Modeling Shapes in an Image Database System[*]

Vincent Oria, M. Tamer Özsu, Lin Irene Cheng, Paul J. Iglinski and Yuri Leontiev
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{oria, ozsu, lin, iglinski, yuri}@cs.ualberta.ca

Abstract

Due to the complex modeling requirement of data handled by image and spatial databases, they are most often built on top of object-oriented or object-relational databases. In the DISIMA image database, an image is composed of salient objects and a salient object has a shape which is a geometric object. The object-oriented modeling of shapes potentially conflicts with the mathematical definitions of geometric objects. Mathematically, a triangle and a rectangle are polygons and a square is a special kind of rectangle. Accordingly, a class Triangle *should be a subclass of the class* Polygon*. In the same way, a class* Square *should be a subclass of* Rectangle *which, in turn, should be defined as a subclass of* Polygon*. But from the point of view of data representation, this leads to a conflict: A polygon minimally requires a list of n consecutive points for its description, whereas a rectangle can be defined by just three points and a square by just two points, if we take advantage of their symmetry. This paper proposes an object-oriented modeling of shapes that accords with their mathematical definitions, optimizes their data representations, and lends power for shape similarity queries.*

1 Introduction

In most spatial databases only a few geometric shapes are represented. The commonly found shapes are points, lines, and polygons. This was sufficient because the first applications of spatial databases were geographic information systems with just points, lines, and polygons. But some other applications, like graphic design, may need precise descriptions of circle, ellipse and other basic geometric shapes.

Due to the type of data handled by spatial databases, they are most often built on top of object-oriented or object-relational databases. Hence, the spatial model should follow object-oriented modeling principles. The modeling of the representational data structures, however, conflicts with the natural hierarchy of geometric objects. Mathematically, a triangle and a rectangle are polygons, and a square is a special kind of rectangle. Following the mathematical definition, the class *Triangle* should be a subclass of the class *Polygon*. In the same way, the class *Square* should be a subclass of *Rectangle* which, in turn, should be a subclass of *Polygon*. Herein lies the conflict. From the point of view of data structures, a polygon, in general, is minimally described by a list of n consecutive points. A rectangle, on the other hand, can be defined by just three points, and a square by just two points, taking advantage of their inherent symmetry. Thus, less data, rather than more, is sometimes sufficient for the more derived types.

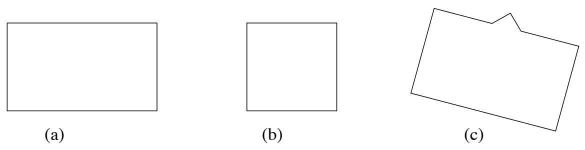

Figure 1: Shape similarity.

The problem has already been addressed in [SLY95] with an elegant solution when subclass and superclass attributes are linearly related (i.e. an object of a subclass is a constrained version of an object from its superclass). This is the case for classes *Square* and *Rectangle*. But the attributes in *Rectangle* are not linearly related to the attributes in *Polygon* since a polygon is represented by a list of point. The IUE (Image Understanding Environment) [Env99] is currently building class hierarchies for image applications. The IUE work is praiseworthy in the sense that it includes a great variety of shapes but the class hierarchy is built using the *abstract and concrete class* model

[*]This research is supported by a strategic grant from the Natural Science and Engineering Research Council (NSERC) of Canada.

[LRG93]. This model keeps the (abstract) hierarchy logically correct but concrete *Square* is totally disconnected from concrete *Rectangle*, for example. As a result, there is no concrete class hierarchy and the code in concrete classes is not reused.

This paper describes the design and the implementation of geometric shapes in the DISIMA (DIStributed Image database MAnagement system) [OÖL$^+$97] project following the model presented in [LOS98]. The model proposes a total separation between interface, implementation, and representation by providing *implementation types* that are used to describe the internal representation of data. We use the term *interface type* to refer to real-world entities and their programmatic interface. The term *implementation type* refers to the internal data representation; *class* or *concrete type* refers to the creation of instances and extent maintenance. Thus, an *interface type* defines a programmatic interface, while an *implementation type* defines an internal data representation. A class (concrete type), which is capable of producing new instances, is multiply derived from an interface type and an implementation type. The entire group of objects of a particular interface type, including its subtypes is known as the *extent* of the interface type and is managed by its class (concrete type). We refer to this as *deep extent* and introduce *shallow extent* to refer only to those objects created directly from the given type without considering its subtypes.

Interface type and implementation type hierarchies are totally independent. Usually, there is one-to-one correspondence between interface types, implementation types, and classes. However, it is possible to use the same implementation type for classes of unrelated types. This occurs when two unrelated interface types use the same or related internal representations. It is also possible to implement an interface type using more than one implementation type. Therefore, objects of the same interface type can have different internal representations. For example, within the same application, some points can be created using Cartesian coordinates while others are represented using polar coordinates. Each type of point will belong to a different concrete type and share the same interface type.

DISIMA, implemented in C++ on top of a commercial object-oriented DBMS, *ObjectStore*, allows image content-based queries at different levels. A user can ask “show me all the pictures containing objects of polygonal shape”. The answer to this query will be incomplete if it does not contain the images with triangles, rectangles or squares. The hierarchical modeling of shapes is more interesting for queries like “show me all the pictures containing an object with a shape similar to a given one”. The given target shape might be a rectangle, while the most similar shapes might be among the squares or polygons. In the example given in Figure 1, shape (c), a seven-sided polygon, is close to the shape of the target rectangle (a); an image containing (c) should be returned as a result of “show me all the pictures containing an object with a shape similar to shape (a)” DISIMA’s shape similarity retrieval algorithms can be combined with other retrieval mechanisms based on color, texture, and spatial relationships [Del99].

The remainder of this paper is organized as follows: Section 2 gives an overview of DISIMA and presents how we solved the shape modeling problem. Section 3 describes how shape similarity is handled in DISIMA, and Section 4 provides a conclusion.

2 Modeling Geometric Objects in DISIMA

This section describes the DISIMA model and the design and the implementation of geometric shapes in DISIMA. In the DISIMA model, an image is composed of physical salient objects, which are, in part, geometric objects (without any semantics) in a space (defined by the image’s co-ordinate system). In addition to shape, a physical salient object has color and texture properties. A physical salient object gets its meaning (semantics) from a logical salient object (LSO) with which it is associated.

2.1 The DISIMA Model: An Overview

The DISIMA model provides an efficient representation of images and related data to support a wide range of queries. The DISIMA model, as depicted in Figure 2, is composed of two main blocks: the image block and the salient object block. We define a *block* as a group of semantically related entities.

The image block is made up of two layers: the *image* layer and the *image representation* layer. We distinguish an image from its representations to maintain an independence between them, referred to as *representation independence*.

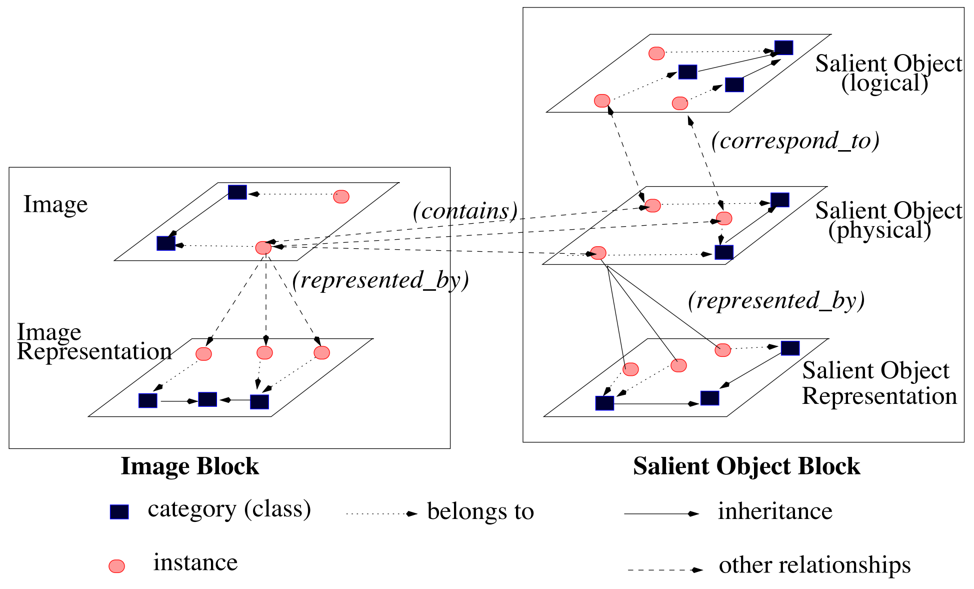

Figure 2: The DISIMA Model Overview.

At the *image* layer, the user defines an image type classification. Figure 3(b) depicts a partial type hierarchy for an application that involves medical images, electronic commerce catalogs, and news images. These first level image types are derived from the type *Image*, the root image type provided by DISIMA. The type *NewsImage* is specialized by three types: *EnvironmentalImage*, *PersonImage*, and *MiscImage*.

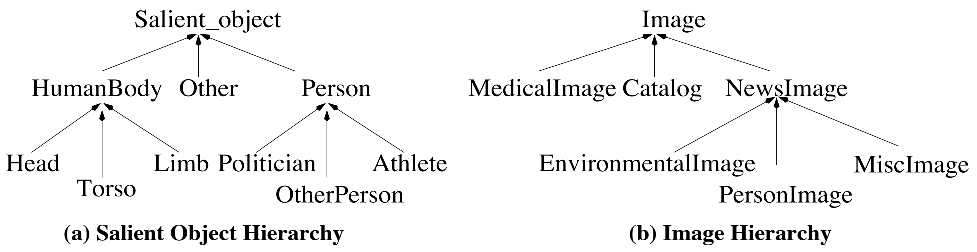

Figure 3: An Example of an Image Hierarchy.

The *salient object* block is designed to handle salient object organization. A simple example of a salient object hierarchy, corresponding to the image hierarchy defined in Figure 3(b), is given in Figure 3(a).

DISIMA distinguishes two kinds of salient objects: physical and logical salient objects. A *logical salient object* is an abstraction of a salient object that is relevant to some application. For example, an object may be created as an instance of type *Politician* to represent President Clinton. The object “Clinton” is created and exists even if there is yet no image in the database in which President Clinton appears. This is called a *logical salient object*; it maintains the image independent generic information that might be stored about this object of interest (e.g., name, office, spouse). Particular manifestations of this object (called *physical salient objects*) may appear in specific images. There is a set of information (data and relationships) linked to the fact that “Clinton appears in an image”. The data can be the colors of his clothes, his localization, or his shape in this image.

2.2 Geometric Shape Modeling

The geometric objects we are interested in are point, segment, polyline, ellipse, circle, polygon, triangle, and square. We give their mathematical definitions below.

- A **point** is defined by its x and y coordinates on the X-Y plane.

- A **polyline** is defined by n consecutive and unique points; the $n-1$ consecutive pairs of these points describe the end points of $n-1$ non-intersecting lines.
- A **segment** is a polyline with $n = 2$.
- An **ellipse** is defined by two points, the foci, and a "diameter", the sum of the distances from any point on the ellipse to the two foci.
- A **circle** is an ellipse with the two foci equal.
- A **polygon** is defined by n consecutive and unique points; the $n-1$ consecutive pairs of these points describe the end points of $n-1$ non-intersecting lines; the first and n^{th} point describe the end points of the n^{th} non-intersecting line to close the shape.
- A **triangle** is a polygon with $n = 3$.
- A **rectangle** is a polygon with $n = 4$, and each segment is 90^o clockwise from the previous one.
- A **square** is a rectangle with all four segments of the same length.

Following the mathematical definitions given above, the desired design of a shape hierarchy is given in Figure 4. A shape (*Geometric_Object*) can be *Composite* or *Atomic*. A composite shape is comprised of more than one atomic shape. Atomic shapes are further divided into three categories: *Point*, 2-dimensional (*2D*) and 1-dimensional (*1D*) shapes. A shape which has an area, e.g., a polygon, is classified under *2D*. A 1D shape has length but not area. Due to the special characteristics of the point shape, it is not classified under *2D* or *1D*. In the proposed design, two class groups are defined under *2D*: the *Polygon* and *Ellipse* groups. The *Polyline* group is defined under *1D*. The *Atomic* class can be extended to incorporate more shapes, such as curve or arc, if required.

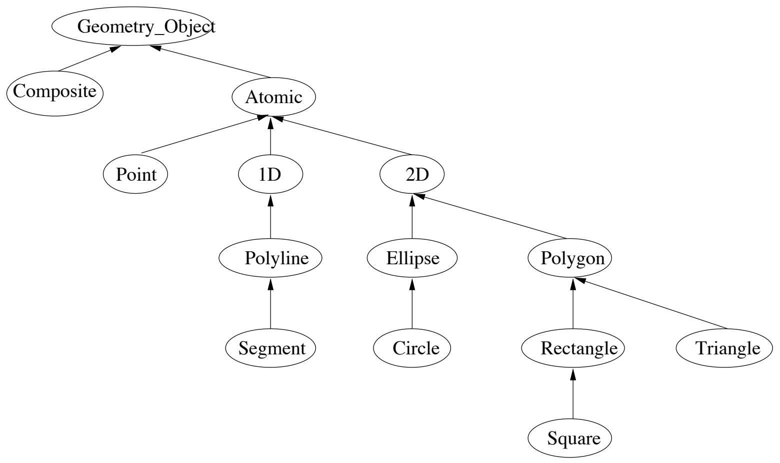

Figure 4: The Desired Geometric Object Hierarchy.

In the classical object-oriented approach, a subclass is more specific and specialized than its superclass. In other words, the subclass is defined by adding more data members or functions. From the point of view of data representation, *Ellipse* should be a subclass of *Circle* because a circle can be defined by its center and its diameter, while an ellipse requires more data: two points (the foci) and a total distance to the foci. However, from a geometrical point of view, *Circle* should be a subclass of *Ellipse* because a circle is defined by imposing an additional restriction on an ellipse; that is, the two foci have to be equal. A similar argument applies to the *Rectangle* and *Square* classes.

The problem can be solved using a model that completely separates interface, implementation, and representation by providing *implementation types* that are used to describe the internal representation of data.

2.3 The Type System: Implementing the Shape Hierarchy

We use three different kinds of C++ classes to simulate the notions of *interface type*, *concrete type* and *implementation type* defined in our model. The interface types, whose names are not prefixed, declare the interface visible to the user; these types are usually abstract with pure virtual functions. An exception is the concrete interface type *Composite*. Some of the classes in this interface type hierarchy, e.g., *Atomic*, *1D*, and *2D*, are termed *abstract type*, since only their subtypes have a create method. The *I*_-prefixed classes are the implementation types; they contain the actual data members. The *C*_-prefixed classes represent concrete types; each of these concrete classes is publicly derived from its interface type and privately derived from its implementation type. Each concrete type provides the implementation of virtual methods in the interface type using the data structures of the implementation type. The interface types contain static factory methods for creating instances of the appropriate concrete types; the *new* operator is not used directly. Thus, when the create method for *Square* is invoked, the object created is actually a *C_Square*, but it is handled by the user as type *Square*.

Every interface type (except the abstract interface types) has a static method *shallowExtent* that returns the extent of the associated concrete type(s). A static method *deepExtent* returns the extents of the associated concrete type and the concrete types associated to the descendant types. For example, the method *shallowExtent* in *Polygon* returns a set of *C_Polygon* objects whereas *deepExtent* will return the extents of *C_Polygon*, *C_Triangle*, *C_Rectangle*, and *C_Square*.

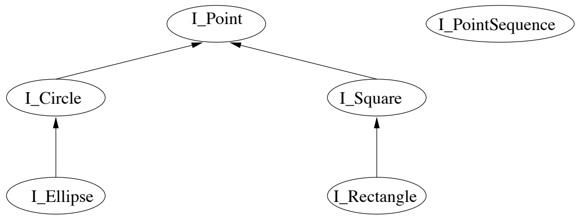

Figure 5: The Implementation Type Hierarchy

The implementation types are organized into a hierarchy as shown in Figure 5 with two roots, *I_Point* and *I_PointSequence*. The implementation class *I_PointSequence* contains one data member, a list of points (a list knows its size). This defines the implementation type for the types *Polygon*, *Triangle*, *Polyline*, and *Segment*. The implementation class *I_Point* has two subclasses, *I_Square* and *I_Circle*. *I_Rectangle* inherits from *I_Square* whereas *I_Ellipse* inherits from *I_Circle*.

Figure 6 shows the final design of the geometric object class hierarchy. All the concrete types are associated with an implementation type and an interface type. Abstract interface types do not have any shallow-extent; they are not associated to any concrete type.

3 Shape Similarity

The *Geometric_Object* class supports three types of similarity match: *full-group, class* and *sub-group,* depending on the similarity threshold specified in the query. The *ellipse* group includes the *Ellipse* and *Circle* classes, and the *polyline* group includes the *Polyline* and *Segment* classes. The *Polygon*, *Rectangle*, *Square* and *Triangle* classes belong to the *polygon* group. The shape similarity algorithm we used is the *turning angle algorithm* [ACH$^+$91] because it is orientation invariant. Basically, the algorithm takes a polygon containing n edges and vertices. It starts from any point on the boundary and traverses the edge counter-clockwise until a vertex is encountered. The first turning angle is the external angle formed by the first edge and an extension of the second edge. A total of n turning angles is recorded after traversing all the edges. Since the external angles of a polygon add up to 360^o, the sum of the turning angles should also be 360^o. With the perimeter normalized to '1', a step-like graph is obtained by plotting the cumulative angle on the vertical axis, and the distance traversed on the horizontal axis. The similarity is defined by a metric on the graphs. Each type class in the *Polygon* subtree provides a behavior, *vertices*, that returns the list of points comprising the vertices of a given object. Each shape is placed in its most specific class. In the following, we give some examples of shape similarity queries:

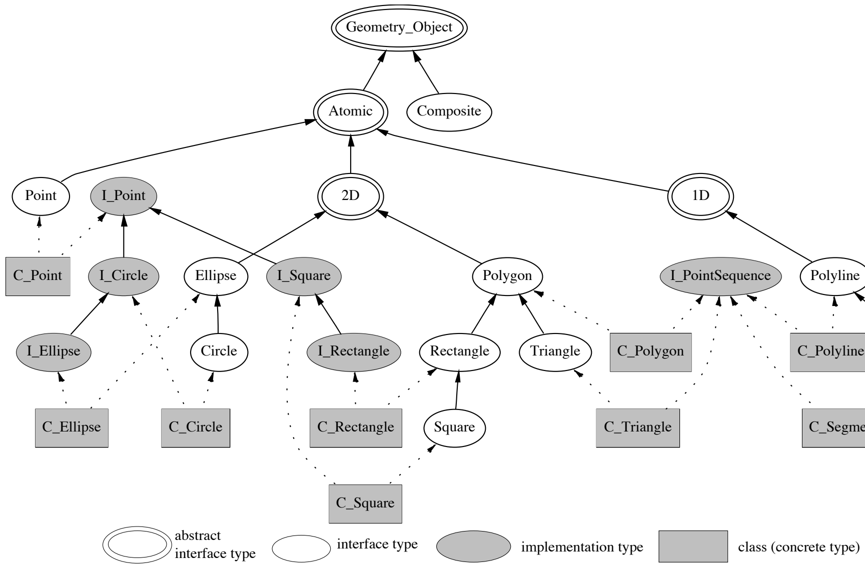

Figure 6: The Final Geometric Object Hierarchy

- Similarity without a given shape

```
SELECT image
FROM image m, lso o
WHERE m contains o
AND o.shape similar rectangle similarity 0.5;
```

In the example given above, the query is understood as “find all images containing salient objects with a rectangular shape”. When a shape is not given in a shape similarity query, the query is processed without any similarity metric. For the example, the query processor will select images for which at least one salient object has a shape of interface type *Rectangle*. The question is which extent to use (shallow or deep extent)? This decision is made with regard to the similarity threshold in the query. If the similarity threshold is set to 1, the query is processed using the shallow extent (class match) otherwise the deep extent is used (sub-group match).

- Similarity with a given shape

```
SELECT image
FROM image m, lso o
WHERE m contains o
AND o.shape similar rectangle ((1,2),(10,2),(10,7))
similarity 0.5;
```

The above query expresses “find all images containing salient objects with a a shape 50% similar to the rectangle ((1,2),(10,2),(10,7))”. If we let ϵ denote the threshold, r the rectangle ((1,2),(10,2),(10,7)), S a set of shapes, PSO a set of physical salient objects , and I a set of images, then the solution of the similarity query is $\{i \in I | \; \exists s \in S, \; \exists o \in PSO, \; d(s,r) \leq \epsilon \wedge s = shape(o) \wedge i \; contains \; o\}$ where d is a distance function using the shape similarity algorithm. The problem here is to find the minimal extent that contains all the shapes satisfying the conditions. For example, if we are trying to match a given rectangle within a similarity threshold of 0.9, should we begin our search with the deep extent of all polygons or simply confine our search to the extent of all rectangles? Are we willing at higher thresholds, to miss matching polygons like Figure 1(c) when we match against a similar rectangle? Of course the lower ϵ is, the wider both the solution and the shape search space will be. In the current implementation, full-group match is applied when the similarity threshold in the search condition is less than 1. Class match is applied when the similarity threshold equals 1 (exact match).

4 Conclusion

We have presented the outline of an object-oriented type system design for modeling geometric shapes within the context of our distributed image database management system, DISIMA. The model provides a generalizable solution to the conflict between data representation and class specialization in a hierarchical type system with code reusability at the interface level as well as the data representation level. More specialized shapes, like a circle, require less representation data than a more general shape like an ellipse. Unlike the work conducted by the IUE group, our concern is not to provide a large variety of shape definition but to provide a general way of defining geometric objects with data representation and code resusability. A cost trade-off is additional virtual function resolution at runtime.

The definition of the geometric objects is based on a model that clearly separates interface, implementation, and representation. In this model a *interface types* refer to real-world entities and their programmatic interfaces, *implementation types* refer to the internal data representations, and *classes* involve the creation of instances and extent maintenance. Interface type and implementation type hierarchies are totally independent. Each of these notions is then represented by a C++ class. The data representation requirements determine the *implementation type* hierarchy. For example *I_Rectangle* inherits from *I_Square* which in turn is a subclass of *I_Point*. The *interface type* hierarchy defines the programming interface and follows the natural mathematical hierarchy: *Square* is a *Rectangle* which in turn is a *Polygon*. A concrete class like *C_Rectangle* inherits from both *I_Rectangle* and *Rectangle*. None of the previous solutions provides code reusability at both the interface and the representation levels.

We also show how this type system can be leveraged in executing different forms of shape similarity queries. A turning angle algorithm can be used with a metric for matching shapes of different types in the polygon group against a similarity threshold, regardless of the number of vertices they contain. Future work could extend this to the ellipse group by approximating those shapes with polygons. Further experiments are planned to explore restricting shape similarity queries to class match searches when thresholds are less than 1.

One direction of our current research involves the development of indexing techniques for optimizing similarity matching.

References

[ACH+91] E. M. Arkin, L. P. Chew, D. P. Huttenlocher, K. Kedem, and J. S. B. Mitchell. An efficiently computable metric for comparing polygonal shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3), March 1991.

[Del99] A. Del Bimbo. *Visual Information Retrieval*. Morgan Kaufmann Publishers, 1999.

[Env99] Image Understanding Environment. The IUE class hierarchy. http://www.aai.com/AAI/IUE/IUE.html, 1999.

[LOS98] Y. Leontiev, M. T. Özsu, and D. Szafron. On separation between interface, implementation and representation in object DBMSs. In *Proceedings of Technology of Object-Oriented Languages and Systems 26th International Conference (TOOLS USA98)*, pages 155—167, Santa Barbara, August 1998.

[LRG93] L. Lavazza, A. Raybould, and J. A. Grosberg. Comments on considering "class" harmful. *Communication of the ACM*, 36(1):663—685, 1993.

[OÖL+97] V. Oria, M. T. Özsu, X. Li, L. Liu, J. Li, Y. Niu, and P. J. Iglinski. Modeling images for content-based queries: The DISIMA approach. In *Proceedings of 2nd International Conference of Visual Information Systems*, pages 339—346, San Diego, California, December 1997.

[SLY95] W. Sun, Y. Ling, and C. T. Yu. Supporting inheritance using subclass assertions. *Information Systems*, 20(8):663—685, 1995.