An Operation-Independent Approach to Extend 2D Spatial Operations to 3D and Moving Objects

Farid Karimipour
Institute for Geoinformation and Cartography, Vienna University of Technology
Gusshausstr. 27-29, A-1040 Vienna, Austria
(+43 1) 58801-12711
karimipour@geoinfo.tuwien.ac.at

Andrew U. Frank
Institute for Geoinformation and Cartography, Vienna University of Technology
Gusshausstr. 27-29, A-1040 Vienna, Austria
(+43 1) 58801-12711
frank@geoinfo.tuwien.ac.at

Mahmoud R. Delavar
Deprtment of Surveying and Geomatics Engineering, College of Engineering, University of Tehran
North Kargar St., Tehran, Iran
(+98 21) 88008839
mdelavar@ut.ac.ir

ABSTRACT
It has been pointed out repeatedly that spatial operations must be extended to include support for 3D and moving objects. The attempt to code by hand each spatial operation for each data type (e.g., static 2D, moving 2D, static 3D, and moving 3D) is forbidding and has led to specific solutions for particular purposes. In this paper, we have advocated an operation-independent approach to extend 2D spatial operations to 3D and moving objects. The approach is based on implementation of the concepts of *n*-dimensional geometry through definition of transformations between domains called "*lifting*". It is explained via some sample spatial operations and then the implementation results for convex hull computation are represented.

Categories and Subject Descriptors
F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages – *Algebraic language theory*

General Terms
Algorithms, Languages, Theory

Keywords
Spatial operations, 3D GIS, Moving Objects, Functor, Lifting

1. INTRODUCTION
The need to extend GIS software to treat 3D and moving objects is widely recognized (for example: [1, 2, 3, 11, 12, 13, 15, 17]). The attempt that needs recoding of each spatial operation for each data type is forbidding and the code for a general 2D, 3D, and moving objects supporting GIS is nearly four times the current code size, offering four variants: static 2D, moving 2D, static 3D, and moving 3D. The complexity of such a growth of code written in one of the currently popular programming languages is hard to manage, resulting in numerous bugs. We believe the sheer size of the task explains why no commercial GIS has a comprehensive offer for treatment of 3D and moving objects.

We have advocated a principled method to extend 2D spatial operations to new data types, e.g., 3D and moving objects, with a minimum amount of recoding [3]. The approach is based on implementation of the concepts of *n*-dimensional geometry using *functors* that transform one domain to another. It is possible to define *lifting* functors that extend the applicability of code to new data types— automatically [5, 6, 7, 8]. This sounds like magic, but is (just) the application of a sound mathematical type theory [16] implemented in modern functional programming languages, e.g., Haskell [18, 19].

In this paper, we show how this is done. The proposed approach is explained through a very simple operation: computation of distance between two points. This operation will be extended to support static 2D, moving 2D, static 3D, and moving 3D points, which is called *generalization* henceforth in this paper. Moreover, the idea is applied to generalize the ordering test, which tests if a point is on the right or left of a line, plane, etc. This generalized test is used for general point in polygon test and also for convex hull computation. We show in this paper how these operations can be generalized such that the same code applies to static or moving *n*-dimensional points. The focus is on minimizing the amount of new code necessary and not on performance: it seems better to have a working program, even if it is slow, than to wait till somebody writes a fast program some time in the future (remember Moor's law: computer speed doubles every 18 months on average!).

Section 2 presents the generalized convex hull computation as a motivation example. Section 3 briefly describes functors as the concept behind the so-called *lifting*, i.e., extension of operations from one domain (e.g., 2D) to another (e.g., 3D or moving). In Section 4, the proposed approach to extend 2D spatial operations to 3D and moving objects is explained through a simple example, i.e., computation of distance between two points. The materials developed here are used in Sections 5 to test if a static/moving point is on the right or left of a static/moving line, plane, etc. It leads to point in polygon test and convex hull computation for static or moving *n*-dimensional points, which is shown in Section 6 and their implantation results are represented in Section 7.

ACM GIS'08, November 5–7, 2008, Irvine, CA, USA.

Finally, section 8 contains some conclusions and ideas for future works.

2. MOTIVATION EXAMPLE

Suppose we have a set of 2D points (e.g., police cars) and a specific target point (e.g., a target car). To answer the question whether the target point is inside the area delimited by the point set is possible by using convex hull computation (See section 6). As Figure 1 shows, if the target point is outside, then it is a part of the boundary of the convex hull of all points.

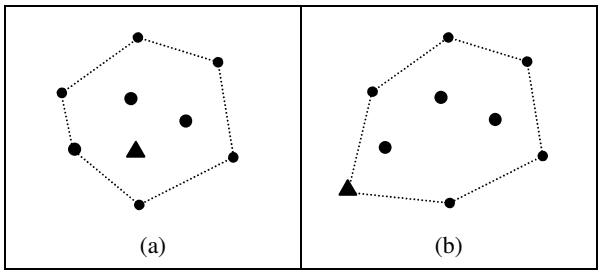

Figure 1. Using convex hull to test if a target point (the triangle) is inside the area delimited by a set of points (the circles): (a) inside (b) outside.

If the point set, the target point or both are moving, then the problem is computing the convex hull of some moving points. Extension of the problem to 3D will test if a static or moving 3D point is inside or outside the volume constructed by a set of static or moving 3D points, which again needs convex hull computation, but this time for some static or moving 3D points.

The straightforward approach to implement this test to support 2D, 3D and moving points is first implementation of one of the existing algorithms for convex hull computation of static 2D points [1]. Then, its extension to moving 2D points may be reached as follows:

- If a moving point crosses an edge of the convex hull (called extreme edge), that edge splits into two new edges (Figure 2a);
- If the angle between two successive edges that pass through a point becomes greater than 180°, then the two edges merge to one edge going through their non-common points (Figure 2b);
- If none of the above cases occur, the edges of the convex hull do not change (Figure 2c).

Note that in all of the three cases, location of the moving points must be updated.

To support static 3D points, some modifications in the algorithm used for static 2D points are required in such a way that it produces the extreme triangles, instead of extreme edges. Finally, modification of the explained approach for moving 2D points will construct the convex hull of some moving 3D points.

Following the above steps, we have four implementations of the convex hull computation for different types of points. However, there are two drawbacks in this approach:

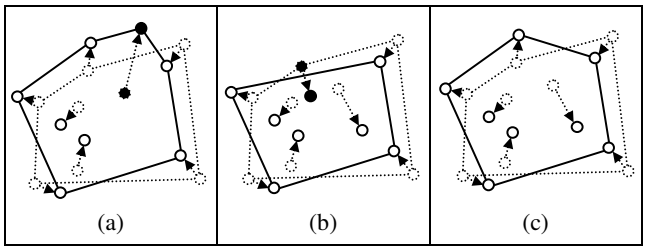

Figure 2. Updating the convex hull of a set of moving points: (a) the black point causes splitting an edge (b) the black point causes merging two edges (c) no changes in the edges.

- **Recoding:** Extension of convex hull computation to each new type of points requires recoding the whole procedure. Thus, the code for a general convex hull computation is four times the current code size for computing the convex hull of static 2D points.
- **Operation-dependency:** The technique explained to extend computation of convex hull of static points to moving points has least effect on the performance of the computations. It reduces the number of cases that must be checked against required changes in the extreme edges. However, it depends on the definition of convex hull. Therefore, extension of another operation, say, Delauney Triangulation to support moving points needs another specialized technique [11].

Considering the number of spatial operations, such approach that focuses on the performance and suggests specialized methods for extension of each spatial operation, seems not promising to achieve 3D and moving counterparts of all of the already implemented 2D spatial operations in the near future.

In the following sections, we introduce an operation-independent approach to extend 2D spatial operations to 3D and moving objects with minimum amount of recoding. The approach implements the concepts of *n*-dimensional geometry using *functors* that transform one domain to another. We define appropriate *lifting* functors to lift a domain (e.g., static 2D) to another domain (e.g., moving 2D). These transformations will extend all operations from the first domain to the second in the same way and without recoding.

3. FUNCTORS LIFT DOMAINS

A collection of algebraic systems and their morphisms, e.g., 2D points and customary operations between them, are called a *category* [10, 14]. Functors are a *principled* way to extend a given algebraic system from one domain to another. As shown in Figure 3, "A *functor F* from [category] *A* to [category] *B* assigns to any object A_k in *A* an object $F(A_k)=B_k$ in *B* and to every operation $f_k:A_i \rightarrow A_j$ [in *A*] an operation $F(f_k):F(A_i) \rightarrow F(A_j)$ [in *B*] such that identity and composition are preserved, which says a functor is a mapping of categories" ([14], p. 131). The fundamental laws are:

$$F(f_k(A_i)) = g_k(B_i) \text{ where } g_k = F(f_k) \text{ and } B_i = F(A_i)$$
$$F(1_{Ai}) = 1_{F(Ai)} \quad (1)$$
$$F(f_k \,.\, f_t) = F(f_k) \,.\, F(f_t).$$

(Note: "." means function composition, i.e., applying the first function to the result of applying the second: $(f.g)(x) = f(g(x))$.

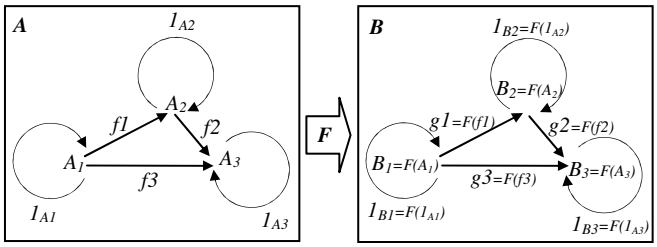

Figure 3. A Functor *F* from category *A* to category *B*

The approach of the paper to extend 2D spatial operations to 3D and moving objects is considering different desired domains in GIS (i.e., static 2D, moving 2D, static 3D and moving 3D) as categories with the same structures. Then, definition of appropriate functors, called *lifting* here, between them will extend all operations from one domain to another. Functors are functions that take another function(s) as argument(s). Such functions are called second order functions and can be treated in functional programming languages, e.g., Haskell [18, 19].

4. LIFTING SPATIAL OPERATIONS TO 3D AND MOVING OBJECTS

In this section, we show how to use the *lifting* functors to extend 2D spatial operations to 3D and moving objects. The example used here is computation of the Euclidean distance between different types of points. Table 1 shows the definition of static and moving 2D/3D points as well as the square distance between two points for each type.

Table 1. Definition of static and moving 2D/3D points and square distance for each type

Point type	Point	Square distance
Static 2D	(x, y)	$(x_2\text{-}x_1)^2+(y_2\text{-}y_1)^2$
Moving 2D	$(x(t), y(t))$	$(x_2(t)\text{-}x_1(t))^2+(y_2(t)\text{-}y_1(t))^2$
Static 3D	(x, y, z)	$(x_2\text{-}x_1)^2+(y_2\text{-}y_1)^2+(z_2\text{-}z_1)^2$
Moving 3D	$(x(t), y(t), z(t))$	$(x_2(t)\text{-}x_1(t))^2+(y_2(t)\text{-}y_1(t))^2+(z_2(t)\text{-}z_1(t))^2$

The first step toward generalization is development of a unified representation of points of any dimension. The *List* data type is a solution: A static or moving *n*-dimensional point in Euclidean space can be represented as a list of numbers as $[e_1, e_2, ..., e_n]$ where e_i is either a constant or time-dependent value. Then, $\sqrt{\sum (e_{i1}-e_{i2})^2}$ is a general definition of distance between two static or moving *n*-dimensional points. As Equation 2 shows, it applies *subtract* to the points, which are defined as two lists, pairwise (also called "pointwise" applications [14]), then applies *square* to each element in the list and finally sums up all of the elements in the list and gets its square root.

$$
\begin{aligned}
&p_1 = [e_{11}, e_{21}, \ldots, e_{n1}],\ p_2 = [e_{12}, e_{22}, \ldots, e_{n2}]\\
&p_1\text{-}p_2 = [e_{11}\text{-}e_{12}, e_{21}\text{-}e_{22}, \ldots, e_{n1}\text{-}e_{n2}]\\
&(p_1\text{-}p_2)^2 = [(e_{11}\text{-}e_{12})^2, (e_{21}\text{-}e_{22})^2, \ldots, (e_{n1}\text{-}e_{n2})^2]\\
&(d\ (p_1, p_2))^2 = (e_{11}\text{-}e_{12})^2 + (e_{21}\text{-}e_{22})^2 + \ldots + (e_{n1}\text{-}e_{n2})^2\\
&d\ (p_1, p_2) = sqrt\ ((e_{11}\text{-}e_{12})^2 + (e_{21}\text{-}e_{22})^2 + \ldots + (e_{n1}\text{-}e_{n2})^2).
\end{aligned}
\tag{2}
$$

To implement this general definition of distance between two points, we need to extend the primitive operations (e.g., +, -, *square*, etc.) such that they applies to a list of numbers. To support moving points, they must be extended to be applicable on time-dependent values, as well. Development of the two extensions using *lifting* functors are described in sections 4.1 and 4.2, respectively.

4.1 Extension to 3D

The key point of this extension is definition of an *n*-dimensional point as a list of numbers:

$$Point = [num, num, ..., num]. \tag{3}$$

Then, a general solution to extend operations on numbers to operations on points is to declare a functor which applies an operation to each element of a list (*liftD1*), pairs of elements of two lists (*liftD2*), and so on:

$$
\begin{aligned}
&liftD1\ (f\ ([e_1, e_2, \ldots, e_n])) = [f(e_1), f(e_2), \ldots, f(e_n)]\\
&liftD2\ (f\ ([e_{11}, e_{21}, \ldots, e_{n1}], [e_{12}, e_{22}, \ldots, e_{n2}])) =\\
&[f(e_{11}, e_{12}), f(e_{21}, e_{22}), \ldots, f(e_{n1}, e_{n2})].
\end{aligned}
\tag{4}
$$

Thus, the operations on points are defined as lifted versions of original operations. For example:

$$
\begin{aligned}
&square = liftD1\ (square)\\
&(+) = liftD2\ (+)\\
&(-) = liftD2\ (-).
\end{aligned}
\tag{5}
$$

Having lifted all of the required primitive operations, we can implement a general distance function: If p_1 and p_2 are two *n*-dimensional points, the distance between them is:

$$
\begin{aligned}
&distance\ (p_1, p_2) = sqrt.sum.square\ (p_1\text{-}p_2)\\
&\quad \text{where } sum\ [a_1, a_2, \ldots, a_n] = a_1+a_2+\ldots+a_n.
\end{aligned}
\tag{6}
$$

4.2 Extension to moving points

Moving points are a prototypical case of temporal data [20]. A moving point has a different position for any given time; it is modeled – if the language is second order and permits this – as a function from time to point:

$$MovingPoint = t \rightarrow Point. \tag{7}$$

It is convenient to define a type *Instant* as Floating number, and a general type *Changing* value, of which a moving point is just a particular case:

$$
\begin{aligned}
&Instant = Float\\
&Changing\ v = Instant \rightarrow v\\
&MovingPoint = Changing\ (Point).
\end{aligned}
\tag{8}
$$

To extend an operation of static points to moving points, all of its arguments must become functions of time. *liftT1*, *liftT2* are used for operations with one and two arguments, respectively:

$$
\begin{aligned}
&liftT1\ (f\ (a)) = f\ (a(t))\\
&liftT2\ (f\ (a, b)) = f\ (a(t), b(t)).
\end{aligned}
\tag{9}
$$

Thus, the operations on moving points are defined as lifted versions of original operations. For example:

$$\begin{aligned} &square = liftT1\ (square) \\ &(+) = liftT2\ (+) \\ &(-) = liftT2\ (-). \end{aligned} \quad (10)$$

Even the definition of distance between two moving points is achieved by *liftT2* functor:

$$distance = liftT2\ (distance). \quad (11)$$

The integration of *liftD*s and *liftT*s functors will provide us with operations that can be applied to static or moving *n*-dimensional points.

4.3 Example: distance between two points

This subsection shows how the operation *distance* works for different type of points. Static points are defined by their coordinates in the Cartesian coordinate system:

$$\begin{aligned} &s2Dpt1 = [2, 3] \\ &s2Dpt2 = [4, 5] \\ &s3Dpt1 = [2, 4, 1] \\ &s3Dpt2 = [4, 3, 2]. \end{aligned} \quad (12)$$

Function *distance* calculates the distance between these static points—independent of their dimension:

$$\begin{aligned} &sDist2D = distance\ (s2Dpt1, s2Dpt2) \rightarrow 2.82 \\ &sDist3D = distance\ (s3Dpt1, s3Dpt2) \rightarrow 2.44. \end{aligned} \quad (13)$$

How can one enter moving points? For this example, we define them as continuous functions. In practice, however, moving points are given by observed time-position elements between which positions are interpolated:

$$\begin{aligned} &m2Dpt1\ t = [(3t+1), (2t-1)] \\ &m2Dpt2\ t = [(2t+3), (7-2t)] \\ &m3Dpt1\ t = [(4t-3), (5t+2), (3t-4)] \\ &m3Dpt2\ t = [(3t-2), (2t+3), (5t-3)] \end{aligned} \quad (14)$$

To print such moving points, they must be given a time instant and then they reduce to ordinary, printable points. For example the position of above moving points at time instant 3 are as follows:

$$\begin{aligned} &m2Dpt1\ (3) = [10, 5] \\ &m2Dpt2\ (3) = [9, 1] \\ &m3Dpt1\ (3) = [9, 17, 5] \\ &m3Dpt2\ (3) = [7, 9, 12] \end{aligned} \quad (15)$$

Again, function *distance* calculates the distance between these moving points:

$$\begin{aligned} &mDist2D = distance\ (m2Dpt1, m2Dpt2) \\ &mDist3D = distance\ (m3Dpt1, m3Dpt2). \end{aligned} \quad (16)$$

In this case the distance depends on the time at which the points were observed; it is a changing value, i.e., a function of time. For example these distances at time instant 3 are reached as follows:

$$\begin{aligned} &mDist2D\ (3) = 4.12 \\ &mDist3D\ (3) = 10.81. \end{aligned} \quad (17)$$

5. ORDER OF POINTS

The test whether three 2D points are in *cw* (clockwise) or *ccw* (counterclockwise) order is called *ordering* or *ccw* test and it is often used in geometric algorithms [9]. The extension of this test to 3D points checks whether a point is on the right or left side of a plane goes through three points. The ordering test is implemented as calculation of determinants, i.e., area or volume calculation and then comparison with zero:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix} \text{ for 2D and } \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \text{ for 3D points} \quad (18)$$

Generally, for $n+1$ number of n-dimensional points, the ordering test is based on the sign of the following determinant:

$$\begin{bmatrix} e_{11} & e_{12} & e_{11} & e_{14} & \dots & e_{1(n+1)} \\ e_{21} & e_{22} & e_{23} & e_{24} & \dots & e_{2(n+1)} \\ e_{31} & e_{32} & e_{33} & e_{34} & \dots & e_{3(n+1)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ e_{n1} & e_{n2} & e_{n3} & e_{n4} & \dots & e_{n(n+1)} \\ 1 & 1 & 1 & 1 & \dots & 1 \end{bmatrix} \quad (19)$$

where e_{ij} is i^{th} element of the j^{th} point. Then, for n-dimensional points, to test if a point is on right or left of n number of points is calculated as follows:

$$\begin{aligned} &right\ (p, [p_1, \dots, p_n]) = det\ (tr\ [p_1, \dots, p_n, p]) \leq 0 \\ &left\ \ (p, [p_1, \dots, p_n]) = det\ (tr\ [p_1, \dots, p_n, p]) > 0 \\ &\quad \text{where } tr\ [e_1, \dots, e_n] = [e_1, \dots, e_n, 1]. \end{aligned} \quad (20)$$

These tests work immediately for n-dimensional moving points, without new code, using *lifting* functors presented in section 4.2:

$$\begin{aligned} &right = lift2T\ (right) \\ &left = lift2T\ (left). \end{aligned} \quad (21)$$

6. POINT IN POLYGON TEST AND CONVEX HULL COMPUTATION

Having implemented the ordering test, the test if a point is inside a polygon is straightforward: A point is inside a polygon, whose nodes are in clockwise order, if it is located on the right side of all edges of the polygon. If a polygon is defined by its vertexes as $[[p_1, p_2], [p_2, p_3], \dots, [p_n, p_1]]$, then:

$$\begin{aligned} &PointInPolygon\ (p, [[p_1, p_2], [p_2, p_3], \dots, [p_n, p_1]]) = \\ &allTrue\ (right\ (p, [[p_1, p_2], [p_2, p_3], \dots, [p_n, p_1]])) \\ &\quad \text{where } allTrue\ [b_1, b_2, \dots, b_n] = (b_1 == true) \text{ and} \\ &\quad (b_2 == true) \text{ and } \dots \text{ and } (b_n == true). \end{aligned} \quad (22)$$

The *right* and *left* tests are defined generally for *n*-dimensional points. Therefore, this definition of point in polygon test works for *n*-dimensional points.

We can go further and use the above operations for convex hull computation. The convex hull of a set of points is defined as the smallest set that contains the points [1]. Figure 4 illustrates the convex hulls for some 2D and 3D points.

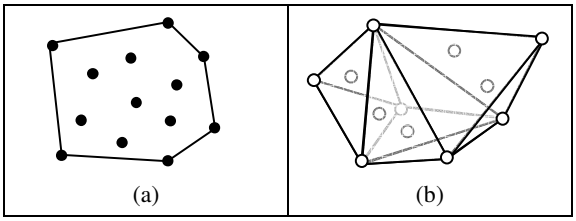

Figure 4. Convexhull for some (a) 2D points (b) 3D points

The algorithm we have used here to construct the convex hull is an incremental algorithm called *IncrementalConvexhull*. Figure 5 illustrates the key concept of the algorithm: When a point is inserted in the convex hull of a set of points, if it is inside the convex hull, no change is needed (Figures 5a and 5c). If it is outside, however, its opposite edges (for 2D) or triangles (for 3D) are replaced by new edges (for 2D) or triangles (for 3D) passing through the new point (Figures 5b and 5d).

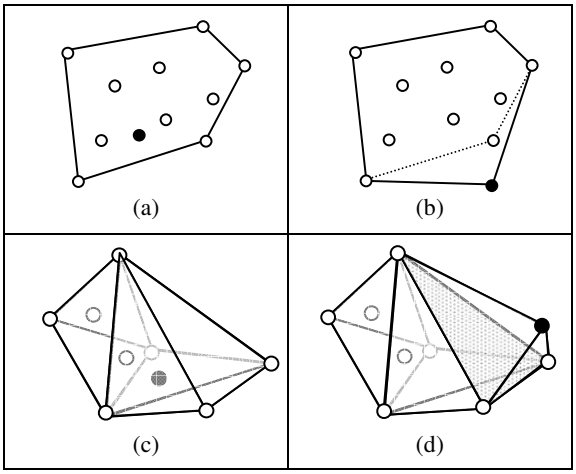

Figure 5. *IncrementalConvexhull* algorithm for 2D and 3D points: the newly added black point is inserted (a) inside the 2D convex hull (b) outside the 2D convex hull (c) inside the 3D convex hull (d) outside the 3D convex hull. In Figures (b) and (d), the removed parts are indicated by dotted line and plane.

Since the point in polygon test was defined generally for *n*-dimensional points, therefore this definition of convex hull works for *n*-dimensional points. Finally, point in polygon test and convex hull computation are extended to moving points using the appropriate *lifting* functors:

$$PointInPolygon = liftT2\ (PointInPolygon)$$
$$ConvexHull = lifT1\ (ConvexHull). \quad (23)$$

7. IMPLEMENTATION

The data types, operations and *lifting* functors introduced in sections 4 to 6 were implemented in Haskell programming language [18, 19]. It is a functional programming language that can handle functional arguments, which is the case here for *lifting* functors. Its syntax is so similar to mathematical notations. For example, translation of Equation 22 to Haskell notation is as follow:

$$PointInPolygon\ pt\ pl = all\ (==true)\ ((map\ right\ pt)\ pl) \quad (24)$$

s2Dpts and *s3Dpts* are two lists of sample static 2D and 3D points introduced like the points of Equation 12. Figure 6 shows the results of *ConvexHull (s2Dpts)* and *ConvexHull (s2Dpts)*.

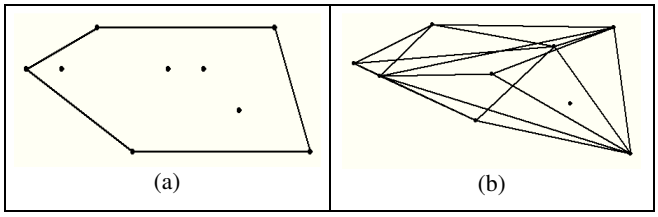

Figure 6. Results of convex hull computation for some sample static points: (a) static 2D (b) static 3D

Some moving 2D and 3D points (*m2Dpts* and *m3Dpts*, respectively) introduced like the points of Equation 13. *ConvexHull (m2Dpts)* and *ConvexHull (m2Dpts)* are functions of time whose results at instants 3 and 10 are shown in Figure 7.

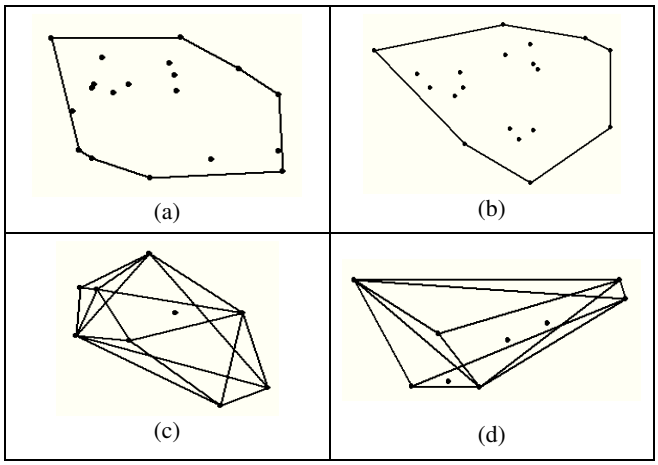

Figure 7. Results of convex hull computation for some sample moving points: (a) moving 2D points at time instant 3 (b) moving 2D points at time instant 10 (c) moving 3D points at time instant 3 (d) moving 3D points at time instant 10

8. CONCLUSIONS AND FUTURE WORKS

The goal here is to show how a minimal amount of new coding allows the use of available solutions for static 2D points by automatic extension for other data types, e.g., 3D and moving points. The method shown here can be used well equally to compute the intersection point of a moving point with the boundary of convex hull as well as the intersection of two lines that go through some moving points, which will appear in the following publications. However, computing the interception time (i.e., when the moving car crosses the boundary) requires another approach, which we will study next. The method shown here answers for any time point of interest, whether a condition is met or not, but not cannot tell when a spatial condition is achieved.

Specialized method to compute with less computational effort may be possible — the focus here was on minimizing effort to produce a GIS that works for static 2D, static 3D, moving 2D, and moving 3D objects alike — with the least amount of code adoption for each case. Performance issues of the approach, however, will be studied in our future works.

We are using the same approach to lift the *inCircle* test (i.e., whether a point is inside or outside the circle goes through three points) and from which an *n*-dimensional Delauney Triangulation program for static and moving points follows. Distance computation together with *ccw* and *inCircle* tests are the connection between metric and topological operations. Having these primitive operations lifted using the explained approach, their combination provides us with a number of metric and topological operations for 3D and moving objects.

9. REFERENCES

[1] De Berg, M., Kreveld, M.V., Overmars, M., and Schwarzkopf, O. 2000 Computational Geometry: Algorithms and Applications, 2nd Edition. Springer-Verlag, Berline.

[2] Erwig, M., Güting, R.H., Schneider, M., and Vazirgiannis, M., "Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases", *GeoInformatica*, 3(3), 269-296, 1999.

[3] Frank, A.U. 1999. One step up the abstraction ladder: Combining algebras – From functional pieces to a whole. In Proceedings of the International Conference COSIT'99, 25-29 August, 1999, Stade, Germany, Freksa, C., and Mark, D.M., (eds.), Lecture Notes in Computer Science, Springer-Verlag, 1661, 95-107.

[4] Frank, A.U. and Gruenbacher, A. 2001. Temporal Data: 2nd order concepts lead to an algebra for spatio-temporal objects. In Proceedings of Workshop on Complex Reasoning on Geographical Data, 1 December, 2001, Cyprus.

[5] Karimipour, F., Delavar, M.R., and Frank, A.U. 2008. A Mathematical Tool to Extend 2D Spatial Operations to Higher Dimensions. In Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2008), 30 June – 3 July, 2008, Perugia, Italy, Gervasi, O., Murgante, B., Lagana, A., Taniar, D., Mun, Y. and Gavrilova, M., (eds.), Lecture Notes in Computer Science, Springer-Verlag, 5072, 153-164.

[6] Karmipour F., Delavar, M.R., and Rezayan, H. 2006. Formalization of Moving Objects' Spatial Analysis Using Algebraic Structures. In Proceedings of Extended Abstracts of GIScience 2006 Conference, Munster, Germany, IfGI Prints Vol. 28, pp. 105-111.

[7] Karimipour, F., Delavar, M.R., Frank, A.U., and Rezayan, H. 2005. Point in Polygon Analysis for Moving Objects. In Proceedings of the 4th Workshop on Dynamic & Multi-dimensional GIS, Gold, C. (ed.), 5-8 September, 2005, Pontypridd, Wales, UK, ISPRS Working Group II/IV, 68-72.

[8] Karimipour, F. 2005. Logical Formalization of Spatial Analyses of Moving Objects Using Algebraic Structures, M.Sc. Thesis (in Persian with English abstract), College of Engineering, University of Tehran, Iran.

[9] Knuth, D. E., 1992 Axioms and Hulls, Lecture Notes in Computer Science, Vol. 606, Springer-Verlag.

[10] Lawvere, F.W. and Schanuel, S.H. 2005 Conceptual Mathematics: A First Introduction to Categories. Cambridge University Press.

[11] Ledoux H. 2008. The Kinetic 3D Voronoi Diagram: A Tool for Simulating Environmental Processes, in Oosterom, P.V., Zlatanova, S., Penninga, F. and Fendel, E. (eds.) Advances in 3D Geo Information Systems. Proceeding of the 2nd International Workshop on 3D Geoinformation, December 12-14, 2007, Delft, the Netherlands, Lecture Notes in Geoinformation and Cartography, Springer-Verlag, 361-380.

[12] Ledoux, H. 2007. Computing the 3D Voronoi Diagram Robustly: An Easy Explanation. In Proceeding of the 4th International Symposium on Voronoi Diagrams in Science and Engineering, Pontypridd, Wales, UK, July 9-12, 2007.

[13] Ledoux, H. 2006. Modelling three-dimensional fields in geo-science with the Voronoi diagram and its dual. PhD thesis, School of Computing, University of Glamorgan, Pontypridd, Wales, UK.

[14] MacLane, S. and Birkhoff, G. 1999 Algebra, 3rd Edition. AMS Chelsea Publishing.

[15] Mostafavi, M.A., Gold, C., and Dakowiczb, M. "Delete and Insert Operations in Voronoi/Delaunay Methods and Applications", Journal of Computers and Geosciences, 29, pp. 523-530, 2003.

[16] Nordstrom, B., Petersson, K., and J. M. Smith 1990 Programming in Martin-Lof 's Type Theory, An Introduction. Oxford University Press.

[17] Oosterom, P.V., Zlatanova, S., Penninga, F., and Fendel, E. (eds.) 2008 Advances in 3D Geo Information Systems, Proceeding of the 2nd International Workshop on 3D Geoinformation, December 12-14, 2007, Delft, the Netherlands, Lecture Notes in Geoinformation and Cartography, Springer-Verlag.

[18] Peyton Jones, S. and Hughes, J. 1999. Haskell 98: A Non-Strict, Purely Functional Language. From http://www.haskell.org/onlinereport/ (accessed June, 2008).

[19] Thompson, S. 1999 Haskell: The Craft of Functional Programming. Addison- Wesley.

[20] Chorochronos Project 2004. http://www.dbnet.ece.ntua.gr/~choros/.