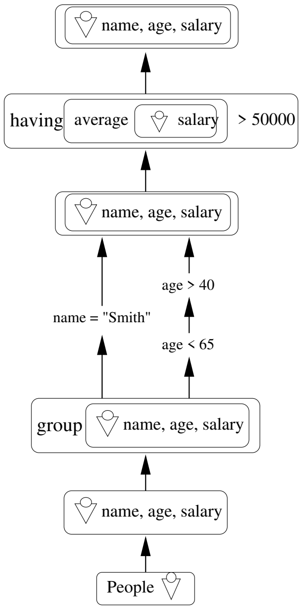

Figure 16: Filtering partitions

```
select struct(name:p.name, age:p.age, salary:p.salary)
from p in People
group by
  part1:x.name = "Smith",
  part2:x.age > 40 and x.age < 65
having avg_salary:avg(select x.salary from
   x in partition) > 50000
```

IMPLEMENTATION

The Kaleidoquery language has been described independent of any implementation details. In this section we briefly describe Kaleidoscape, the 3D implementation of Kaleidoquery that has been designed using the framework in [16]. We have chosen to implement the language in a three dimensional environment as we wanted to examine the impact of the utilisation of differing forms of hardware ranging from the standard monitors and desktop mice to head mounted displays, 3D mice and auto-stereoscopic displays. A belief exists that conventional direct manipulation interfaces i.e. WIMP (windows, icons, menus, pointers) while working well for some tasks are a limiting factor for other tasks. Interface designers are comfortably stuck in a rut, building WIMP interfaces when there is an opportunity for searching for new interface paradigms and utilising them in the creation of the next generation of interfaces [18, 21]. The interface hardware of today's computers is increasingly allowing gesture and speech recognition. We wished to research this area and examine any advantages and disadvantages of presenting the query interface in a 3D environment, rather than following the standard line of generating a windows interface, and in doing so inheriting the disadvantages as well as the advantages of such an interface style. A more thorough description of the database query and browsing interface will be presented in a future paper, but a brief outline is given now.

Currently the database schema is displayed as 3D icons composed of a class visualisation and the class's name. We have not displayed the database schema using ER diagrams or similar notations as novice users may not be familiar with such notations, although this does not rule out the examining the use of ER diagrams for displaying the database schema and query to users familiar with such notations in the future.

The participant has the option of specifying the layout and grouping of these classes, or a graph algorithm can be used to manage the layout. Above each of the database classes may be placed an extent associated with that class. To examine either a class or an extent the participant selects the visualisation. On selection they are brought to the location of the class or the extent and see either the attributes of the class, or will move *inside* the extent, and move to a different environment where they are able to browse over the instances in the extent.

To build a query, the participant must first select the class with which they wish to begin querying. They are brought to the class and the visualisation of the class alters to show its attributes. If *simple* attributes of a class are chosen, e.g. strings, numbers, etc. then they can select operators to apply to that attribute depending on the attribute type, e.g. =, >, >=, <, <= and *like*. After the operator is selected they can then either enter a value, select an attribute for comparison or build any other form of condition that is valid for completing the constraint.

For example, in Figure 17 we see a simple database schema composed of several classes. In this figure we also see a visualisation of the participant's *hand* that is controlled with a 3D mouse. This *virtual hand* allows the participant to navigate through the environment and also to select artifacts in the environment. The participant does not need to use the 3D mouse but could use the keyboard and desktop mouse for navigation and selection. When the participant selects the *Person* class with their *virtual hand* or mouse pointer, the display zooms into the *Person* class, until in Figure 18 the attributes are presented. From these the participant can select an attribute to apply a constraint to. The participant selects

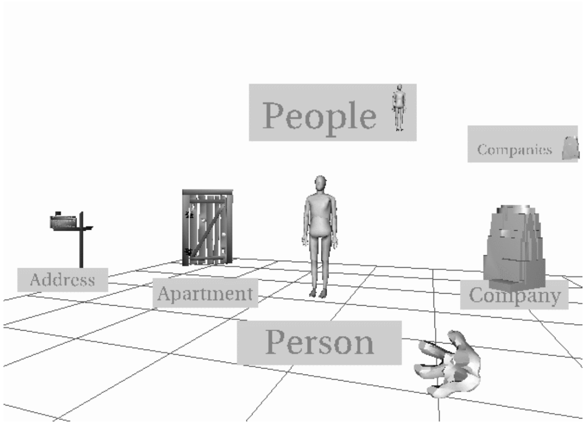

Figure 17: Database Schema

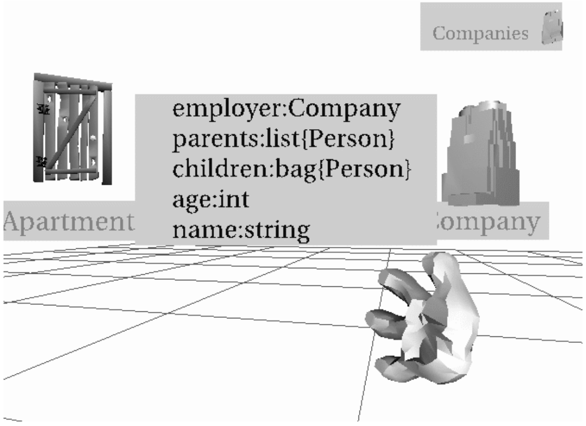

Figure 18: Person attributes

the attribute *age* and some operations that can be applied to the attribute appear, as shown in Figure 19. The participant can then select the less than operation, <, and enter the value 20 at the keyboard. On completion of this, the query in Figure 20 is shown (this is the same as the query in Figure 3).

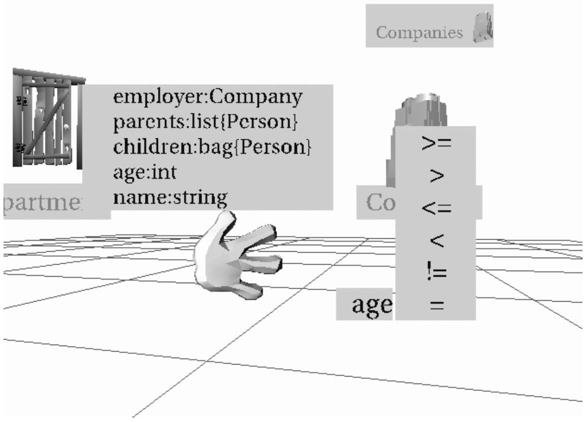

Figure 19: Age operations

At present the interface chooses the *People* extent as the input to the query as no other extent over the class *Person* exists. The participant can then select to view the results by selecting the results visualisation at the top of the query. As with examining extents, the participant is transferred to another environment to view the results, with the ability to alter the query while they are in the results space and so alter the results visualisation. Alternatively, they can continue adding more constraints to the query, or specifying which attributes they wish the results to contain.

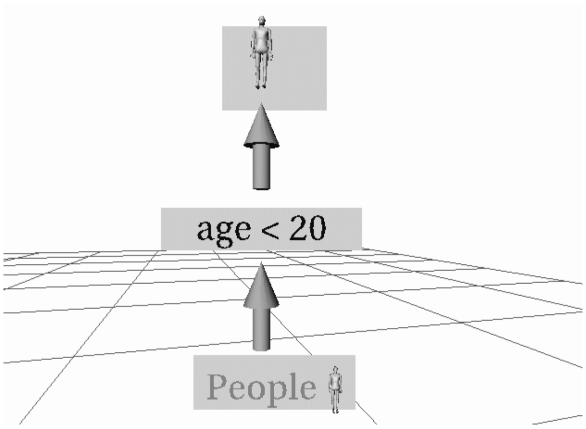

Figure 20: Simple query

If the participant were to select a *complex* attribute, i.e. one that relates to another class, then the participant is moved to that class and the attributes of the related class come into view. They can then proceed as before, selecting *simple* attributes to be used in constraints or selecting *complex* attributes and being navigated towards them.

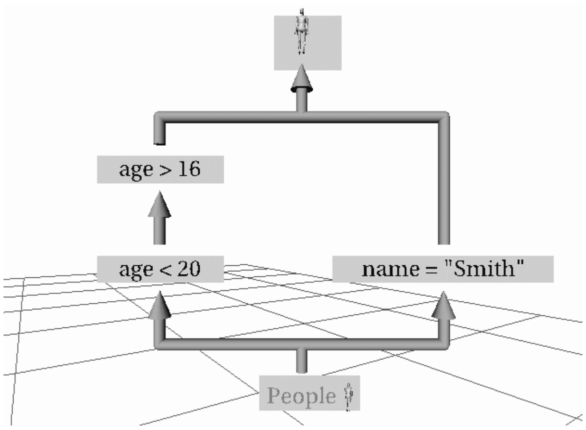

Figure 21: Visualising *and* and *or*

If we continue and add some further *simple* constraints on the attributes of the *Person* class, we could build the query as shown in Figure 4 (c). The equivalent of this query in our present 3D environment is shown in Figure 21.

As can be seen from the screenshots, the display of attributes and operations available to the participant is limited to a simple textual menu-like display. At present we are implementing the Kaleidoquery language in the 3D environment. When this is completed we shall concentrate our efforts on enhancing the 3D interface, the interaction techniques and participant support within the environment.

Preliminary prototypes and evaluation of the Kaleidoquery interface were undertaken using VRML (Virtual Reality Modelling Language) version 2.0. We then progressed to implementing the Kaleidoscape interface on Silicon Graphics workstations using the 3D graphics library OpenInventor and C++. The environment currently supports the use of 3D mice, desktop mice, and keyboards for user input, with output being displayed via stereo projectors, head mounted displays, auto-stereoscopic displays, or normal monitors. Audio output is achieved through the use of a midi library. Queries constructed in Kaleidoscape are translated into OQL and these are then passed via the network to the database. The query is evaluated by the database, and the results returned over the network to be displayed in the Kaleidoscape environment. We are currently using the O_2 object database to obtain the schema information that is displayed in the environment Kaleidoscape, and also to execute the OQL queries that are generated from the visual queries constructed in the Kaleidoscape environment.

RELATED WORK

Visual query languages have attempted to bridge the gap of usability for users (for a survey see [6]). Forms based query languages such as QBE [23], present the database structure as tables or forms into which queries can be placed. Graph based query languages (e.g. Guidance [12], Gql [17]) have the advantage over forms style interfaces in that they can directly represent relationships within the structure of the database and the query. Icon based languages (e.g. Iconic Browser [20]) represent database concepts pictorially and can allow for the direct manipulation of icons to represent queries. Multi paradigm query interfaces also exist to allow the user to pick and choose or alternate between interface styles, [9, 5]. Visual query languages are now also incorporating three dimensional aspects into the query language (e.g. AMAZE [4]) but with AMAZE constraints are still specified via a forms interface. In addition AMAZE is much less powerful than Kaleidoscape lacking such features as aggregation. Many advances are being made in the area of dynamic query visualisation (e.g. Spotfire [1]) though these tend to be limited in the expressive power of their queries and the range and type of information they can query.

Current graphical query languages to ODMG compliant object databases are limited to Quiver [8] and GOQL [14]. Only two simple examples of the Quiver query language are given in [8], but preliminary evaluations have shown that it is easier to use than the standard textual OQL interface. GOQL is a graph style query language and a complete description of its constructs is given in the paper. Neither of the query languages describe the interface that has been implemented to support the visual language.

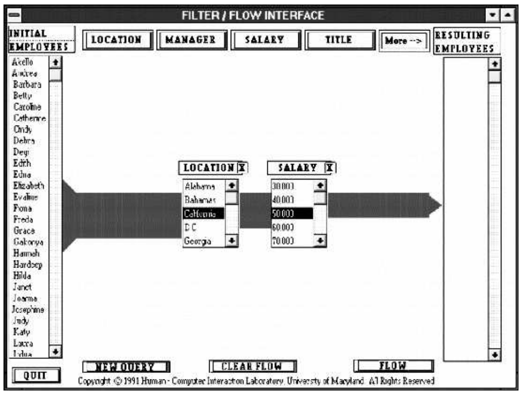

Figure 22: Filter flow interface

The filter flow model for depicting the boolean *and* and *or* relations had a prototype constructed and evaluated, [22], see Figure 22. The prototype was not connected to a real database but used a single relation over which users could select the attribute they wished to constrain. From this operation they would obtain a list of all the instances of that attribute in the table. Instances that the user was interested in could then be selected. Figure 22 depicts an *and*-ed query. It can also be seen in the figure how they altered the thickness of the data flow to reflect the size of the results, although with the prototype actual values were not used but rather the thickness was determined heuristically. The early evaluations of this form of depiction of the boolean operators has shown promise, and so was adopted for the Kaleidoquery language and extended so as to be used with a real database.

CONCLUSION

We have described the design of Kaleidoquery, a graphical query language for the querying of object databases. We have described the features of OQL and shown how they are supported by Kaleidoquery. Queries produced by Kaleidoquery are translated into OQL and then submitted to an object database so the Kaleidoquery interface should be compatible with any ODMG compliant database requiring minimal modifications.

The design philosophy behind Kaleidoquery has been to build a language that can be used by any database user without requiring any knowledge of database diagrams such as entity relationship diagrams. Although the users will have to learn the constructs of Kaleidoquery, and the metaphor behind the filter flow design of the queries, we hope this will be easier to master than a textual OQL interface with its well known problems of:

1. steep learning curve,
2. semantic and syntactic errors,
3. the structure of database classes, attributes and relationships is not readily available to the users,
4. the increasing complexity of specifying the order of boolean operators with parenthesis as the query grows,
5. the differences in meaning between the English and boolean logic meaning behind the *and* and *or* functions.

By creating a visual query language and generating a display of the database schema with interactive help, and utilising the filter flow technique for representing boolean operations, we have produced an interface that attempts to alleviate these problems and provides:

1. a powerful visual query language for OODBs, supporting the capabilities of the OQL language. In this paper we have examined the main concepts of OQL and shown their equivalent visual representation in Kaleidoquery. A more formal specification of the language is under development.
2. compliance with the ODMG model version 2.0 and consistency with OQL with its well understood language constructs plus direct support for evaluation,
3. a filter flow oriented visual model,
4. an implementation in 3D, much more expressive than earlier 3D interfaces to databases,
5. a separation of the tasks of writing the query constraints and organising the structure and ordering of the results, that supports a more dynamic evolution of queries than OQL.

We also hope that the filter flow approach to query construction will make the task of building and understanding queries easier than graph based queries, although we shall of course need to test this hypothesis with some user evaluations.

At present few visual query languages for object databases exist and even fewer query languages are implemented in a 3D environment. Many visual query languages have been implemented under WIMP (window, icon, menu, pointer) environments. As well as creating a visual query language for object databases we wish to examine the advantages and disadvantages of a 3D environment for query construction, and to examine how we can use the added dimension to aid the user in understanding and navigating the database schema, composing a 3D query, visualising and manipulating the results.

Much work is being undertaken in the visualisation of complex data in a 3D environment. We wish to extend this by testing a 3D query environment and furnishing it with appropriate tools to ease the task of querying a database, and removing the need for the user to learn the syntax of a complex textual language.

REFERENCES

1. Christopher Ahlberg. Spotfire: An Information Exploration Environment. *SIGMOD Record*, 24(4):25–29, December 1996.
2. W. L. Bewley, T. L. Roberts, D. Schroit, and W. L. Verplank. Human Factors Testing in the Design of Xerox's 8010 'Star' Office Workstation. In *Proceedings ACM CHI'83 Conference*, pages 72–77, Boston, MA, December 1983.
3. Christine L. Borgman. The User's Mental Model of an Information Retrieval System; An Experiment on a Prototype Online Catalog. *International Journal of Man-Machine Studies*, 24:47–64, 1986.
4. J. Boyle, S. Leishman, and P. M. D. Gray. From WIMP to 3D: the development of AMAZE. *Journal of Visual Languages and Computing*, 7:291–319, 1996.
5. T. Catarci, S. K Chang, and G. Santucci. Query Representation and Management in a Multiparadigmatic Visual Query Environment. *Journal of Intelligent Information Systems*, 3(3):299–330, 1994.
6. Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, and Carlo Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.
7. R. G. G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.
8. Manoj Chavda and Peter Wood. Towards an ODMG-Compliant Visual Object Query Language. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 456–465, 1997.
9. D. K. Doan, N. W. Paton, and A. C. Kilgour. Design and User Testing of a Multi-paradigm Interface to an Object-Oriented Database. *ACM SIGMOD Record*, 24(3):12–17, September 1995.
10. C. Egido and J. Patterson. Pictures and Category Labels as Navigational Aids for Catalog Browsing. In *Proceedings ACM CHI'88 Conference*, pages 89–91, 1988.
11. S. Greene, S. Devlin, P. Cannata, and L. Gomez. No IFs, ANDs, or ORs: A Study of Database Querying. *International Journal of Man-Machine Studies*, 32:303–326, 1990.
12. David Haw, Carole Goble, and Alan Rector. GUIDANCE: Making it Easy for the User to be an Expert. In *Proc. 2nd Int. Workshop On Interfaces to Database Systems*, pages 19–43. Springer-Verlag, 1994. P. Sawyer (Ed).
13. C. J. Kacmar and J. M. Carey. Assessing the Usability of Icons in User Interfaces. *Behaviour and Information Technology*, 10(6):443–457, 1991.
14. Euclid Keramopoulos, Philippos Pouyioutas, and Chris Sadler. GOQL, a Graphical Query Language for Object-Oriented Database Systems. In *Basque International Workshop on Information Technology*, pages 35–45, 1997.
15. A. Michard. Graphical Presentation of Boolean Expressions in a Database Query Language: Design Notes and an Ergonomic Evaluation. *Behaviour and Information Technology*, 1(3):279–288, 1982.
16. Norman Murray, Carole Goble, and Norman Paton. A Framework for Describing Visual Interfaces to Databases. To be published in the Journal of Visual Languages and Computing.
17. A. Papantonakis and P. J. H. King. Syntax and Semantics of Gql, a Graphical Query Language. *Journal of Visual Languages and Computing*, 6:3–25, 1995.
18. Jef Raskin. Looking for a Humane Interface: Will Computers Ever Become Easy to Use? *Communications of the ACM*, 40(2):98–101, 1997.
19. Ben Shneiderman. Visual user interfaces for information exploration. In *Proceedings of the 54th Annual Meeting of the American Society for Information Science*, pages 379–384, Medford. NJ, 1991. Learned Information Inc.
20. K. Tsuda, M. Hirakawa, M. Tanaka, and T. Ichikawa. Iconic Browser: An Iconic Retrieval System for Object-Oriented Databases. *Journal of Visual Languages and Computing*, 1(1):59–76, 1990.
21. Andries van Dam. Post-WIMP User Interfaces. *Communications of the ACM*, 40(2):63–67, 1997.
22. Degi Young and Ben Shneiderman. A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation. *Journal of the American Society for Information Science*, 44(6):327–339, 1993.
23. M. Zloof. Query-By-Example: A Data Base Language. *IBM Systems Journal, Vol. 4*, pages 324–343, December 1977.