GEOMETRIC REASONING: A NEW PARADIGM FOR PROCESSING GEOMETRIC INFORMATION

{Farhad Arbab, Jeannette M. Wing}

Computer Science Department
University of Southern California
Los Angeles, California 90089-0782, USA

Abstract: Existing approaches to geometric modeling use rigid, static data structures, often tuned for one specific application. This inflexibility restricts users to inadequate means of manipulating geometric models. Our alternative approach, Geometric Reasoning, applies deductive reasoning to manipulate geometric information at an abstract level. Geometric reasoning is finding attributes of geometric objects using their intrinsic properties, their relationships with other objects, and the inference rules that bind such properties together in a geometric space. Instead of using data structures tailored for numerical computation, we use an inference mechanism that understands the semantics of geometric objects and allows dynamic definition of abstractions, e.g., shapes and relationships.

1. Motivation

Recent advances in computer graphics have stimulated an interest in using visual techniques as a means of communication between people and machines. Computer graphics activity can be divided roughly into picture rendering and geometric modeling. Geometric modeling refers to the construction and manipulation of data structures to represent geometric information; picture rendering is production of visual images from a geometric model. The most dramatic advances in computer graphics have occurred in picture rendering. Problems of picture rendering are by now well understood to the extent that commercial black-box hardware and software packages for production of life-like images are available. Consequently, application programs no longer need to carry the burden of rendering, but can focus on modeling.

Modeling, and more generally, processing of geometric information is a prime concern in not only graphics, but also in applications like computer-aided design of mechanical parts, computer-aided process planning, computer-aided manufacturing, programming of industrial robots, and computer vision. The increasing interest and activity in all these areas indicate the growing importance of geometric information processing in future computing. Historically, the focus of most such activity has been on individual applications, e.g., computer-aided drafting, in isolation from other related applications, e.g., manufacturing process planning. Recently, because of the recognition that geometric information is common to many of these applications, e.g., CAD and CAM, there has been a move towards using unified schemes, e.g., CAD/CAM databases, to represent this common information. This trend that leads to databases for geometric information is analogous to the evolution and use of databases in other disciplines. We argue that applying a conventional database approach to modeling of geometric information is inadequate because it restricts users to fixed schemas for representation and viewing of data.

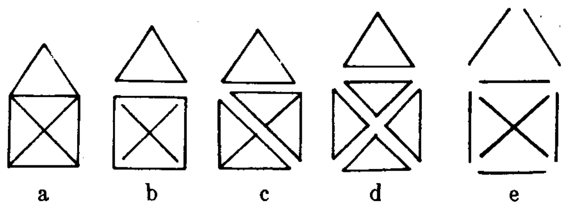

Figure 1. A House

CH2134-5/85/0000/0080$01.00 © 1985 IEEE

Geometric information is inherently multifaceted. For example, the geometric information in Figure 1.a can be interpreted as a triangle, a square, and a cross, as in Figure 1.b; three triangles and two line segments as in Figure 1.c; five triangles as in 1.d; or eight line segments as in 1.e; to name but a few. Furthermore, one interpretation of geometric information may be more appropriate than others for some applications. Consider the following two examples.

(1) A polygon can be viewed as a set of edges, each represented by the equation of a line segment, or as an ordered list of vertices, each represented by its coordinates. Some algorithms may prefer the edge-set view of polygons because it directly provides the coefficients of the equations defining their edges. On the other hand, an area clipping algorithm would prefer the vertex-list view of a polygon because it needs to maintain the area attributes (e.g., intensity, color, shading) of a polygon through clipping. The information about the vertices of a polygon and their traversal order is not explicitly available in the edge-set view.

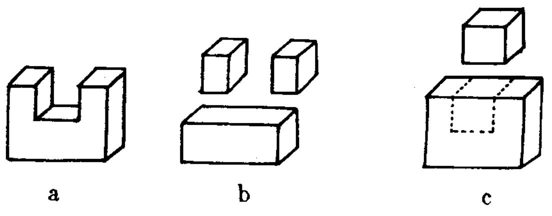

Figure 2. A U-Shaped Block

(2) The object in Figure 2.a can be viewed as the combination of three blocks as in Figure 2.b or as a block with a piece removed as in Figure 2.c. A design engineer who is interested in finite element analysis of such a model, may prefer the view in Figure 2.b. On the other hand, a manufacturer who is interested in the volumes that must be removed from stock material to fabricate this object may prefer the view in Figure 2.c.

Not only may there be multiple views of the same geometric entity, as the above examples demonstrate, but the number and nature of potentially useful interpretations of more complex geometric information is, in general, hard to predict. We contend that it is both highly desirable and conceivable to support the automatic, dynamic derivation of different interpretations of geometric information.

The traditional methods of geometric information processing use highly data dependent programs that deal with geometry mostly through direct manipulation and interpretation of numbers. The information processed by these programs is captured in rigid data structures or record-oriented databases with static relationships. Such models are incarnations of a single view of their information contents, generally tuned for a single application. Numerically-oriented computation also encourages decomposition of a problem into a multitude of narrowly specialized subproblems, each of which must be solved explicitly and separately. For example, finding a line given a sufficient number of its properties is partitioned into several isolated special cases, e.g., when it is tangent to two circles, when it connects two points, when its slope and intercept are given, etc. This breakdown spreads the knowledge about properties of shapes, relationships, and geometric rules across many pieces of programs and data structures. Consequently, existing geometric modelers have no explicit understanding of such fundamental concepts as coincidence, parallel, tangent, intersection, etc., and cannot support dynamic definition or derivation of new concepts that more directly apply to a new problem [2,3,8,9,14].

We propose a new paradigm for processing geometric information: geometric reasoning. The prime motivation for our proposal is to accommodate the inherent multifaceted nature of geometric information. A secondary motivation is to accommodate this dynamically, e.g., as users' views change. Significant features of geometric reasoning include its descriptive (definitional) versus prescriptive (operational) nature, the focus on information abstraction versus data representation, and the support for dynamic redefinition and processing of data.

Section 2 describes the basic formal concepts underlying our notion of geometric reasoning; Section 3 pursues further the aforementioned features of our approach; Section 4 contains numerous examples; Section 5 contrasts geometric reasoning with other trends in computing.

2. What is Geometric Reasoning?

Informally, geometric reasoning is the process of defining and deducing the properties of a geometric entity using the intrinsic properties of that entity, its relationships with other geometric entities, and the rules of inference that bind such properties together in a geometric (Euclidean) space. Thus, the two prime activities in geometric reasoning are *defining properties* and *defining classes of objects*. We perform both activities within the same formal system, which a user may extend using the same language as that of the formal system.

More formally, let there be a *universe* of objects. The formal system underlying geometric reasoning is based on first-order predicate calculus with equality on these objects (FOPCE). This FOPCE logical system includes: the usual logical symbols, i.e., negation, connectives, and quantifiers; the equality symbol (=); variable symbols; and well-formed formulae (wffs), logical axioms, and rules of inference (e.g., modus ponens) written in terms of the symbols. A user of a geometric reasoning system extends FOPCE with nonlogical symbols, in particular constant and predicate symbols, and wffs, axioms, and rules of inference written with these new symbols. *

This simple basis supports the activities of defining properties and defining classes of objects as follows. First, all properties of objects and relationships among objects are defined in terms of predicates on objects, which are written in terms of the nonlogical predicate symbols in a user-extended FOPCE. Note that by taking a uniform view toward properties and relationships, no function symbols of non-zero degree need ever be added to FOPCE by a user. The decision to allow users to define only predicates and not also functions is motivated by recognizing that the distinction between properties and relationships is not appropriate for geometric information. For example, a line segment as an object is just as well defined in terms of what may be considered its properties (e.g., end-points, or length, slope, and intercept), as in terms of its relationships with other objects (e.g., parallel to a line, tangent to a circle, and connecting two other lines).

Second, classes of objects are defined through instantiating free variables appearing in predicates. The set of all objects whose substitution for the same free variable of a given predicate make that predicate true is called a *class* of objects. Each constant symbol, e.g., the integer 5, appearing in a predicate is in its own class. We further define an *Inherit* relation over these classes. For classes, $C1$ and $C2$, $<C1, C2>$ is in *Inherit* if for all properties $P: C2 \times T1 \cdots \times Tn \rightarrow Bool$, for all $x: C2$, if $P(x, v1, \ldots, vn)$ holds for some $v1: T1, \ldots, vn: Tn$, then there exists some property $P': C1 \times T1 \cdots \times Tn \rightarrow Bool$ such that for all $y: C1$, $P'(y, w1, \ldots, wn)$ holds for some $w1: T1, \ldots, wn: Tn$. We require that *Inherit* be a partial-order. For convenience, P and P' are usually given the same name. We use the expressions "$C1$ is a *subclass* of $C2$" or "$C1$ *inherits all properties* of $C2$" if $<C1, C2>$ is in *Inherit*. From our partial-order restriction on *Inherit*, notice that (1) if $<C1, C2>$ and $<C2, C1>$ are both in *Inherit*, then $C1$ and $C2$ are the same class, and (2) multiple inheritance is allowed since $<C1, C2>$ and $<C1, C3>$ may both be in the *Inherit* relation where $C2 \neq C3$.

* Strictly speaking, it is the responsibility of the user to show that any axiom or rule added to FOPCE does not lead to an inconsistent deductive system. In practice, however, such an inconsistency may be discovered in the course of a "proof" in the system, thus notifying the user that an inconsistent axiom or unsound rule was added.

An *Inherit* relation induces a *class hierarchy* on the classes of objects. This hierarchy provides another way --a very convenient way-- of defining properties of objects: through inheritance for subclasses in terms of previously defined classes. That is, a user is freed from explicitly (and laboriously) introducing predicate symbols, axioms, and rules defining properties of objects in a subclass for those properties that would otherwise automatically be inherited from their parent classes through a hierarchy. For geometric objects, which "naturally" can be categorized into classes (e.g., points, lines, polygons), this convenience is extremely useful because of similarity among the kinds of properties they have (e.g., number of angles, number of equal sides).

3. Significant Features of Our Approach

In this section, we pursue the implications of three features of our approach: the descriptive versus prescriptive nature of geometric reasoning; the emphasis on abstraction versus representation; and the dynamic manner of viewing geometric information.

Geometric reasoning is descriptive, not prescriptive, in nature.** Properties of objects are either defined as axioms or derivable from the axioms and inference rules of a user-extended FOPCE. It is the capability of deduction in first-order logic that allows the user to infer properties of objects not explicitly stated in any given axiom. The axioms and rules allow users to acquire the information they need, independent of how it was given to or stored by the system. Consequently, users have access to both explicit and implicit information. This is important in geometry because in complex shapes, implicit information can be just as significant as explicitly defined information. For example, polygons not only result from explicitly defining polygon objects, but also from coincidence of the vertices of other polygons, and also from intersection of independent lines.

** This descriptive versus prescriptive contrast is similar to that seen in other fields, e.g., applicative versus imperative programming, definitional versus operational semantics.

Additionally, with deductive reasoning, users do not specify algorithms that prescribe how data items should be manipulated. Instead, through definitions and inference rules, users describe the desired properties of data and relationships they wish to see. As a consequence, users can do "proofs" about their data and possibly gain knowledge implicitly contained in their definitions and rules.

A second feature of our approach is that in reasoning about geometric objects in terms of their properties, we support information abstraction. Modern high-level programming languages, e.g., CLU [20], Mesa [21], Ada [1], that have linguistic support for data abstraction allow users to define abstract data types by defining a set of operations that can be performed on their values. The underlying representation is hidden from users of abstract data types. We also reason about geometric information at an abstract level, e.g., we reason about lines in terms of lines and not necessarily in terms of slopes and y-intercepts or end-points. Consequently, we are not bound to a single predetermined model of information. Instead, as many models as desired can be supported, which means no one particular model serves to bias any user's viewpoint of information.

What is unusual about our approach to information abstraction is our notion of class versus the more standard notion of "class" in the Smalltalk sense [11] or "type" in the Ada sense. For us, classes of objects are defined implicitly as side-effects of defining predicates on objects. In our approach, an object can possess any set of properties, independent of any class structure (e.g., class or type hierarchy), provided that its properties are consistent, i.e., the conjunction of the predicates defining its properties does not lead to a contradiction in the extended FOPCE.

Finally, geometric reasoning is a dynamic process in two senses. First, users can dynamically change the knowledge database of a geometric reasoning system by incrementally adding new axioms and rules to define properties and relationships and thus extend those built-in or previously added. Second, the addition of new information by users may change their previous view and thus some previously established properties and relationships may no longer hold or new properties and relationships may be (unexpectedly) deduced by the system.

Notice that because of our unusual notion of classes of objects, i.e., that classes are implicitly defined through satisfying predicates, we are not bound to a single, statically-defined class hierarchy, but allow for a set of class hierarchies, each of which supports a particular view. One user may describe the same geometric information in terms of a different set of predicates from another user. Each different set of predicates describing the same information imposes a possibly different class hierarchy. Thus, many hierarchies may simultaneously exist reflecting different views that different users may have of the same information, where the nature and number of these views may change over time. See Section 4.2 for examples of object class hierarchies.

4. Examples

In this section we give some simple examples of how one can use a geometric reasoning system. We distinguish between the deductive reasoning implemented in a system needed to support deduction in first-order logic, versus the reasoning performed by users in order to express their problems of interest. Our problem solving paradigm encompasses both kinds of reasoning, but our examples focus on the latter. Thus, we assume the existence of a system (complete with an inference mechanism and a user language) that supports geometric reasoning, but we stress that geometric reasoning itself is independent of any implementation.

In our examples, we use a Prolog-like[6] notation for defining predicates and stating queries. Definition of a predicate is akin to adding a rule of inference to FOPCE. The definition of a predicate is given by writing on the left-hand side of a ":-" symbol, a predicate name and a list of variables and constant symbols, and on the right-hand side, a wff involving the variables introduced on the left. Response to a query is akin to (implicit) definition of classes of objects. A *query* is a wff containing variables and constant symbols. Free variables appearing in a query can be instantiated by previously declared (named) objects, or by creating new objects that satisfy the properties required by the query. By convention, we use an initial lowercase letter (e.g., p and x) for variables, and an initial uppercase letter (e.g., *P1* and *C1*) for constants, e.g., previously declared objects.

4.1. Defining and Deducing Objects and Properties

Let us begin by considering a simple universe of points, lines, circles, numbers, and booleans. We assume numbers and booleans and predicates describing their properties are built-in. For points, we define predicates such as *abs(p, c)*, which is true if the coordinate c is the abscissa of the point p, and *ord(p, c)*, which is true if c is the ordinate of p. For example, *abs(P1, 2)* is true if the abscissa of the previously defined object (point) *P1* is 2, and since p is a free variable, *abs(p, 2)* requires any instantiation of p to be an object whose abscissa is 2. Given *abs* and *ord*, we can define equality of points in terms of their Cartesian coordinates:

equal(p1, p2) :- abs(p1, a) & ord(p1, o) &
abs(p2, a) & ord(p2, o)

We emphasize that no specific representation for points, e.g., the existence of an underlying Cartesian coordinate system, is necessarily assumed.

Additionally, we can define the following predicates on points, lines, and circles. (The names of the predicates were chosen to be indicative of their meaning so we leave off the right-hand sides of their definitions.) For all *p, p1, p2*: point, *l, l1, l2*: line (segment), *c*: circle, *n*: number, we have:

lines

slope(l, n)
y-intercept(l, p)
x-intercept(l, p)
end-points(l, p1, p2)
parallel(l1, l2)
length(l, n)
equal(l1, l2)

points and lines

coincide(p, l)
intersect(p, l1, l2)

points and circles

coincide(p, c)

points, lines, and circles

intersect(l, c, p1, p2)
tangent(l, c, p)

The next seven predicates are simple example queries. For each predicate, the system instantiates its free variables to all objects whose substitution in the predicate would make it true. Note that this process is *not* a search through previously defined objects to find those that satisfy the query (as in the case of a typical database query); objects are "created" by the system, if necessary, to satisfy a query.

1. *abs(p1, 2) & ord(p1, 3) & abs(p2, 3) & ord(p2, 6) & equal(p1, p2)*
   Here, the system uses the definition of equal on two points to deduce that this predicate is false, i.e., there are no instantiations of *p1* and *p2* that make them equal. To satisfy the predicate, *p1* and *p2* must be instantiated to two objects with the following properties: the abscissa and ordinate of *p1* must be 2 and 3, respectively (hence *p1* must be instantiated to a point), the abscissa and ordinate of *p2* must be 3 and 6, respectively (hence *p2* must be a point also), and *p1* and *p2* must be equal. The definition of *equal* states that two objects (points) are equal if they have the same abscissa and ordinate. Clearly, this contradicts the other properties of *p1* and *p2* stated above, therefore the predicate fails.

2. *abs(p1, 2) & ord(p1, 3) & equal(p1, p2) & abs(p2, x) & ord(p2, y)*
   This predicate is satisfied by (simultaneously) instantiating *p1* to a point with the given coordinates, *p2* to a point equal to *p1*, and *x* and *y* to the abscissa and ordinate of *p2*, 2 and 3, respectively.

3. *x-intercept(l, 4) & parallel(L1, l)*
   This predicate is satisfied by instantiating *l* to the line that intersects the x-axis at x = 4 and is parallel to the given line *L1*. This query corresponds to creating a line parallel to a given line, passing through the point with Cartesian coordinates (4,0).

4. *y-intercept(l, 5) & x-intercept(l, 4) & parallel(L1, l)*
   This predicate is satisfied if line *l* whose x- and y-intercepts are 4 and 5, happens to be parallel to line *L1*. This query corresponds to verifying whether a given line is parallel to the line passing through the points (0,5) and (4,0).

5. *coincides(P1, l) & coincides(P2, l)*
   This predicate is satisfied by instantiating *l* to the line that passes through the two given points *P1* and *P2*. If the two points are not distinct, i.e. *equal(P1, P2)* is true, then *l* is only partially defined. In this case *l* gets instantiated to a partially constrained variable which represents the family of lines that pass through the given common point. This query corresponds to constructing a line passing through two given points.

6. *coincides(p, L1) & coincides(p, L2)*
   This predicate is satisfied by instantiating *p* to the point that is the intersection of the two lines *L1* and *L2*, if such intersection exists. The above predicate is false if the two lines do not intersect. This query corresponds to finding (i.e., computing) the intersection point of two given lines.

7. *length(l, n) & tangent(l, C1, P1)*
   This predicate is satisfied by instantiating *n* to be the length of one of the tangents to a given circle from a given point. Since *l* can be instantiated to either of two tangents in order to satisfy the predicate, the response to the query would be both the (unique) distance and the set of two tangents that satisfy the predicate. If we were interested in only the length of a tangent, the fact that two lines happen to satisfy the predicate would be incidental knowledge to us.[†]
   However, we can further use this incidental knowledge and reason about the two tangents. For example, knowing that there are two tangents, we may want to define lines parallel to one of them.

The predicates used in these examples could be user-defined or built-in, depending on the sophistication of the implementation of a geometric reasoning system. In either case, predicates introduced by a user would be given in the same language as those built in the system. So for example, if neither of the *intersect* predicates were built-in, a user could define them as follows:

intersect(p, l1, l2) :- coincide(p, l1) & coincide(p, l2)
intersect(l, c, p1, p2) :- coincide(p1, l) & coincide(p2, l) &
coincide(p1, c) & coincide(p2, c) &
¬equal(p1, p2)

Recall the examples mentioned in Section 1. To reason about these examples, we need to extrapolate from the properties given about points, lines, and circles, and imagine having a richer set of defined properties. For the example of Figure 1, we would need such concepts as squares, triangles, ontop, and inside, and for the polygon example, we need concepts such as edge-set, vertex-list, and polygons.

† If we were originally interested in finding all lines tangent from *P1* to *C1*, we would need only state the predicate *tangent(l, C1, P1)* and the same set of lines would result as the answer. This merely points out that there may be more than one way to specify a problem in terms of the predicates defined.

For the example of Figure 2, we need to extrapolate further into three dimensions and imagine having definitions of blocks, solids, etc. Since we support multiple views of any geometric entity, all of the views of Figure 1 would be supported by our system. If one is not explicitly supported, then a user would be allowed to add enough definitions (e.g., predicates defining properties of a square and relationships between lines and squares) to add this view to the logical system. No explicit algorithms are required of users to define the transformation of one view to another.

Let us look at our edge-set and vertex-list polygon example more carefully. To support both views of a polygon, we provide the following predicates: *polygon(p, n)* is true if *p* is an *n*-sided polygon; *vertex-of(v, i, p)* is true if point *v* is the *ith* vertex of polygon *p* in some arbitrary (but fixed) order; *edge-of(e, p)* is true if line segment *e* is an edge of the polygon *p*. Thus, *vertex-of(v, 5, P)* instantiates *v* to the fifth vertex of a previously defined polygon *P* (if such a vertex exists), and *vertex-of(v, i, P)* returns every vertex of *P* together with its index. The predicate *vertex-of(V, i, P)* is true if *V* is any vertex of polygon *P*, in which case *i* will be instantiated to its index. The predicate *edge-of(L,P)* is true if *L* is a line segment and it is an edge of the polygon *P*. On the other hand, *edge-of(e, P)* instantiates *e* to the members of the set of edges of the polygon *P*, in no particular order.

The predicates *edge-of* and *vertex-of* are flexible enough to support the edge-set and the vertex-list views of polygons, respectively. In particular, if one is not provided by the system, it can be defined by a user, using the other one. For example, *edge-of* can be defined as:

edge-of(e, p) :- end-points(e, p1, p2) &
vertex(p1, i, p) & vertex(p2, j, p) &
polygon(p, n) & next-to(i, j, n)

where *next-to(i, j, n)* is true if either *i* or *j* follows the other in the circular list of integers 1 to *n*. This rule states that *e* is an edge of *p* if *e* is an object (line segment) with end-points *p1* and *p2*, *p1* is the *ith* vertex of (polygon) *p*, *p2* is the *jth* vertex of *p*, *p* is an *n*-sided polygon, and *i* and *j* are next to each other in an *n*-cycle -- an intuitively clear set of conditions for a line segment to be an edge of a polygon.

To define *vertex-of* given *edge-of*, we must decide on a vertex traversal order and some means of "marking" the first vertex of each polygon.[††] Suppose we desire the traversal order to be clockwise, starting from a distinguished vertex for each polygon. We can assume the predicate *distinguished(v, p)* is defined such that it is true if *v* is the distinguished (i.e., "marked") vertex of polygon *p*. We define *clockwise(p1, p2, p3)* to be true for three arbitrary points *p1*, *p2*, and *p3*, if the traversal from *p1* to *p2* to *p3* is in clockwise direction. The predicate *vertex-of* can be defined as:

```
vertex-of(v, 1, p) :- distinguished(v, p)
vertex-of(v, i, p) :- edge-of(e1, p) & end-points(e1, x, v) &
                      edge-of(e2, p) & end-points(e2, v, y) &
                      clockwise(x, v, y) & vertex(x, j, p) &
                      sum(j, 1, i)
```

where *sum(k, l, m)* is true for integers *k*, *l*, and *m* if $k+l = m$. The first predicate in the definition of *vertex-of* simply states that *v* is the vertex with index 1 for polygon *p* if it is the distinguished vertex of *p*. The second predicate states that if *x*, *v*, and *y* are the end-points of two edges of polygon *p* connected at *v*, and *v* follows *x* in clockwise direction, then the index of vertex *v* is one higher than the index of vertex *x*.

4.2. Class Hierarchies

To illustrate the notions of a class hierarchy, multiple class hierarchies, and multiple inheritance, consider how one might use a class hierarchy to define properties of different kinds of closely related polygons. Figure 3 depicts a hierarchy for polygons with two immediate subclasses, triangle and quadrilateral. Triangle has two immediate subclasses, scalene and isosceles, and isosceles has one subclass, equilateral. Quadrilateral has a chain of subclasses, from trapezoid to square. If a property *sides(p, n)* is defined for polygons (to stand for the number of sides of a polygon), then the triangle (quadrilateral) subclass would inherit the *sides* predicate and redefine it such that for all triangles (quadrilaterals), *p*, *sides(p, 3)* (*sides(p, 4)*) holds. Notice that the *sides* property is inherited all the way down to the square subclass, i.e., all squares have 4 sides.

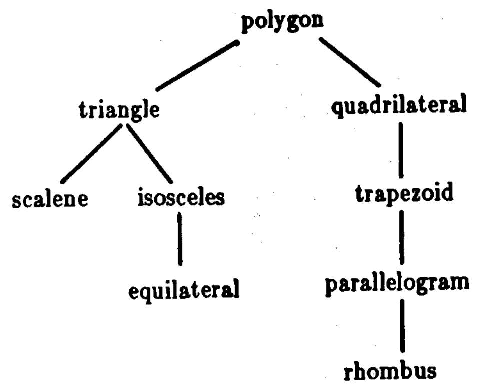

Figure 3. Polygon Class Hierarchy 1

As mentioned in Section 3, we do not impose a fixed hierarchy on classes of objects. A different user may very well have defined a different set of properties of polygons, triangles, and quadrilaterals such that the class hierarchy depicted in Figure 4 results. Here, triangle has three immediate subclasses, right-angle, isosceles, and equilateral, whereas in Figure 3 it has only two. The class hierarchy rooted at the quadrilateral class is considerably more complicated in Figure 4 than in Figure 3. To illustrate multiple inheritance, consider the rectangle class. Rectangle is a subclass of two classes, two-right-angles and parallelogram. If the *sides* property is defined (and inherited) as before for polygons, triangles, and quadrilaterals, then again, rectangle inherits the *sides* property. It also inherits the properties of both its immediate parent classes, e.g., *has-two-right-angles* from the two-right-angles

[††] For example, we could decide that the first vertex of every polygon is the one that is closest to the origin of a Cartesian coordinate system, the one with the smallest angle to the x-axis if several vertices are at the same distance from the origin.

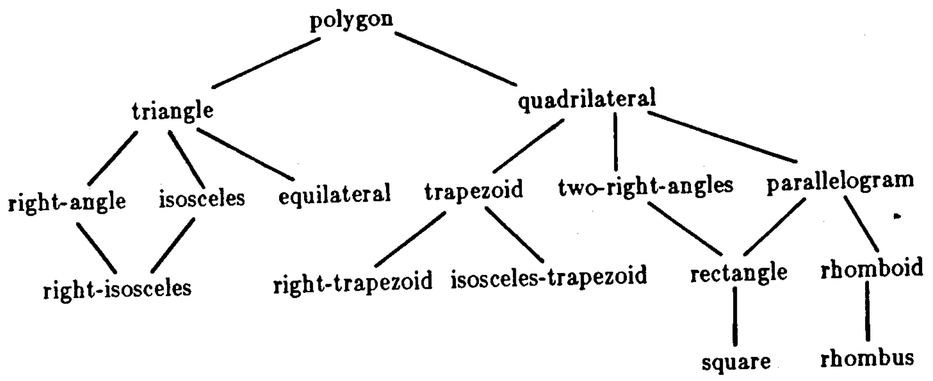

Figure 4. Polygon Class Hierarchy 2

class and *has-two-pairs-of-parallel-sides* from the parallelogram class. An additional property defined for objects in the square subclass of the rectangle class would be *all-equal-sides* (to stand for whether all sides of a quadrilateral are of equal length). Notice that objects of the rhombus class would also have a *all-equal-sides* property, but would not have the *has-two-right-angles* property, thus distinguishing rhombuses from squares.

5. Relationship to Other Trends

Geometric reasoning can be seen as a focal point of many related trends: deductive databases, expert systems, logic programming, object-oriented programming, and executable specifications. Two main differences between geometric reasoning and other trends is that we are interested in a specific, but rich, domain of discourse, namely geometry, and that this domain has a universal, formal semantics.

We believe geometry is an interesting domain for investigation in all above areas because the semantics of concepts in this domain are well-defined. For instance, objects such as points, lines, and planes, and relationships like intersects, tangent-to, parallel-to, etc., all have properly defined mathematical meanings. Therefore, there is no ambiguity in what "line L is tangent to circle C" can mean. In contrast, the semantics of such concepts as "employee," "manager," "department," and "is-employed-by" in a typical databases are often ambiguous and highly dependent on the application, query at hand, or a user's intuition.

Previous work on using databases for processing of geometric information has concentrated on "implementing" geometric models on top of conventional record-oriented databases [18,19,15,25]. We argue that because of the rigidity of their schemas [16,13] conventional data models are poor mechanisms for representing the rich semantics of geometric information. Recent object-oriented semantic data models [24,12,22], which have not yet been seriously considered for geometric modeling, are also inappropriate for the domain of geometry because of some of their underlying assumptions, e.g., the static distinction between types and instances and the primacy of schema. Deductive databases, a more recent trend in logic programming and database research, do not impose such restrictive assumptions and agree with our motivation for geometric reasoning [7,17]. Like deductive databases, we are interested in using deductive reasoning to derive non-explicit information that is collectively implied by other pieces of information in a database. Unlike generalized deductive databases, however, we deal with a specific domain of discourse.

Like expert systems, we are interested in specializing in a particular problem domain. Typical expert systems, however, deal with problem domains whose concepts are not (and cannot be) formally well-defined and rely on heuristic rules. In contrast, the well-defined semantics of our geometric domain can be expressed in the form of algebraic equations. A geometric reasoning system can then translate geometric concepts into a system of simultaneous equations. Such systems of equations can then be manipulated by an equation solver to verify their consistency or to derive the values of their unbound variables.

We borrow ideas from two trends in programming languages, object-oriented programming and logic programming, in the sense that we are interested in identifying and manipulating (geometric) objects and reasoning about them in a formal logical system. Also, we observe that the definitional, yet processable, nature of geometric reasoning is related to the trend of executable specifications, where formal specifications of programs can be either symbolically tested or run through a theorem prover.

Finally, there is little similarity between geometric reasoning and classical geometric theorem-proving [10]. We are not interested in proving theorems in general or in Euclidean geometry, but in deriving properties of specific geometric entities.

References

[1] "The Programming Language Ada Reference Manual," Lecture Notes in Computer Science 106, 1981.

[2] J. W. Boyse and J. E. Gilchrist, "GMSolid: Interactive Modeling for Design and Analysis of Solids," *Computer Graphics and Applications* **2**, 2, March 1982.

[3] C. M. Brown, "PADL-2: A Technical Summary," *Computer Graphics and Applications* **2**, 2, March 1982.

[4] M. L. Brodie, "On the Development of Data Models," in *On Conceptual Modelling*, Springer-Verlag, 1984.

[5] K. L. Clark and S. A. Tarnlund, *Logic Programming*, Academic Press, 1982.

[6] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[7] V. Dahl, "On Database Systems Developed Through Logic," *ACM TODS* **7**, 1, March 1982.

[8] C. M. Eastman and M. Henrion, "The GLIDE Language for CAD," *Journal of the Technical Councils of ASCE*, August 1980.

[9] W. Fitzgerald, F. Gracer, and R. Wolfe, "GRIN: Interactive Graphics for Modeling Solids," *IBM J. Res. Dev.*, **25**, 4, July 1981.

[10] H. Gelernter, "Realization of a Geometry-Theorem Proving Machine," *Automation of Reasoning*, **1**, Siekmann, J. and Wrightson, G. Springer-Verlag, 1983.

[11] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[12] M. Hammer and D. McLeod, "The Semantic Data Model: A Modeling Mechanism for Database Applications," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, June 1978.

[13] M. Hammer and D. McLeod, "Database Description with SDM: A Semantic Database Model," *ACM Trans. on Database Systems*, **6**, 3, September 1981.

[14] R. Hillyard, "The Build Group of Solid Modelers," *Computer Graphics and Applications*, **2**, 2, March 1982.

[15] H. R. Johnson, J. E. Schweitzer, and E. R. Warkentine, "A DBMS Facility for Handling Structured Engineering Entities," *Proc. ACM-IEEE Database Week*, May 1983.

[16] W. Kent, "Limitations of Record-Oriented Information Models," *ACM Trans. on Database Systems*, March 1979.

[17] R. Kowalski, "Logic as a Database Language," Tech. Rep. Imperial College, July 1981.

[18] Y. C. Lee and K. S. Fu, "A CSG Based DBMS for CAD/CAM and its Supporting Query Language," *Proc. ACM-IEEE Database Week*, May 1983.

[19] Y. C. Lee and K. S. Fu, "Integration of Solid Modeling and Data Base Management for CAD/CAM," *Proc. 20th Design Automation Conf.*, June 1983.

[20] B. H. Liskov et al., "CLU Reference Manual," MIT/LCS/TR-225, October 1979.

[21] J. G. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual," Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

[22] J. Mylopoulos and H. K. T. Wong "Some Features of the TAXIS Data Model," *Proc. 6th Int. Conf. Very Large Data Bases*, 1975.

[23] A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *Computing Surveys*, **12**, 4, December 1980.

[24] J. M. Smith and D. C. P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Trans. on Database Systems*, **2**, 2, June 1977.

[25] D. Weller and R. Williams, "Graphic and Relational Data Base Support for Problem Solving," *Computer Graphics*, **10**, 2, 1976.