Kinetic Convex Hull Maintenance Using Nested Convex Hulls

Mohammad Reza Razzazi[1], Ali Sajedi[2]

[1]Software Research and Development Laboratory, Computer Engineering Department, AmirKabir University of Technology, Tehran, Iran

razzazi@ce.aut.ac.ir

[2] Software Research and Development Laboratory, Computer Engineering Department, AmirKabir University of Technology, Tehran, Iran

alisajedi@yahoo.com

ABSTRACT

In this paper we present an effective kinetic data structure and algorithm for efficient maintenance of convex hull of moving points in 2d space. Given *n* points continuously moving in the plane we give an efficient algorithm for maintaining their convex hull. Our algorithm partitions the original points into several groups, each group's points forming a convex polygon and the polygons are nested.

1- INTRODUCTION

The problem of convex hull has been exhaustively studied in computational geometry [1, 2, 3, 6], but almost in the context of static objects with operations like insertion and deletion. Our emphasis is on maintenance of convex hull under continuous motions of the given objects. Our algorithm takes advantage of concurrency and neighbourhood in motions to achieve a minimal number of combinatorial events. From this point of view our data structure is similar to the dynamic computational geometry framework introduced by Atallah [7], which studies the number of combinatorially distinct configurations of convex hull resulting from continuous motion of objects. Our data structure does not need to know the full motion of the objects in the beginning.

Bash, Guibas, and Hershberger in [8] introduced a useful technique for maintaining convex hull and closest pair of moving points in the plane called *kinetic*. Kinetic solutions are based on occurrence of events. Each event corresponds to changes in combination of a constant number of points such as reversing the sign of angle ABC, or crossing the point A with line segment BC. They called these changes 'certificates'. Events are collected and scheduled in a global event queue. In kinetic solutions we try to minimize the number of events to reduce process time and space.

A good kinetic algorithm is local, in other words, each point is involved in only polylogarithmically many certificates, and occurrence of one event does not affect so many points and events. For more information about kinetic solutions and parameters refer to [9].

In [4], [5] the convex hull algorithm is based on upper envelopes of duals of points in 2d space and calculates a good number of events. Their work is on line segments and envelopes, but our algorithm acts directly on points and convex hull of them, hence our algorithm uses a sensible and direct approach.

Our algorithm only schedules events for adjacent points in the data structure, and hence does not involves too many events.

Each object is assumed to be a point. Thus at any time we want to have the convex hull of *n* points continuously moving in a restricted area such as a rectangle. We assume that each point has a flight plan that defines the moving direction and speed of that point. This direction can only change because of a collision between the point and borders of the region. Also we assume that points can cross each other without any collision.

2- PRELIMINARIES

It is obvious that during the time between occurrences of two consequent events, the points present in convex hull does not change, in other word the convex hull is formed of the same points (with changing places) resulting the moving convex hull. We do not need to calculate the convex hull all the time. We initialize the data structure at the beginning and then only at the scheduled times apply the events, possibly changing the status of the convex hull.

In the following, we first, present a simpler version of our kinetic data structure and then to improve its locality some modifications will be applied. However, the main algorithm is the same and can use each version of the data structure.

3- KINETIC DATA STRUCTURE

Our kinetic data structure is a set of nested convex hulls (NCHs) containing all points of the problem (maybe there will be only one or two points in the most inner convex hull, that represents respectively a point or a line segment. We assume that it also represents a convex hull). The convex hulls are kept in a simple data structure such as array of linked lists (array of convex hulls) or linked list of linked lists (linked list of convex hulls). What is important is the sequence of points in each convex hull. We study each convex hull in clockwise order and **next** and **prior** pointers for each point of it.

It is obvious that having NCHs, the convex hull is always available (The convex hull of all points is always the outer convex hull). By occurrence of any event we will update the NCHs, possibly changing the place of some points in two adjacent convex hulls or just changing the child(s) of a point, which would be defined later.

The manner of creating NCHs from initial points is as follows:

```
Algorithm createNCHs(pointsArray)      // Returns NCHs
        Input: pointsArray[1..n]
                  // Array of coordinates of input points
        Output: nested convex hull of (fixed) points
{
    S: NCHs;
    flag: array [1..n] of Boolean;
    for all points, i, in pointsArray do
        flag[i] ← false;
    while (there is any unflagged point in pointsArrsy){
        obtain the convex hull of all unflagged points of
            pointsArray, naming CH;
        add CH to S;
        set flags of all points of CH to true;
        }
    return S;
}
```

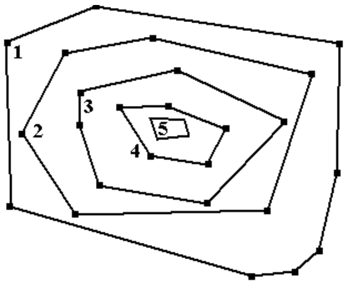

Figure 1- Nested Convex Hulls used in our kinetic data structure

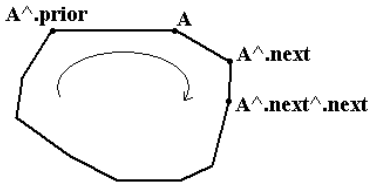

Figure 2- order of points in each convex hull is clockwise and each point has two pointers to next and prior points.

Note that only once we need to create the NCHs from initial points; the other times we change the NCHs when an event occurs. These changes are all local and we do not have a global event.

3-1- List of events

For each event we should work on the points of NCHs that cooperate in that event.

For each point there are (possibly) internal and external convex hulls. A part of NCHs is shown in Figure 3 (convex hulls A, B, C...). Lines connecting B_3 to C_2 and B_3 to C_6 indicates the FirstChild and LastChild of B_3 that will be defined for fully localizing the algorithm.

These definitions for a point, X, as follows:

- firstChild: Consider a point, P, on immediate inner layer of X (IL(X)) which is not visible from X. Moving clockwise from p on IL(X), the first point visible from X is called firstChild(X).

- lastChild: Same as firstChild, the last point visible from X is called lastChild(X).

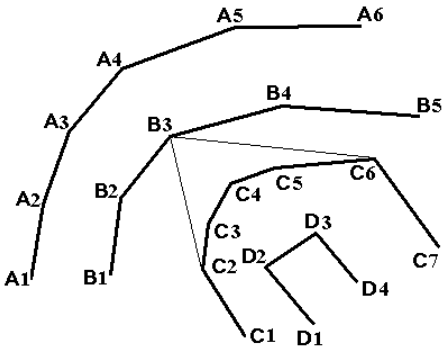

Figure 3- a part of NCHs named alphabetically

For example for point B_3 the points as firstChild and lastChild are shown in Figure 3 (C_2 and C_6).

Important events that may change the outer convex hull include: moving a point, p, from convex hull *i* to convex hull *i+1* or *i-1* (*goOut(p) and goIn(p)*).

List of possible events for a point such as B_3 is as follows:

1- Reaching the border of the region and reflecting the direction of motion. We call this event changeDir(B_3)

2- Moving B_3 toward the inner convex hull (C). We call this event *goIn(B_3).*

3- Moving B_3 toward the outer convex hull (A). We call this event *goOut(B_3).*

4- Exiting C_2 from visibility region of B_3 (C_3 intersects line segment B_3C_2). We call this event notFirstChild(B_3).

5- Entering C1 into visibility region of B_3. We call this event beFirstChild(B_3).

6- Exiting C_6 from visibility region of B_3 (C5 intersects line segment B_3C_6). We call this event notLastChild(B_3).

7- Moving C_7 into visibility region of B_3. We call this event beLastChild(B_3).

The last four events ensures having correct firstChild and lastChild for each point at event times.

At each scheduled event we do necessary modifications to NCHs. These changes are all local. For example in case of Figure 3, moving B_3 toward convex hull C leaves B_2 and B_4 in convex hull B as neighbours (deleting B_3 form convex hull B), also inserts B_3 as a point of convex hull C between C_2 and C_6. This results entering C_3, C_4 and C_5 recursively in inner convex hulls. This (entrance to inner convex hulls

recursively) continues until the node that should enter inside has only its two Children as visible points of inner convex hull; in this case deleting it from outer convex hull and inserting it in the inner convex hull is enough. This operation is called *goIn*. In Figures 4, 5 examples of simple goOut and simple goIn are shown. Simple events only change the two adjacent convex hulls, and do not change the other internal convex hulls, whilst complex events change several nested convex hulls recursively.
All these operations are applied at each event and NCHs are updated accordingly.

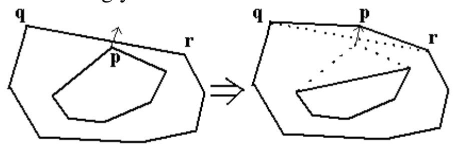

Figure 4- Simple *goOut(p)* event and changes made in the NCHs

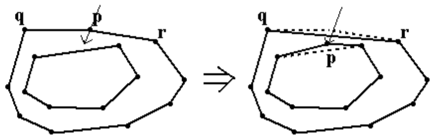

Figure 5- Simple *goIn(p)* event and changes made in the NCHs

In Table 1 we show the changes needed to do with occurrence of each event:

Event name	Certificate	Changes after occurrence
changeDir (B_3)	Crossing B_3 with the border of the region	A reflected direction of B_3 with respect to the border will be applied
goIn(B_3)	Crossing B_3 with line segment B_2B_4	B_3 will be inserted in the inner convex hull that may result in goIn(X_i) for some X and i ($X \in$ layers C, D, …) recursively
goOut (B_3^. firstChild)	Crossing B_3 ^.firstChild (e.g. C_2) with the line segment B_3B_3^.prior	B_3^.firstChild (e.g. C_2) will be deleted from inner convex hull and will be inserted between B_3 and B_3^.prior (e.g. B_2) that may result in goOut(X_i) for some i and X ($X \in$ layers C, D, …) recursively
notFirstChild(B_3)	Crossing the point B_3^.firstChild ^.next (e.g. C_3) with line segment B_3B_3^.firstChild (e.g. B_3C_2)	B_3^.firstChild^.next (e.g. C_3) will become the new B_3^.firstChild
beFirstChild(B_3)	Crossing the point B_3^.firstChild ^.prior (e.g. C_1) with extension of line segment B_3B_3^.firstChild (e.g. B_3C_2)	B_3^.firstChild^.prior (e.g. C_1) will become the new B_3^.firstChild
notLastChild(B_3)	Crossing the point B_3^.lastChild ^.prior (e.g. C_5) with line segment B_3B_3^.lastChild (e.g. B_3C_6)	B_3^.lastChild^.prior (e.g. C_5) will become the new B_3^.lastChild
beLastChild(B_3)	Crossing point B_3^.lastChild ^.next (e.g. C_7) with extension of line segment B_3B_3^.lastChild (e.g. B_3C_6)	B_3^.lastChild^.next (e.g. C_7)will become the new B_3^.lastChild

Table 1- List of possible events

In the next section we present the algorithm working with the data structure to maintain the convex hull.

4- KINETIC CONVEX HULL MAINTENANCE ALGORITHM

The high-level pseudo code for this algorithm is given below. The code simulates NCHs maintenance and animates moving NCHs. Note that all links are implemented by pointers; each point has a pointer to next and previous points, and pointers to its firstChild and lastChild. With occurrence of each event these pointers are modified according to type of event.

```
Algorithm Kinetic_Convex_Hull:
        Input:   pointsArray[1..n]
                     // Array of coordinates of input points
                 simTime
                     // The simulation time
        Output:  moving convex hull of points
Var
    S: NCHs;
    E: linked list of Events sorted by event time;
    t, occurrenceTime: Time
{
    S ← createNCHs(pointsArray);
    S ← linkChilds(S);
        // : for all points finds firstChild and lastChild points
        // (if any)
    E ← scheduleAllEvents(S);
        // For almost all points there will be 7 events
        // according to definition in section 3-1.
        // Exceptions are the points on innermost convex hull
    t ← getTime();
    simTime ← simTime + t;
    occurrenceTime ← E^.occurrenceTime;
        // E^.occurrenceTime is the time of first event
    while(occurrenceTime < simTime){
        draw S till time occurrenceTime;
            // S has the same configuration during this time
        applyFirstEvent(E, S);
        // Applies the first event of list E. Applying this
        //  event may change both E and S. Because of
        // this, these two these parameters are called by
        // reference and change their values in the
        // function. The places of points of NCHs are updated
        // and their related events in E are rescheduled.
        E ← E^.nextEvent;  // pointing E to next Event.
        occurrenceTime ← E ^.occurrenceTime;
    }
}
```

5- CONCLUSIONS

Using the framework defined in [8] we proposed an efficient algorithm using a direct approach. Because of the nature of convex hull, it is difficult to localize the problem. By using nested convex hulls we showed that each point's motion may only change the status of some neighboring points, and as a result were able to eliminate many events and achieve efficiency.

6- REFERENCES

[1] 1. J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. BIT, 32:249-267, 1992.

[2] 2. M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. J. Comput. Syst. Sci.,23:166-204, 1981.

[3] 3. F. P. Perparata and M. I. Shamos. Computational Geom-etry: An Introduction. Springer-Verlag, New York, NY, 1985.

[4] 4. P. K. Agarwal, O. Schwarzkopf, and M.Sharir. The overlay of lower envelopes and its applications. Discrete Comput. Geom., 15:1-13, 1996

[5] 5. J. Hershberger. Finding the upper envelope of n line seg-ments in *O(nlogn)* time. *Inform* Process. Left., 33:169-174, 1989.

[6] 6. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry, Algorithms and applications. Springer-Verlag, Berlin, 2000.

[7] 7. M. J. Atallah. Some dynamic computational geometry problems. Comput. Math. Appl., 11:1171-1181, 1985.

[8] 8. J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. SoDA, 1997.

[9] 9. J. Basch L. J. Guibas, C. D. Silverstein and L. Zhang. A Practical Evaluation of Kinetic Data Structures. 13th Symposium on Computational Geometry, 1997.