Software project

Participating in the
ACM SIGSPATIAL GIS CUP 2013
programming contest

May 2013

Sebastian Kürten (sebastian.kueten@fu-berlin.de)

Contents

1 Introduction

The problem of the SIGSPATIAL GIS CUP 2013 programming contest deals with the problem of evaluating spatiotemporal predicates defined on sets of geographic data. The basic data types involved in the problem are **points** and **regions**. A single point is defined by its coordinate and may represent for example the position of a car. A region is an object of two-dimensional shape, a discrete representation of an area, specified by a set of polygonal objects. In the form considered here these areas may consist of several disjoint polygons and each of those polygons may contain holes. A region is capable of modeling the geometry of complex geographic objects like country boundaries or clouds.

On top of this, the data used in our problem is not only of geometric nature but also contains a temporal component. Instead of points we deal with sequences of points over time and instead of regions we deal with sequences of regions over time. Think of a sequence of points as of the trajectory of a moving object, like a car for example. So instead of one position of the car, we have the position of the car at several points in time. Similarly a sequence of regions may represent the movement and transformation of a cloud, where not only its position changes depending on the time but also its shape. We are dealing with discrete representations of such data here, that means they are not only discrete in terms of their geometric representation but also in respect to their temporal component. We are looking at different representation of the same object for different points in time and each representation is considered valid for the span of time beginning at its definition until it is replaced by a more recent representation.

The algorithmic problem to solve here is to evaluate two predicates defined on sets of such data. First, given a point with an associated timestamp we would like to find all regions that the given point lies within at the specified time. Second, given a point with timestamp we would like to find all regions that are, at the specified time, within a specified distance of that point.

1.1 Outline

Section 1.2 explains the input data provided by the contest committee and introduces formalization that will be used in the remainder of the document. Based on this, section 1.3 formally describes the problems to be solved. Section 2 presents the training data provided by the committee. Sections 3 to 5 present the solutions to the problems that have been implemented. The preliminary steps that had to be undertaken such as parsing input data, preprocessing data and formatting output are described in section 3. Section 4 presents algorithms for the INSIDE query and section 5 deals with algorithms for the WITHIN-N query.

1.2 Definitions

- The basic data types point and region consist of
  - an **identifier** that allows to model a moving object by defining multiple instances with the same identifier,
  - a **timestamp** which defines the starting time of validity of an instance
  - and an **geometric component** that models the geometric part of the object at the specified time.
- Let $p = (p_{Id}, p_t, p_{geom})$ be a point with identifier p_{Id}, timestamp p_t and geometry p_{geom}
- Let $r = (r_{Id}, r_t, r_{geom})$ be a region with identifier r_{Id}, timestamp r_t and geometry r_{geom}
- The problem presented here deals with collections of points and regions:

  - let $\mathcal{P}$ be the set of all input points,
  - let $\mathcal{R}$ be the set of all input regions.

- For a given point p, let $seq(p)$ be the set of all points with the same identifier, that is $seq(p) = \{q \in \mathcal{P} | q_{Id} = p_{Id}\}$. We will call $seq(p)$ the sequence of p.
- For a given region r, let $seq(r)$ be the set of all regions with the same identifier, that is $seq(r) = \{s \in \mathcal{R} | s_{Id} = r_{Id}\}$. We will call $seq(r)$ the sequence of r.
- Let $seq(\mathcal{P})$ be the set of all sequences of input points
- Let $seq(\mathcal{R})$ be the set of all sequences of input regions

1.3 Problem definition

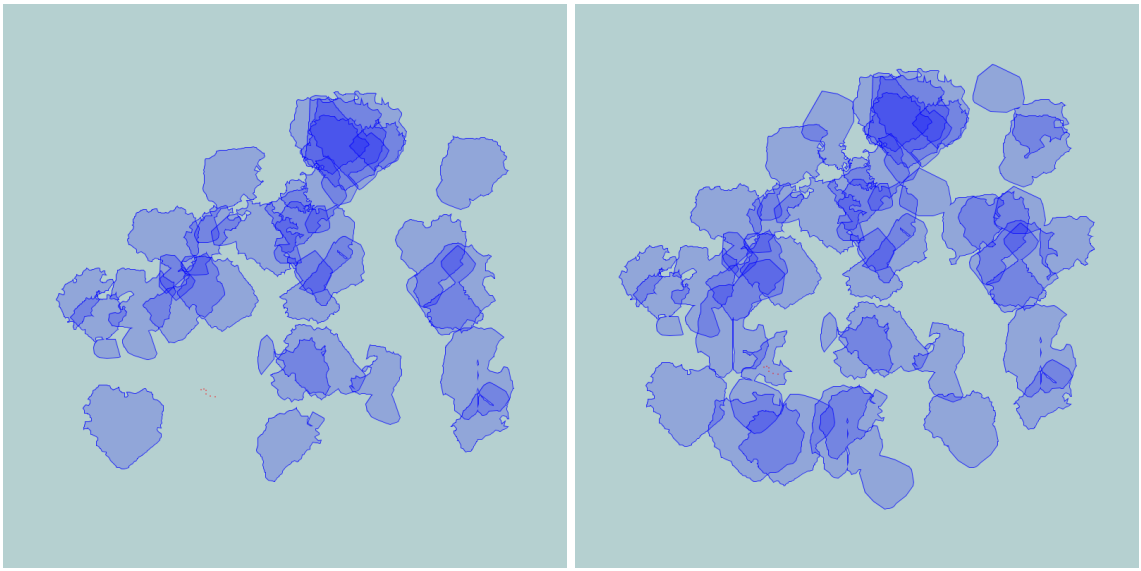
- Input:
  - Set of general regions $\mathcal{R}$, $|\mathcal{R}| \leq 500$
  - Set of points $\mathcal{P}$, $|\mathcal{P}| \leq 1million$
- Output:
  - List of tuples $(point, region)$ that satisfy a given spatio-temporal predicate
  - Two different predicates:
    - p INSIDE r: point p is *inside the interior* of region r
    - p WITHIN-N r: point p is *within a specified distance* of region r
  - In addition to the spatial aspect, these queries are further specified to have a temporal constraint:
    - For each point, not the whole set of regions is to be considered as candidates for satisfaction of the predicates, but only the one instance over time of each sequence of regions that exists at the point in time that is associated with the query point.
    - That means that for each sequence of regions R there is at each point in time at most one instance $r \in R$ that is eligible for satisfying the predicate for a specified point.
  - more formally: Let
    - $p = (p_{Id}, p_t, p_{geom})$ be a query point,
    - $r = (r_{Id}, r_t, r_{geom})$ be a region with $r \in \mathcal{R}$,
    - then p INSIDE r is true iff $(p_{geom}$ INSIDE $r_{geom}) \wedge (r_t \lneq p_t) \wedge (\forall s \in (seq(r) \setminus r) : (s_t < r_t \vee s_t > p_t))$
    - and p WITHIN-N r is true iff $(distance(p_{geom}, r_{geom}) < N) \wedge (r_t \lneq p_t) \wedge (\forall s \in (seq(r) \setminus r) : (s_t < r_t \vee s_t > p_t))$

2 Training data

The training data consists of two sets of regions and two sets of points. The polygonal training data contains 10 and 15 sequences of regions while the point training data contains 500 and 1000 sequences of points.

2.1 Regions

The following figure illustrates the regions found in the training data set. The dataset contains 30 and 40 regions distributed over 10 and 15 sequences respectively.



(a) 10 sequences (b) 15 sequences

Figure 1: Polygonal training data

2.2 Points

The following figures illustrate the points found in the training data set along with the regions. The dataset contains approximately 40000 and 70000 points distributed over 500 and 1000 sequences respectively.

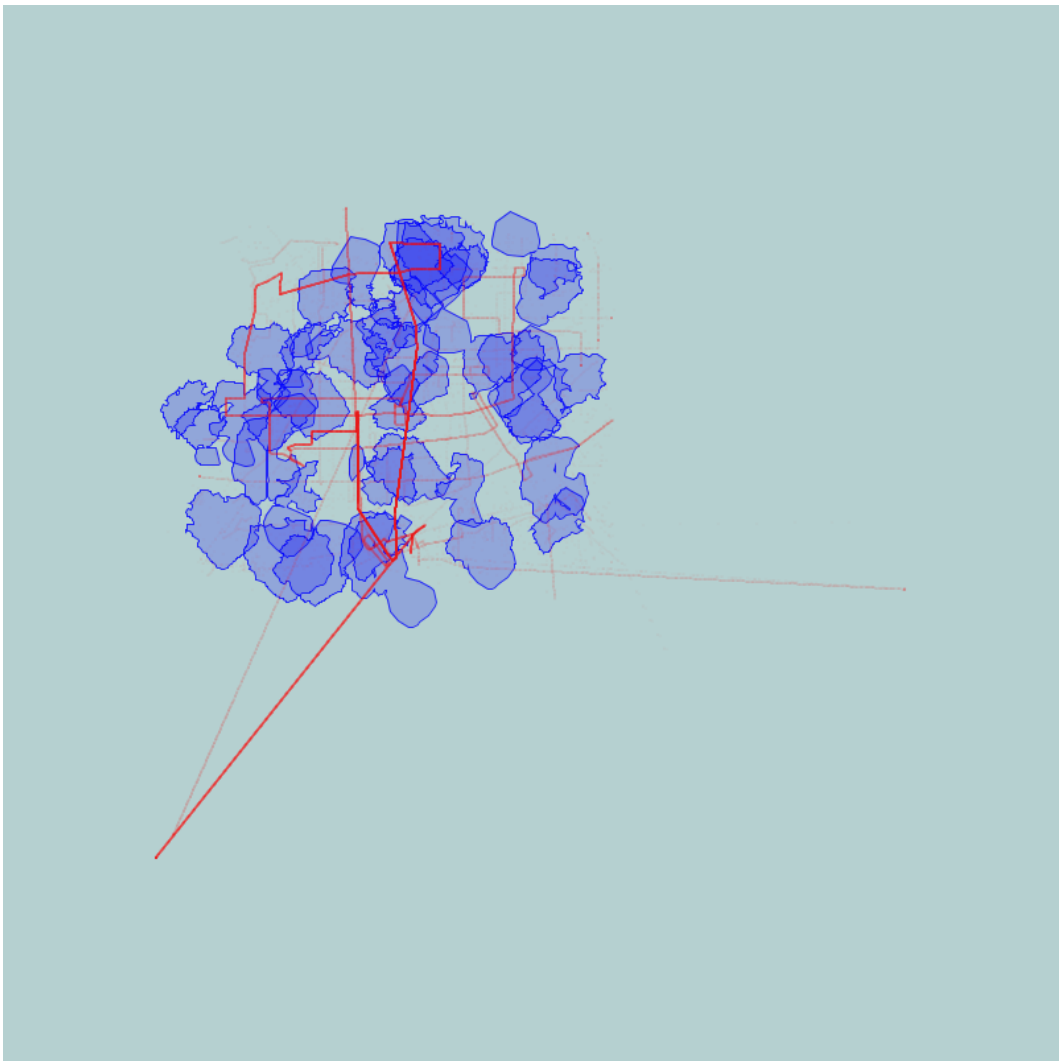


Figure 2: All points together with polygonal input

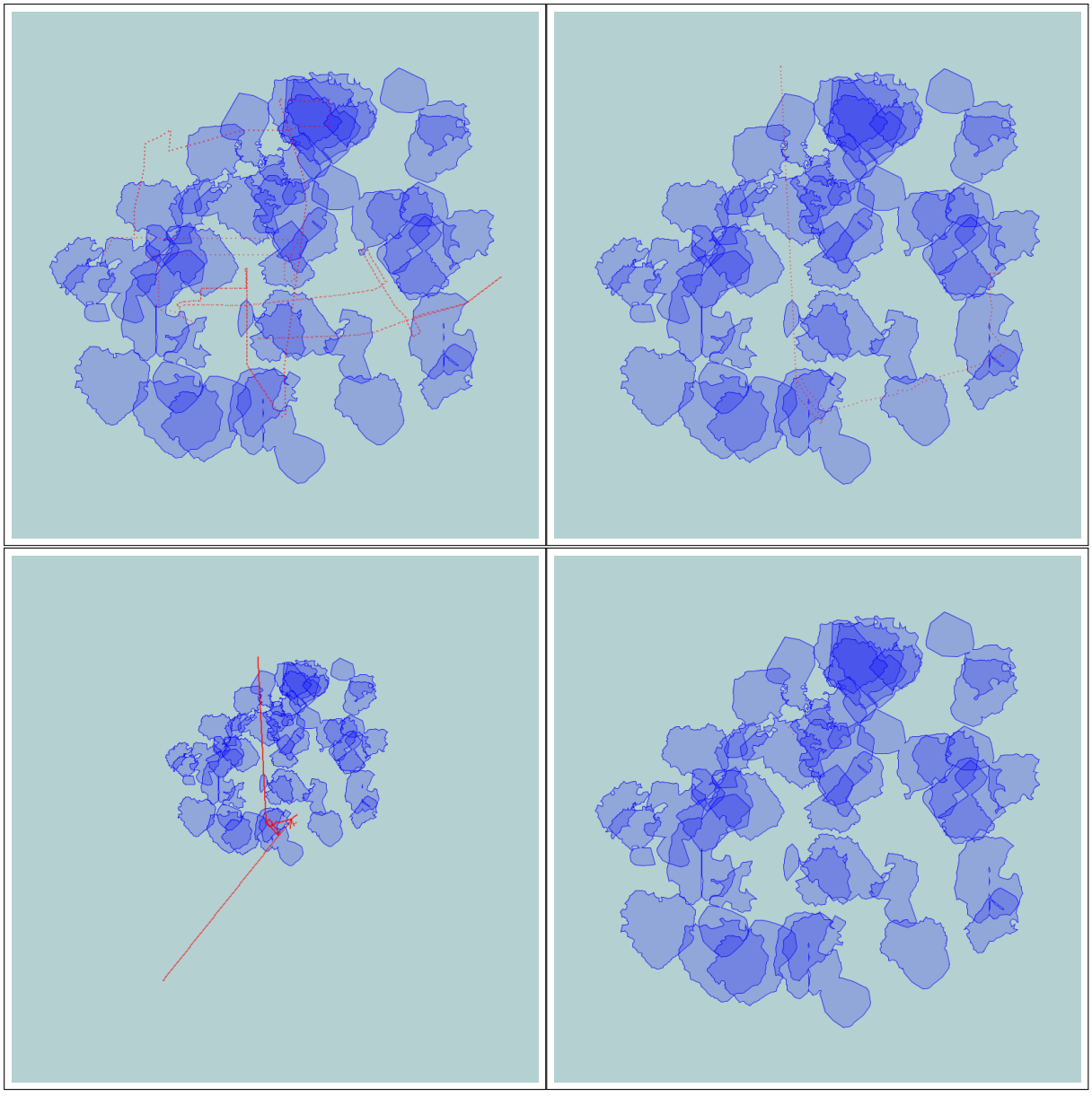


Figure 3: Four sequences of point input together with polygonal input

3 Preliminaries

In order to deal with the primary algorithmic problems of the contest some secondary problems had to be solved. This includes reading and parsing of input data, formatting results for output and some preprocessing that makes the data easier to handle in further steps. This section will deal with those problems so that the remainder of this report may focus on the algorithmic problems, i.e. predicate evaluation for sets of already preprocessed data.

3.1 Input

The program receives as input two files, one for points and one for regions. The input data comes in form of an odd line based text format that combines structured text data with XML strings embedded within such structured text. Each line starts with the string `POINT:<ID>:<TIMESTAMP>:` or `POLYGON:<ID>:<TIMESTAMP>:` followed by an XML snippet in the so-called GML format which is a flexible and verbose format for storing geographic data. Figures 4 and 5 show examples of such input data where a single line of input data has been reformatted with additional whitespace and line-breaks for better readability.

```
POINT:0:501:<gml:Point srsName="EPSG:54004"
          xmlns:gml="http://www.opengis.net/gml">
        <gml:coordinates decimal="." cs="," ts=" ">
                -1.3148438178342E7,3986007.33707501
        </gml:coordinates>
</gml:Point>
```

Figure 4: Example of point input data

When parsing the GML part of the input with an appropriate library that handles this kind of data with all its flexibility such as the GML implementation of the GeoTools[1] opensource project, the program takes so much time parsing input data that this step becomes by far the dominant part of the overall program.

Thus specialized methods have been developed to parse the input data with minimal overhead, using as much information about the structure of data as possible while maintaining a certain level of robustness against variations that may occur in the input data as a result of the flexibility of the GML format. Anyway I am making assumptions about the input that are not necessarily true. For instance I am completely ignoring the GML attributes that define the decimal point character and the character that separates two successive coordinates but instead assume them to be the same as presented in the training data set.

The first part of each line, the part that defines identifier and timestamp of an instance is easy to parse and can be done by inspecting the characters one by one.

For the points a simple parser has been written that reads one character at a time and is controlled by a really simple finite state machine that skips over irrelevant data and cumulates only the characters belonging to the definition of the coordinates. When the parser recognizes the end of characters that belong to the coordinates, it stops parsing the line and delegates parsing of the strings known to contain floating-point numbers to the default mechanism for parsing floating point numbers provided by the core Java libraries.

For the regions a more flexible approach has been chosen because on the one hand the GML format encountered here has a deeper XML structure with variable number of elements making parsing more complicated and on the other hand because the execution time is not as crucial as with the points simply because the amount of data is much smaller. Thus a specialized XML

[1] http://www.osgeo.org/geotools

```
POLYGON:1:1:<gml:Polygon srsName="EPSG:54004"
          xmlns:gml="http://www.opengis.net/gml">
        <gml:outerBoundaryIs>
                <gml:LinearRing>
                        <gml:coordinates decimal="." cs="," ts=" ">
                                -1.31653586977661E7,3983548.08445849
                                -1.31653674099771E7,3983543.00364778
                                -1.31653536413005E7,3983539.38244563
                                ....
                                -1.31619654875391E7,3977038.16221225
                                -1.31620352006778E7,3976815.51994722
                        </gml:coordinates>
                </gml:LinearRing>
        </gml:outerBoundaryIs>
        <gml:innerBoundaryIs>
                <gml:LinearRing>
                        <gml:coordinates decimal="." cs="," ts=" ">
                                -1.31422316898252E7,4006504.77674048
                                ...
                                -1.31422316898252E7,4006504.77674048
                        </gml:coordinates>
                </gml:LinearRing>
        </gml:innerBoundaryIs>
</gml:Polygon>
```

Figure 5: Example of region input data

parser has been implemented on top of the relatively lightweight SAX-Parser API provided by standard Java. This parser is nowhere near to implementing all possible ways of representing regions in GML but implements just those parts of the format that occur within the training dataset and again I assume that the data presented to the program will be of approximately the same format. At least this implementation is not vulnerable for instance to whitespace formatting issues and is able to cope with at least some of many possible ways of representing regions in GML.

3.2 Preprocessing

As described in section 1.2 each region instance defined in the input file belongs to sequence of regions that models the same object only at different points in time. Each instance has an associated timestamp that defines when on the one hand this instance becomes valid and on the other hand when its predecessor becomes invalid. This implies that instead of a single timestamp each region instance really has an associated interval in which it is valid. Knowing these intervals is crucial when evaluating predicates with regions and points because one of the conditions of the predicate depends exactly on this interval: the timestamp of a point p must be within the time-interval of a region instance r so that (p, r) is allowed to fulfill any predicate.

Because these intervals are only implicitly given, they have to be derived from each sequence of region instances. Fortunately this is neither complicated nor significantly time consuming. We just have to gather all instances of the same region in a list, sort them by their timestamp and may derive the time interval for each instance by looking up the instance and its successor.

For example consider the following four lines of input data that define four instances of the region with identifier 7:

```
POLYGON:7:7:<gml:Polygon ... >
POLYGON:7:136355:<gml:Polygon ... >
POLYGON:7:161765:<gml:Polygon ... >
POLYGON:7:676995:<gml:Polygon ... >
```

We can derive that there are four intances of region number 7 being valid at the following intervals:

- POLYGON:7:7: valid from 8 to 136355
- POLYGON:7:136355: valid from 136356 to 161765
- POLYGON:7:161765: valid from 161766 to 676995
- POLYGON:7:676995: valid from 676996 to ∞

In the remainder of the report I assume that this preprocessing has happened and that for each region instance r the respective interval may be accessed via $r.time_{start}$ and $r.time_{end}$.

3.3 Output

The output format is specified as a file containing one line for each pair of region and point that matches the specified predicate with the line being formatted like this:

```
POINT-ID:POINT-TIMESTAMP:REGION-ID:REGION-TIMESTAMP
```

For instance the following lines are valid output data:

```
2:503:3:4
2:2498:3:4
4:3749:5:5
4:3749:8:8
4:4277:5:5
4:4277:8:8
...
```

The first line describes that the point with identifier 2 and timestamp 503 matched the specified predicate together with the region instance with identifier 3 and timestamp 4.

Implementing the output operations did not incur any problems.

4 The INSIDE query

This section describes the algorithms implemented to solve the INSIDE query. Input for this query is a set of regions $\mathcal{R}$ and a set of points $\mathcal{P}$. Each instance has a geometric component and additionally each region instance has a temporal interval of validity and each point instance has an associated timestamp. To find is the set of all pairs $(p, r) \in \mathcal{P} \times \mathcal{R}$ for which geometrically p INSIDE r is true and for which the point's timestamp is contained within the region's interval of validity.

4.1 Simple solution

The simplest solution appears to be a loop over both regions and points and then to evaluate for each pair $(region, point)$ whether it satisfies the spatiotemporal INSIDE-predicate. Algorithm 1 shows pseudocode for this solution.

Algorithm 1 Calculating all pairs in $P \times R$ that satisfy the spatiotemporal INSIDE predicate

```
1: for each point p ∈ P do
2:     for each region r ∈ R do
3:         if SPATIOTEMPORAL-INSIDE(p, r) then
4:             OUTPUT(p, r)
5:         end if
6:     end for
7: end for

8: function SPATIOTEMPORAL-INSIDE(p, r)
9:     return p.time > r.time_start and p.time ≤ r.time_end and INSIDE(p_geom, r_geom)
10: end function
```

4.2 Using spatial indexing

Spatial indexes such as R-Trees are data structures that store sets of spatial data and that allow for efficient retrieval of intersecting data for rectangular and point queries. They may be used to efficiently replace loops over large sets of data resulting in a typical pattern[RSV01] of *filter* step and *refinement* step. Algorithms following this pattern use a query to the spatial index to retrieve a set of candidates during the *filter* step, which will then, in the *refinement* step, be examined in detail to find out whether a candidate really satisfies a predicate.

Indexing regions

Spatial indexes may be used to store the regions to increase query performance when compared to a naive nested loop implementation once the number of regions becomes sufficiently high. Algorithm 2 shows a straightforward transformation of algorithm 1 to an algorithm following the *filter-refinement* pattern.

The second algorithm improves the first algorithm in that for each point, not every region is considered for satisfaction of the predicate, but only a subset of all available regions. This subset contains all those regions that are viable for the satisfaction of the predicate because their bounding box overlaps the coordinate of the query point.

The running time of the query to the spatial index in line 6 is output sensitive and depends on the number of candidates that will be found in the search tree. When we assume that the regions are reasonably scattered across our area of interest we would expect the number of regions to be reasonably small in each step so that each step should take only logarithmic time in the number of regions contained within the data structure.

Algorithm 2 Calculate all pairs in $P \times R$ that satisfy the spatiotemporal INSIDE predicate using a spatial index to efficiently store and retrieve regions

```
1: index ← initialize spatial index
2: for each region r ∈ R do
3:     index.insert(r)
4: end for
5: for each point p ∈ P do
6:     C ← index.query(p)
7:     for each region r ∈ C do
8:         if SPATIOTEMPORAL-INSIDE(p, r) then
9:             OUTPUT(p, r)
10:        end if
11:    end for
12: end for
```

Indexing points

Similarly, instead of indexing regions, the points may be inserted into a spatial index, which may then be queried for each region to build a set of candidate points. The resulting algorithm for the INSIDE query is the same as algorithm 2 except that the semantics of points and regions have to be swapped. As it takes $O(n \cdot log(n))$ time to build an R-Tree and the number of points is relatively high the overall running time with this approach has been found to be suboptimal for the training data set.

4.3 The point in polygon problem

The strategies explained in the previous section allow us to find efficiently the set of all pairs of points and regions (p, r) that are viable candidates for the satisfaction of the INSIDE predicate because p is contained within the *bounding box* of r and that the temporal constraint is met for (p, r) as well. What remains to find is an efficient implementation of the *point in polygon* problem to decide whether a candidate pair (p, r) really belongs to the result set of a query.

The point in polygon problem is a well studied problem and [Hai94] gives an overview of the algorithms that are efficient in practice. Most of the algorithms found here are based on one of the following observations:

- Jordan Curve Theorem: when shooting a ray from the point in an arbitrary direction, one may count the number of crossings of the ray with the polygon's edges. If the number of crossing edges counted is odd, the point is inside the polygon and outside if that number is even.
- Winding numbers: they count the number of times a polygon boundary goes around the point. It may be calculated by summating the signed angles of each segments endpoints when viewed from the point. Depending on the definition of interior and exterior of the polygon, the winding number tells us whether the point is inside or outside.

Only a subset of the algorithms described by Haines have been implemented and tested on the training data set. One of the most promising and yet simple approaches is the so called *crossings test* which has been implemented and is described in section 4.3.1.

I tried to improve the crossings test by reducing the number of edges that have to be inspected for each query point. Those approaches can be found in sections 4.3.2, 4.3.3 and 4.3.4 respectively. In an attempt to build an even more efficient implementation for the point in polygon problem, another algorithm has been developed which can be found in section 4.3.5. This

algorithm builds a data structure that approximates the shape of the region with rectangles of decreasing size such that in most cases queries may be answered by a simple tree lookup.

4.3.1 The crossings test

The crossings test is a ray shooting algorithm based on the Jordan Curve theorem first presented by Shimrat[Shi62]. The basic form of the crossings test is given in algorithm 3. The algorithm consists of a loop over all segments of a polygon which will then be tested for intersection with a ray shot from the point along the x-axis to positive infinity. Each intersecting segment updates a counter variable. The counter will be decremented if the segment crosses the ray from above the ray to below the ray and will be incremented if it crosses from the other direction. At the end of the loop, the point is classified as in the polygon if the counter variable is not zero. Algorithm 3 gives pseudocode for the basic crossings test.

Algorithm 3 The basic crossings test

```
 1: function CROSSINGSTEST(P, x, y)
 2:     int crossings ← 0
 3:     for each segment s ∈ P do
 4:         if ray from (x, y) intersects s then
 5:             if s starts above the ray and ends below the ray then
 6:                 decrement crossings
 7:             else
 8:                 increment crossings
 9:             end if
10:         end if
11:     end for
12:     return crossings != 0
13: end function
```

The crossings test may be expressed in real programming languages as described in algorithm 4 which is a quite similar to the presentation of the algorithm in [Hai94]. In the form shown here segments will be skipped very fast if they are completely above or completely below the ray. In the next step, we look at the x-coordinates and if the segment is completely to the right of the point, we know that the ray intersects the segment. Similarly, if it is completely to the left of the point, we know that the ray does not intersect the segment. Only if one vertex of the segment is to the left of the point and the other is to the right, we need to calculate the actual intersection with the ray.

4.3.2 An interval-tree based crossings test

In the basic crossings test, the y-coordinates of each segment's endpoints are checked first and may let us skip a segment quickly if an intersection with the ray is impossible. An intersection of a segment s with a ray parallel to the x-axis and originating at point (p_x, p_y) is only possible if $p_y \geq s.y_1 \wedge p_y \leq s.y_2$ (assuming that $s.y_1 \leq s.y_2$). For many real world polygons we would expect that many edges will be skipped because of the query point's y-coordinate and only very few being inspected further. This leads to the idea of inserting all segments of the polygon into an interval tree with the endpoints of each interval being the y-coordinate of the respective segment. Executing the crossings test then breaks down to querying that interval tree based on the query point's y-coordinate and then for each segment that we retrieve execute the same statements as in algorithm 4 between lines 7 and 17. Algorithm 5 describes this approach in pseudocode.

Algorithm 4 The basic crossings test, in concrete code

```
1: function CROSSINGSTEST(P, x, y)
2:     int crossings ← 0
3:     for each segment s ∈ P do
4:         boolean yflag0 ← (s.y1 ≥ y)
5:         boolean yflag1 ← (s.y2 ≥ y)
6:         if yflag0 != yflag1 then
7:             boolean xflag0 ← (s.x1 ≥ x)
8:             boolean xflag1 ← (s.x2 ≥ x)
9:             if xflag0 == xflag1 then
10:                if xflag0 then
11:                    crossings += (yflag0 ? -1 : 1)
12:                end if
13:            else
14:                if (s.x2 − (s.y2 − y) · (s.x1 − s.x2)/(s.y1 − s.y2)) ≥ x then
15:                    crossings += (yflag0 ? -1 : 1)
16:                end if
17:            end if
18:        end if
19:    end for
20:    return crossings != 0
21: end function
```

Algorithm 5 The crossings test with segments of a polygon stored in an interval tree

```
1: function INTERVALTREE-CROSSINGSTEST-PREPARE(P)
2:     tree ← initialize interval tree
3:     for each segment s ∈ P do
4:         tree.insert(s.y1, s.y2, s)
5:     end for
6: end function

7: function INTERVALTREE-CROSSINGSTEST(x, y)
8:     C ← tree.query(y)
9:     for each segment s ∈ C do
10:        if ray from p intersects s then
11:            update crossings counter
12:        end if
13:    end for
14: end function
```

4.3.3 A bucket based crossings test

The crossings test runs in $O(n)$ time for a polygon with n segments. The idea of the bucket based crossings test is to reduce the running time of the algorithm by some constant factor with minimal preprocessing overhead. To achieve this we create an array of buckets that will be iterated instead of the segments in the main loop of the crossings test. Each bucket contains a chain of segments of the original polygon and stores, as calculated during the preprocessing phase, the minimum and maximum y-coordinate of all segments of that chain. When iterating the buckets, we may now skip complete chains of segments after comparing only the y-coordinate of the query point to the minimum and maximum y-value of a bucket. Only if $p_y \geq bucket.y_{min} \wedge p_y \leq bucket.y_{max}$ we take a look at each segment stored in the respective bucket. Of course we may still end up looking at each segment of the polygon but for certain polygons we can hope to skip a lot of segments faster than with the basic crossings test.

4.3.4 A bins based crossings test

In a publicly available draft[2] of [Hai94] Haines describes another optimized version of the crossings test. This algorithm divides the bounding box of the polygon in m horizontal bins of equal vertical size such that each bin represents an y-interval. In the preprocessing step we find for each bin all segments that intersect the bin and store them in an appropriate data structure along with the bin. Now to execute the crossings test for a given point, we first determine in constant time, based on the point's y-coordinate, the bin that contains all segments that are relevant for the crossings test. Although we may still end up looking at nearly all segments for degenerate polygons we would again assume that for many polygons we would only look at a subset of the segments for most query points and that this subset is significantly smaller than the number of segments of the whole polygon. This algorithm may be improved to increase performance:

- by storing for each bin the x bounds so that a query point outside these bounds can be classified as outside the polygon in constant time.
- by sorting the segments within each bin according to their decreasing maximal x-coordinate so that we may stop iterating the segments once we have reached a segment whose maximal x-coordinate is less than the point's x-coordinate.
- by storing a flag for each segment whether it crosses the bin's y-interval completely. For such segments we may skip the initial check of the y-coordinates of the segment because the point's y-coordinate is then known to lie within the y-range of the segment.

4.3.5 The adaptive grid approximation method

I developed another algorithm that aims to approximate the shape of the polygon with rectangles of decreasing size such that these rectangles are completely contained in the interior of the polygon. The idea is to insert those rectangles into a spatial index such as an R-Tree so that for the biggest portion of the polygon a query to that spatial index will classify interior points as in the polygon so that we are done quickly. Additionally the rectangles crossed by the boundary of the polygon that have not been further partitioned are stored in the index as well. If the query to the index returns such a rectangle we will have to do some more calculations to decide whether the point is inside or outside. If the query to the index returns an empty resultset, we know that the point is outside and we are done quickly as well.

[2] http://erich.realtimerendering.com/ptinpoly/

The algorithm that sets up the data structures works recursively on rectangles beginning with the bounding box of the polygon. For a rectangle under consideration it will be tested whether the rectangle is completely contained within the polygon. If this is the case, the rectangle is stored as completely interior and the recursion stops. Otherwise the rectangle intersects the boundary of the polygon. Unless the area of the rectangle does not fall below some threshold the rectangle will be split along its longest side and the algorithm continues recursively for both parts. If on the other hand the area of the rectangle does fall below that threshold, the recursion ends and the rectangle will be stored as a boundary rectangle.

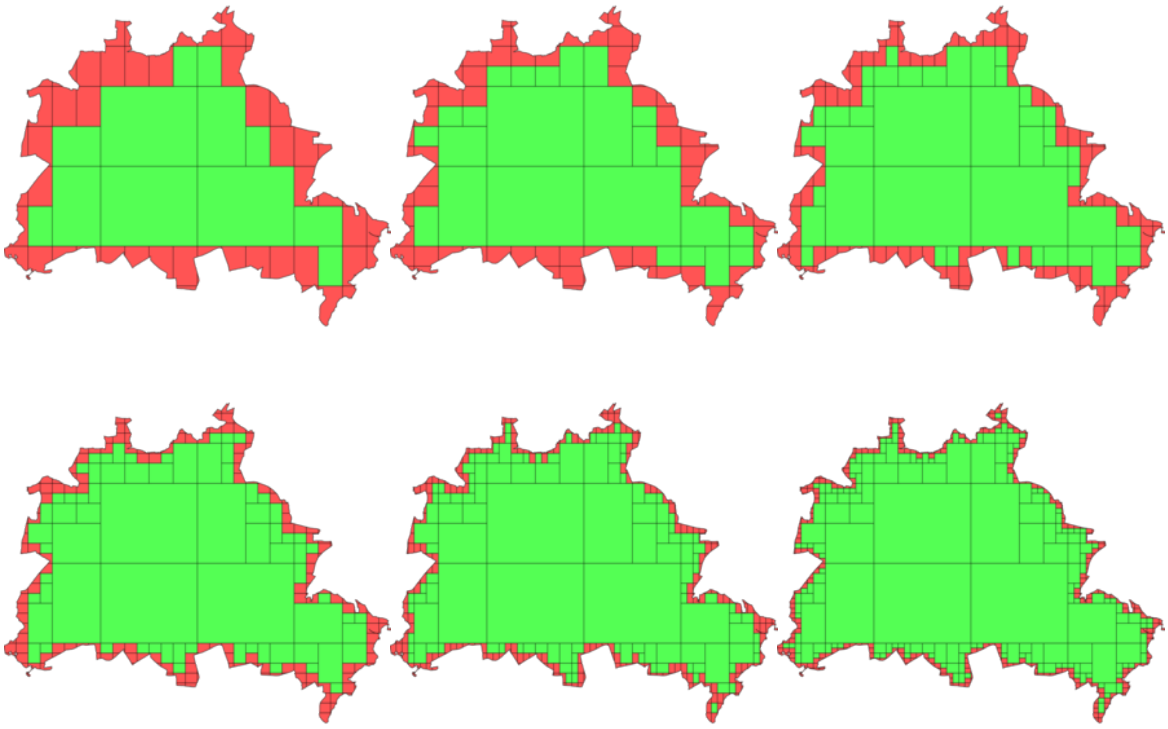


Figure 6: Data structure built for the boundary of Berlin

Figure 6 visualizes the resulting data structures built by the algorithm for the boundary of the city of Berlin for decreasing values of the recursion threshold. Green cells represent rectangles that are completely within the interior of the polygon and red cells represent boundary cells where the recursion stopped at rectangles that still overlap with the boundary. Only the intersection of the red rectangles with the boundary is shown in the figure.

When querying the data structure built up so far with a query point, three cases may occur:

1. The query returns a green rectangle. That means the point is inside the polygon and we are done.
2. The query returns no rectangle at all. That means the point is outside the polygon and we are done.
3. The query returns a red rectangle.

While in the first two cases we know how to classify the point, the third case requires a fallback method to classify the point. One way to handle this case is to simply run the basic crossings test or any other general purpose point in polygon test for those points. While this definitely works we can do something more clever here. Observe that all boundary cells are of equal size and thus partition the bounding box of the polygon into horizontal slices. Thus we

can use the bins based crossings test from section 4.3.4 as a fallback method and we would not even have to compute the index of the bin from the y-coordinate of the point because this may be precomputed and stored in the red rectangle. The bins based crossings test may easily be generalized to also work with rays parallel to the y-axis instead of the x-axis and also to work with rays shot towards either positive or negative infinity along the respective axis. This gives us four different tests that we could use for each red rectangle and we can easily compute during the preprocessing phase which one of those tests is the most promising for the respective rectangle because we can count the number of segments that will have to be considered for rays shot from within this rectangle in each of the four possible directions.

4.4 Performance evaluation

When run with the training data set, the basic crossings test nearly yields the best results. On the one hand the algorithm does not require any preprocessing and on the other hand the algorithm probably benefits from its simplicity: it consists of one loop with a few basic arithmetic operations in its body. The Java virtual machine's Just-in-time (JIT) compiler has good chances optimizing that part of code early and efficiently.

Only two algorithms of those implemented achieve to improve the running time of the basic crossings test. The first of those algorithms is the bucket based crossings test. This algorithm has only a very simple preprocessing phase that takes $O(n)$ time for a region with n segments. Furthermore the algorithm itself is not much more complicated than the basic crossings test and thus has similar prospects of getting optimized by the JIT compiler: it basically splits the one loop over all segments into two nested loops of which hopefully many of the inner loops do not get executed because of a simple conditional expression. Unfortunately the performance gain for the training data is only at about 11% for the time spent in the algorithm and only at about 2% in global program execution time including input, parsing and output.

The other algorithm that gains performance when compared to the basic crossings test is the bins based crossings test. Its preprocessing phase is a little more expensive, especially if the improvements described at the end of section 4.3.4 are being implemented. When querying the data structures previously built during preprocessing, we need only constant time to locate the bin in which the query point falls. Within that bin we essentially run the crossings test with all segments contained in that bin. The algorithm may thus be optimized by the JIT compiler similar to the basic crossings test and still for reasonably shaped regions, many segments should be skipped. As with the bucket based crossings test the performance gain is at about 10% of the time spent for point in polygon queries.

The remaining two approaches lead to longer execution times. They have both more expensive preprocessing phases and worse prospects concerning JIT compiler optimization. For the training data set the interval-tree based crossings test takes about 150% additional time when compared to the basic crossings test and the adaptive-grid based approach needs even additional 250%. It appears as if on the one hand the number of points is not high enough to justify the expensive preprocessing phase. On the other hand the regions consist of only about 150 to 300 segments which is in turn not enough for the strategies of the algorithms to take effect.

To mitigate the effects of the preprocessing phase of the latter two algorithms I tried to implement a hybrid approach, which first computes all candidates of points and regions grouped by regions. Then for each region, depending on the number of candidate points, the hybrid algorithm would choose between say a simple and a more advanced algorithm. Although the idea of this approach was to combine the benefits of two algorithms to form a stronger algorithm, the approach has its drawbacks which outweighed the benefits. On the one hand precomputing and storing temporarily the list of candidates has its overhead which is yet not too big to seem overly significant. On the other hand, when using two separate algorithms,

each algorithm gets used less often so that the JIT compiler will optimize it later. In sum it appears that no performance gain could be achieved for the training data set with this approach, but in the contrary performance dropped a little.

5 The WITHIN-N query

This section describes the algorithms implemented to solve the WITHIN-N query. Input for this query is again a set of regions $\mathcal{R}$ and a set of points $\mathcal{P}$. Each instance has a geometric component and additionally each region instance has a temporal interval of validity and each point instance has an associated timestamp. To find is the set of all pairs $(p, r) \in \mathcal{P} \times \mathcal{R}$ for which geometrically $distance(p, r) \leq N$ is true and for which the point's timestamp is contained within the region's interval of validity.

5.1 Simple solution

Similar to the INSIDE query, the simplest solution appears to be a loop over both regions and points and then to evaluate for each pair $(region, point)$ whether it satisfies the spatiotemporal WITHIN-N-predicate. Algorithm 6 shows pseudocode for this solution.

Algorithm 6 Calculating all pairs in $P \times R$ that satisfy the spatiotemporal WITHIN-N predicate

```
1: for each point p ∈ P do
2:     for each region r ∈ R do
3:         if SPATIOTEMPORAL-WITHIN(p, r, N) then
4:             OUTPUT(p, r)
5:         end if
6:     end for
7: end for

8: function SPATIOTEMPORAL-WITHIN(p, r, N)
9:     return p.time > r.time_start and p.time ≤ r.time_end
10:                and WITHIN-DISTANCE(p_geom, r_geom, N)
11: end function
```

5.2 Using spatial indexing

Again spatial indexes may be used to store the regions to increase query performance when compared to a naive nested loop implementation once the number of regions becomes sufficiently high. Algorithm 7 shows a transformation of algorithm 6 to an algorithm following the *filter-refinement* pattern. The notable difference to algorithm 2 is that we cannot query the spatial index with the query point itself, but instead need to query it with a rectangle that ensures that the index will return all candidate regions whose distance to the query point may be less or equal to N. Using a square with size $2 \cdot N$ centered on the query point as a query rectangle ensures this.

5.3 The within distance problem

The strategies explained above allow us to find efficiently the set of all pairs of points and regions (p, r) that are viable candidates for the satisfaction of the WITHIN-N predicate because p is sufficiently near to the the *bounding box* of r and that the temporal constraint is met

Algorithm 7 Calculate all pairs in $P \times R$ that satisfy the spatiotemporal WITHIN-N predicate using a spatial index to efficiently store and retrieve regions

```
1: index ← initialize spatial index
2: for each region r ∈ R do
3:     index.insert(r)
4: end for
5: for each point p ∈ P do
6:     q ← square around p with size 2 · N
7:     C ← index.query(q)
8:     for each region r ∈ C do
9:         if WITHIN-DISTANCE(p, r, N) then
10:            output(p, r)
11:        end if
12:    end for
13: end for
```

for (p, r) as well. What remains to find is an efficient implementation of the *within distance* problem to decide whether a candidate pair (p, r) really belongs to the result set of a query.

A point p is considered within distance N of a polygon P if it is either *inside* P or if the distance from p to the boundary of P, denoted by ∂P, is less or equal than N. Thus there are two parts of the problem that may be solved separately. On the one hand we need to determine whether the point is *inside* a polygon. This problem has already been considered in section 4.3. On the other hand we need to determine whether there is any segment of P whose distance to p is less or equal to N. I call this problem the WITHIN-DISTANCE-TO-BOUNDARY-problem and algorithms to solve it will be discussed in the following subsections.

5.3.1 Simple solutions

To solve the WITHIN-DISTANCE-TO-BOUNDARY-problem, we can iterate over the segments of the polygon and calculate the distance from the point to each segment. Once we find a segment whose distance to p is less or equal to N we may stop the iteration and know that $distance(p, P) \leq N$. If on the other hand we do not find such a segment, we know that $distance(p, \partial P) > N$ and that whether $distance(p, P) \leq N$ is true depends on the point in polygon predicate. Algorithm 8 shows pseudocode for this.

Algorithm 8 A simple algorithm to determine whether $distance(p, \partial P) \leq N$

```
1: function WITHIN-DISTANCE-TO-BOUNDARY(P, x, y, N)
2:     for each segment s ∈ P do
3:         d ← distance(p, s)
4:         if d ≤ N then
5:             return true
6:         end if
7:     end for
8:     return false
9: end function
```

Algorithm 9 shows how algorithm 8 may be implemented using squared distances to avoid square root computations. The algorithm distinguishes three cases concerning the projection of the point onto the line going through a segment. Depending on the location of the projection of the point either the distance to the startpoint, to the endpoint or to the nearest point on the segment, i.e. to the projection will be calculated.

Now implementations of INSIDE and WITHIN-DISTANCE-TO-BOUNDARY can be combined to

Algorithm 9 A simple algorithm to determine whether $distance(p, \partial P) \leq N$

```
1: function WITHIN-DISTANCE-TO-BOUNDARY(P, x, y, N)
2:     distance^2 ← N · N
3:     for each segment s ∈ P do
4:         squaredLen ← squaredDistance(s.x1, s.y1, s.x2, s.y2)
5:         if squaredLen == 0 then
6:             d ← squaredDistance(x, y, s.x1, s.y1)
7:             if d < distance^2 then
8:                 return true
9:             end if
10:        end if
11:        t ← ((x − s.x1) · (s.x2 − s.x1) + (y − s.y1) · (s.y2 − s.y1))/squaredLen
12:        if t < 0 then
13:            d ← squaredDistance(x, y, s.x1, s.y1)
14:            if d < distance^2 then
15:                return true
16:            end if
17:        else if t > 1 then
18:            d ← squaredDistance(x, y, s.x2, s.y2)
19:            if d < distance^2 then
20:                return true
21:            end if
22:        else
23:            tx ← s.x1 + t · (s.x2 − s.x1)
24:            ty ← s.y1 + t · (s.y2 − s.y1)
25:            d ← squaredDistance(x, y, tx, ty)
26:            if d < distance^2 then
27:                return true
28:            end if
29:        end if
30:    end for
31:    return false
32: end function
```

solve the WITHIN-DISTANCE-predicate. This can be done by first executing INSIDE and then, if the result is not true execute WITHIN-DISTANCE-TO-BOUNDARY as shown in algorithm 10. Alternatively we can also execute both subprograms in swapped order as shown in algorithm 11. At least on the training data the first approach has been found to be more efficient.

Algorithm 10 Determining whether $distance(p, P) \leq N$

```
1: function WITHIN-DISTANCE(P, x, y, N)
2:     if INSIDE(P, x, y) then
3:         return true
4:     end if
5:     return WITHIN-DISTANCE-TO-BOUNDARY(P, x, y)
6: end function
```

Algorithm 11 Determining whether $distance(p, P) \leq N$

```
1: function WITHIN-DISTANCE(P, x, y, N)
2:     if WITHIN-DISTANCE-TO-BOUNDARY(P, x, y) then
3:         return true
4:     end if
5:     return INSIDE(P, x, y)
6: end function
```

5.3.2 Using spatial indexing for segments

Similar to the approach used in section 5.2 I tried to use spatial indexing to speed up the execution of the WITHIN-DISTANCE-TO-BOUNDARY procedure. In a preprocessing phase we insert all segments of a polygon P into a spatial index. When testing the distance of segments of P to a query point p we can exclude all those segments whose bounding box is so far away from p that it is impossible for the segment to be within the specified distance. As in section 5.2 we cannot query the index with the query point itself but need to use a square of size $2 \cdot N$ as a query rectangle to be sure that all viable segments of P will be returned from the index. For each segment found in the spatial index, we perform the same steps as seen in the previous subsection, resulting in a semantically equivalent algorithm.

Although this method could in principle lead to reduced execution time, it does not for the training data set. It appears that the number of segments of the polygons is too low for the algorithm to benefit from this technique. Although the execution times with this technique are longer than with the simple approach (but only a little bit longer) I assume that for polygons with a much higher number of segments this technique would be beneficial.

5.4 Performance evaluation

For the within distance problem the fastest algorithm implemented is the simple solution described in section 5.3.1 which first executes the crossings test to determine whether the query point is within the region and if that test returns false determines whether the point is sufficiently near to the boundary of the region by iterating its edges. Any attempt to reorder parts of that algorithm, i.e. swapping *point in polygon* test and *within distance of boundary* test, or building an interleaved version that combines both loops over all edges into a single loop led to longer execution times in total.

Building a spatial index for quick access to edges of regions did also not have a positive effect but instead the running time for the relevant part of the program became about 20% longer. Probably the number of edges of the regions is with 150 to 300 not high enough for the indexing to have positive effects.

6 Outlook

The point in polygon test developed in section 4.3.5 could be studied in more detail: on the one hand its correctness should be proven and its running time could be analyzed for different types of polygons and for polygons of different sizes. On the other hand the algorithm requires as input a parameter to determine the granularity of the polygon decomposition. It is hitherto unclear how to choose this parameter wisely depending on the characteristics of the polygon or the expected number of queries that are going to be performed. Also there are many ideas for improvements of construction of the data structures as well as more efficient solutions for the fallback point in polygon strategy for points that fall into the region near the boundary where no approximation with rectangles has been created during preprocessing.

Haines[Hai94] discusses more point in polygon strategies than those considered in this report. One of those is an approach that is based on the *triangle fan test* which originally is a point in polygon test for convex polygons. There an extra vertex is chosen that lies inside the polygon which is used to construct a triangle for each edge of the polygon. Now a binary search on the start and end angles of the triangles lets us find quickly the only triangle that a query point may possibly lie within with which we can determine whether the query point is inside the polygon. This method may be generalized to work with general polygons as well, where the number of triangles overlapping a query point has to be counted instead of locating a unique triangle. It would be nice to see this algorithm's performance in practice.

Another strategy discussed in [Hai94] is a method that overlays the bounding box of the polygon with a regular grid and precomputes the inside predicate for a corner of each cell in the grid. Now the cell that a query point falls within can be computed in constant time similar to the bins based approach, where the bin can be located in constant time. Once the grid cell is located we shoot a ray from the query point, but only check for intersections with edges of the polygon that in turn intersect with the grid cell where the set of these edges have been precomputed for each cell during preprocessing. The drawback of this approach that it is relatively space-consuming, depending on the granularity of the grid. It would be nice to evaluate the algorithm's performance when compared to the adaptive grid method and also to compare the space complexity of both methods.

References

[Hai94] Eric Haines. Graphics gems iv. chapter Point in polygon strategies, pages 24–46. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[RSV01] Philippe Rigaux, Michel Scholl, and Agnes Voisard. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

[Shi62] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Communications of the ACM*, 5(8):434, August 1962.