Trixel Buffer Logic for I/O Bound Point in N-Polygon Inclusion Tests of Massive Bathymetric Data

João Fradinho Oliveira[1] Marek Ziebart[2] Jonathan Iliffe[3] James Turner[4] Stuart Robson[5]

[1]C3i/Instituto Politécnico de Portalegre, Portalegre, Portugal. jfoliveira@estgp.pt

[2,3,4,5] Department of Civil Environmental and Geomatic Engineering, University College London, U.K.

{m.ziebart, j.iliffe, james.turner, s.robson}@ucl.ac.uk

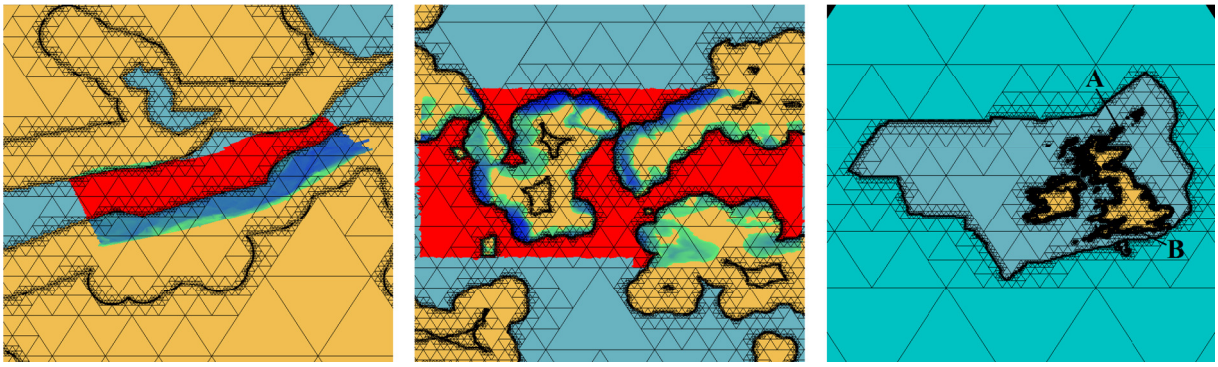

Figure 1. I/O bound point in N-polygons inclusion queries - "Valid" points in hues of blue/green (colour coding the sea floor depth) are inside the project polygon limit area AND outside the offset coastal inner polygon AND inside the offset costal outer polygon. "Invalid" points in red are inside the project polygon, outside the offset coastal inner polygon BUT outside the offset coastal outer polygon. Left: result of 1.757 billion bathymetric point queries (Solent[B]1h36m DuoCore2.5Ghz, details in Section4) Center: 61 million queries (Kirkwall[A]~ 3.36min). Right: project polygon limits in dark blue and quadtree root in light blue.

ABSTRACT

"Trixel Buffers is a new spatial data-structure for fast point in multiple polygon inclusion queries. The algorithm utilizes a pre-processing step in which the inside/outside status of a quadtree´s leaf triangles without polygon geometry is pre-computed automatically; at run-time point queries lying within these triangles simply inherit their inclusion status. If a point query lies in a leaf triangle enclosing polygon vertices or crossing edges, a ray is fired from the point towards the triangle center whose polygon inclusion properties has also been pre-computed: rules are then applied to the intersection count and center-point properties to infer the polygon inclusion status. Our main contribution is that rays need not be followed until the polygon limits, and consequently the algorithm is I/O bound with shallow trees. It took 1h36m rather than days of using a standard ray test to determine the multiple polygon (~270,000 line segments) inclusion of 1.75 billion points on a 2.5GHz DuoCore computer.

Keywords

Point in polygon test, point-location problem, trixel buffer logic, point buffers, bathymetric data.

1. INTRODUCTION

The seemingly simple task of determining whether a point is inside or outside a given polygon, or indeed a set of several polygons at the same time is an integral task within many diverse applications, such as a geometric editing/polygon selection, climate simulation, and interactive computer graphics. A popular solution used for example in ArcGIS is simply to render polygons and lookup the rendered pixel attributes (polygon IDs) at our mouse click position. This approach works well by relying on the user and user interface to zoom interactively to a high level of detail in a local region of interest if they wish more accuracy in their polygon selection. Unfortunately, this strategy would not work well in the context of determining the point-in polygon status of large volumes of data/query points from a file such as bathymetric survey data from the sea floor as the user cannot afford to set the zoom level/center the view location for each point to obtain accurate enough results from rendering. Furthermore, high detail polygons can rapidly make such a rendering

approach for point polygon inclusion cumbersome; in addition files containing the point queries can be streamed from different users over a network and be of arbitrarily locations, making the position and zoom levels of such renderings difficult to optimize.

Several algorithms model polygon geometry directly offering resolution independence and higher accuracy results. A ray test [Tay94a, Prep85a] handles convex and concave polygon but without a hierarchical spatial tree will be comparatively slow as is the winding number method that still needs to test/reject several polygon segments. Quadtrees [Sam90a] can be used to solve the point-location problem rapidly [Sar86a], [Edel86a, Kir83a, Pat06a], and depending on the application scenario one could use rectangular quadtrees [Pov04a], or triangular quadtrees [Fek90a] if mapping the whole globe. To avoid distortions at the poles, each of the 20 equilateral triangles of an unrolled icosahedron is a root quadtree (Figure 2).

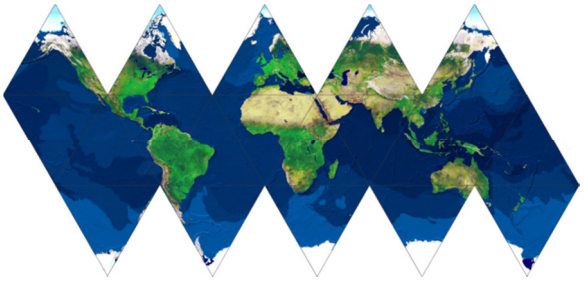

Figure 2. Icosahedron, 20 equilateral root triangles. Courtesy [Ham12a].

The idea of solving point-in polygon queries by using quadtree leaf cells to buffer and cache pre-computed polygon-inclusion results is not new [Pov04a], Trixels [Fek90a]. However several problems arise when addressing point-in polygon queries of massive bathymetric data using complex high detail polygons. The first problem is memory consumption, which scales badly with increasing the vertex spacing resolution of the polygons. I/O bound performance of reported strategies for finding the polygon inclusion status of point queries near polygon geometry require that the tree was subdivided until the size of leaf cells matches the resolution of the polygons. If considering applications running on mobile devices with even more limited memory this problem is even greater. Secondly, existing methods require the manual setting of a known interior polygon position to propagate results during the pre-processing stage. When dealing with thousands of offset polygons in the context of our application this becomes cumbersome or impractical to ask the user, as the size of many polygons derived automatically from real data can be very small (Fig 1, center). Thirdly our application must be robust to deal with convex and concave polygons with arbitrary vertex ordering resulting from merging/importing of several polygon shape files from different tools.

In this article we present a hierarchical spatial database (Trixel Buffers) solution to these problems, where leaf triangles void of geometry are termed trixel buffers; point queries lying in a trixel buffer simply inherit the polygon-inclusion status of that triangle (for example triangle T(in) and T(out) in Fig.3. In contrast a point query (Q) lying in a leaf triangle that has polygon vertices or crossing edges T(test) requires an additional ray test with the known polygon inclusion point buffer P(out). Trixel Buffers were designed as part of the Vertical Offshore Reference Frame (VORF) project [Ili06a], which modeled the datum surfaces used for spatial data on land and at sea around the UK and Ireland. The work carried out enables transformations between datums used by satellite positioning systems, marine datums used for bathymetric data, and land datums used for topographic data. With 17 different land datums and multiple complex polygons defining navigable rivers and harbors, robust and efficient position tests need to be performed on each point in very large datasets.

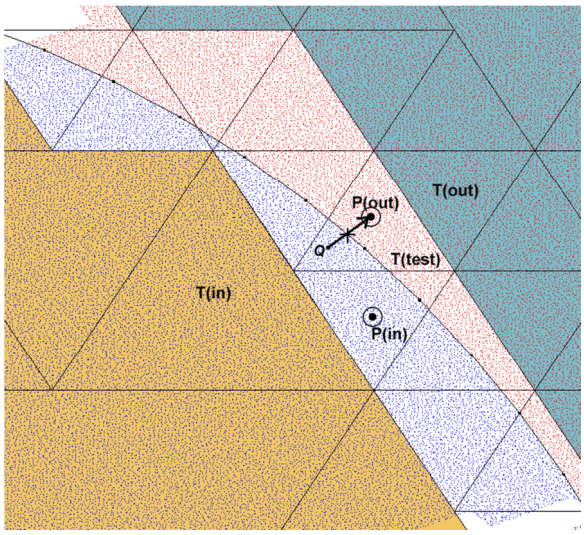

Figure 3. Trixel buffer logic.

The main concept of our algorithm is in the case of a query point that lies in a leaf triangle with polygon geometry, to cast a ray from the query point to its triangle center and apply trixel buffer logic to the center point´s pre-computed polygon inclusion status and the number of intersections of the ray with any polygon geometry in the leaf triangle to infer the inside/outside status of the query point, rather than continue to trace a ray until it exits the polygon or quadtree.

Contributions:

We present a new point in N-polygon algorithm that:
- Extends the Gauss-Jordan theorem to work efficiently with hierarchical spatial trees.
- Is I/O bound with massive data sets, requiring relatively shallow trees.
- Extracts interior/exterior of polygons automatically.

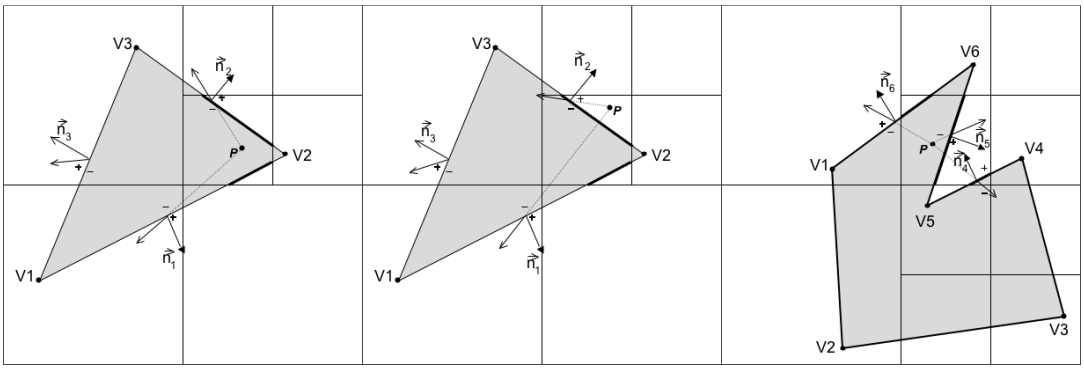

Figure 4. ***Inner product with convex polygons (left & center) and with concave polygons (right)*****. left: BOTH inner products with n1 and n2 are positive, P is inside; center: ONE OF the inner products with n2 is negative,P is outside. right:ONE OF the inner products (n4) is negative and P is incorrectly labeled as out.**

- Handles convex and concave polygons with arbitrary ordering.
- Is extendable to other dimensions.
- Can be used to extend existing quadtree methods by calculating the polygon inclusion status of point buffers (which are infinitely small trixel buffers) and applying our presented rules.

We briefly review related work in Section 2. We note that we use the words 'trixel buffer' and 'triangle nodes' interchangeably throughout the paper, and the words 'coastal line' in reference to 'polygon lines that represent real coast lines'. In section 3, we show how the quadtree is used to create trixel buffers automatically, and how our new short ray-strategy is used in conjunction with trixel buffers to determine which polygons a point lies within. In section 4 we present results; specifically we use a brute-force ray algorithm [Tay94a] to inspect and validate our polygon inclusion results; we also show that our method does not require the resolution of leaf triangles to match the resolution of the input polygons to achieve I/O bound performance. In section 5 we present a discussion, and conclude in section 6.

2. PREVIOUS WORK

A vast body of literature exists in computational geometry for the detection of whether a point is inside a convex hull, or inside a generic planar polygon [Berg97a, Prep85a, Hai94a, Edgel86a]. Perhaps the simplest way to determine whether a point is inside or outside of a polygon is to fire a ray horizontally from the point in question to + or - infinity and apply the Jordan curve theorem on the number of intersections of the ray with the edges of the polygon. If the number of intersection is odd, the point is deemed inside, if the number is even the point is deemed outside. Special care [Prep85a] is taken for rays that pass through the vertices of a polygon, horizontal edges are ignored, and if the ray intersects a vertex, and the vertex has the largest ordinate of the edge the intersection is counted, otherwise it is ignored. Even though many polygon edges can be trivially rejected from any intersection by checking if both ordinates of the edge vertices are both greater or both smaller than the ray's ordinate, this algorithm has a query complexity of O(N) as it checks every edge of a polygon before determining to test it for intersection or not. Fast solutions exist for the planar point location problem: given a planar subdivision of space, the task to establish which cell or polygon contains our query can be achieved in O(log(N) using persistence search trees [Sar86a], fractional cascading [Edel86a], and triangulation refinement [Kir83a]. Recently sub-logarithmic complexity for queries has been achieved with support for dynamic planar subdivisions [Pat06a]. In particular Kirkpatrick [Kir83a] shows O(N log(N)) preprocessing time with O(n) storage using hierarchical triangle subdivisions. Similarly we use a triangular quadtree in this paper for the point location problem, and modify it to support a ray strategy for solving the point-in polygon problem. Poveda et al. [Pov04a] use a square quadtree to buffer the polygon inclusion status in cell nodes void of geometry, and report I/O bound results with quadtree leaf buffers whose length matches the vertex spacing resolution of a convex polygon set. For query points in a cell with geometry, they use an inner product test (Fig.4-left&centre), but to our understanding unfortunately this test will not work in the case of concave polygons (Fig.4-right).

This method also requires the manual seeding of a known interior point. As mentioned earlier Fekete [Fek90a] uses 20 equilateral triangles of an icosahedron as root nodes of triangular quadtrees instead of rectangular quadtrees to avoid distortions at poles [Ran02a, Oli06a]. Fekete stressed the need to combine a spherical visualization representation with the actual data coordinates for global simulation of the atmosphere [Ran02a]. Hence the length of the position vectors, defined by the triangle edge midpoints are adjusted during subdivision to a set radius or property, thus creating a spherical quadtree.

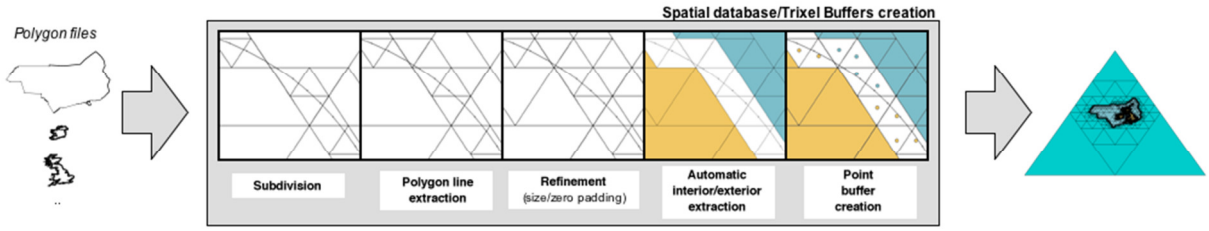

Figure 5. Overview of the construction of the Trixel Buffer spatial database.

Fekete does not address directly point-in polygon tests in his data structure, but uses a technique called connected component labeling that uses connectivity information stored in a tree node to access and propagate the inside or outside results of cells to adjacent nodes. A seed cell is manually chosen inside a landmass, and the result is propagated to the limits of the landmass.

	[Tay94a]	[Fek90a]	[Pov04a]	Trixel Buffers
Manual (m) / automatic (a) interior extraction	a	m	m	a
Convex Polygons	y	n/a	y	y
Concave Polygons	y	n/a	n	y
Tree depth for I/O Performance w/ polygon data of Fig.1	n/a	n/a	16	12

Table 1. Comparison of properties of existing methods for point in polygon tests of massive bathymetric data.

One can envisage that points inside cells void of geometry simply inherit the extracted landmass IDs. Point queries inside cells with geometry could use the point buffers presented in this paper to determine landmass inclusion. Table 1 compares the properties of existing methods for massive bathymetric tests.

3. TRIXEL BUFFER CREATION

Our system is built with five main steps (Fig.5, middle). The first step creates a hierarchical spatial database through quadtree triangular subdivision. This step solves the point location problem. The second step termed polygon line extraction adds further triangles in areas of passing edges to ensure that all polygon segments can be readily referenced. Further refinement of the tree in step 3 maximizes the area void of geometry. In step 4 the polygon inclusion status of triangles void of geometry is computed automatically through the interior/exterior extraction algorithm. Finally in step 5 the center point inside each leaf triangle with geometry has its interior/exterior status calculated.

Subdivision

Although we use a triangular quadtree rather than squares, the process of subdivision is similar. In the context of our project, one of the twenty base equilateral triangles of the icosahedron sufficed to enclose our survey data. The construction of the quadtree starts with a simple O(N) pass on every polygon vertex in order to establish the maximum and minimum coordinates of the set. An equilateral triangle which encloses all the data can then be computed. A subsequent subdivision process, using the three middle points of each of the triangle's edges, creates four smaller equilateral triangles. In our implementation, for precision purposes the width and the height of the root triangle are stored once separately. The width and height of a quadnode of any level can be calculated by a single division made to the original width/height of the root triangle by powers of two representing the subdivision level.

This process of subdivision is repeated until the set maximum depth level is reached. In order to maximize the number of query points not requiring geometric tests at run-time, we first compute the average vertex spacing distance of the polygon set, the limit of the subdivision depth can then be set by the length of the triangle edge being subdivided. If the length is smaller than the average spacing we stop the subdivision. Standard recursive spatial data structures such as the quadtree set recursion limits such as the maximum number of input points allowed in the deepest triangles and/or the maximum tree depth allowed. Unfortunately this strategy can yield leaf node triangles that are very large. We record the width and height of the smallest quadnode found, and in a second pass, we further refine leaf nodes that have geometry if their size is larger than the smallest triangle found.

Polygon line extraction and refinement

Whilst the subdivision step is centered on the polygon vertices, a water tight front of leaf nodes covering the complete geometry, including crossing edges is required before rays from any position in the tree can reliably test the polygons for inclusion, independently of the spacing of the vertices of the polygon. To detect crossing edges, and minimize the size of these areas, we insert leaf nodes in the tree in

areas covering long edges; in addition these nodes/triangles also store the ID of one of the vertices of the long edge, this ID can then be used later to build the edges for ray intersection tests. To build this connected front a procedure with a result similar to that of Bresenham´s line drawing algorithm is performed, starting by looking up the leaf nodes containing each endpoint of each polygon edge. We refer to the leaf node containing the first vertex of an edge as the starting leaf node, and the leaf node containing the endpoint vertex as the target leaf node. If the two vertices are contained in the same leaf node, no leaf nodes need be inserted, otherwise we systematically insert leaf nodes adjacent to the node containing the starting vertex of the edge, until the inserted leaf node is adjacent to the leaf node containing the end vertex of the edge. Here adjacency requires that the leaf nodes share two vertices.
Since we do not store adjacency/connectivity information, we retrieve adjacent triangles covering the polygon edge, using a three step process.

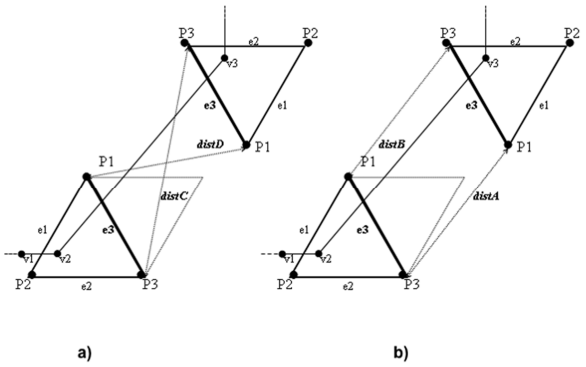

Figure 6. Minimum sum distance between leaf triangle edges. The starting leaf triangle containing the first vertex V2 of the edge V2-V3 has minimum sum distance (distC+distD(a)>distA+distB(b)) between the triangle edge e3 of the start triangle and the edge e3 of the target triangle.

The first part establishes which edge from the starting leaf node has the smallest summed vertex distance to any edge of the target leaf node (Fig.6-a&b)). However, if one were to insert an empty node/triangle adjacent to the edge simply determined to be of shortest distance to the target triangle, no guarantee could be given that the polygon edge was completely contained by the inserted leaf nodes (Fig.7-left). This distance calculation is however useful to determine the stopping condition of zero distance between adjacent and target triangles.
The second part of the algorithm finds which edge(s) of the starting triangle are candidates for inserting the adjacent empty leaf node. The third part determines which of the potential two candidates is chosen to insert the adjacent node. A polygon edge can at most intersect a triangle in two of its three edges. For example Fig.6-a) shows that the edge v2-v3 intersects the starting triangle in one edge (e3), and intersects an adjacent triangle in two edges.

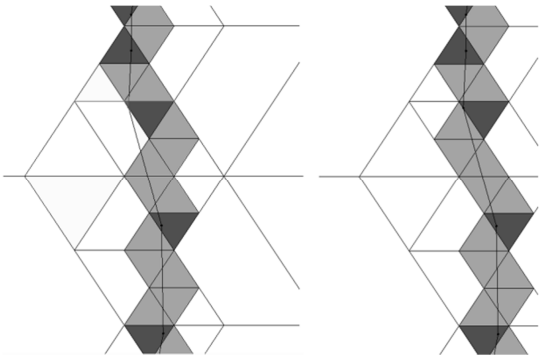

Figure 7. Polygon line extraction. - Dark grey triangles contain vertices of the polygon. Light grey triangles on the left image denote incorrectly inserted empty triangles along the edge using the minimum sum distance to the target triangle, light grey triangles on the right image were inserted using the polygon edge geometry (Fig.8) instead.

We note that once the leaf nodes of the polygon edge vertices are found (recursive look-up of the tree), the line connecting the polygon edge vertices is guaranteed to intersect both the starting and target leaf node triangles if the vertices are in different leaf triangles (if adjacent no triangles are inserted, if separated by an area void of leaf nodes, new empty leaf triangles are inserted in the path of the edge, if a leaf node is already present in the path, the ID of the passing edge is stored in the node instead) or can have its two vertices in the same leaf triangle (start and target triangle are the same, no adjacent triangle needs to be inserted for this vertex pair as any passing ray will retrieve both edges connected to each vertex). In order to narrow down the candidate edges to two, we determine which triangle edges the polygon edge might intersect. Specifically we compute the signed distance of each vertex of the triangle to the polygon edge, the triangle edges whose vertex distances have different signs indicates that the triangle edge is intersected by the line going through the polygon edge. Fig.8-a) shows that the triangle edge P1-P3 and the triangle edge P2-P3 have different point distance signs, although only the edge P1-P3 is intersected by the polygon edge V2-V3. The final step of the coastal line extraction algorithm is to determine which of the two candidate edges, is the edge in which the system is going to insert an adjacent empty node. For this effect we use an efficient ray rejection technique [Xu03a], where the endpoints of this polygon edge are tested against the candidate edges. The triangle edge that yields different signs in the edge endpoint distance test is selected. For example, the triangle edge P1-P3 of Fig.8-b) is the only edge with different distance signs.

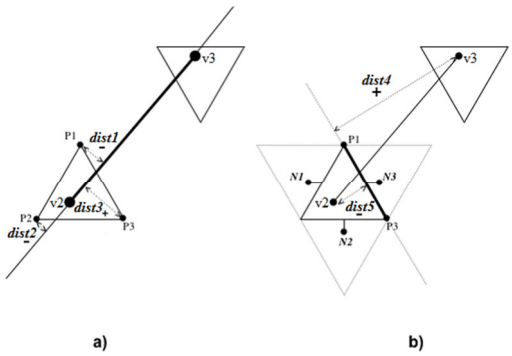

Figure 8. Polygon line extraction using the polygon edge geometry.

Finally we add the ID of the first endpoint of the polygon edge to the adjacent empty node (this enables for point queries in the adjacent triangle to retrieve and build the edge that crosses it), if there was an adjacent node already with geometry the vertex ID is added to it nevertheless. The adjacent triangle then becomes the starting triangle. The polygon edge is used as before with the new triangle, however the starting vertex (v2) used for the step (Fig.6-a) and b)) becomes the offset point (N3 of Fig.8-b)). This process is repeated until the starting triangle is the same as the target triangle or is adjacent to the target triangle (Fig.7-right)). To make sure any point query location has a non-overlapping leaf node, a refinement procedure is done, specifically, every node from the root is checked to see if a subnode exists with adjacent null node pointers, new empty triangle nodes representing smaller areas void of geometry then replace the null pointers.

Automatic interior/exterior extraction

Most point in polygon methods require manual seeding of a node known to be interior to the polygon. Our algorithm calculates automatically the polygon inclusion status of nodes void of geometry using the following rule:

Rule 1 – Any point within a node void of geometry has the same polygon inclusion status as any other point within the node. If one fired a ray from any point of the node towards infinity in any direction, and the node was indeed completely interior to a polygon, then the Gauss Jordan rule will give the same result for that polygon, independent of the number of intersections. Special care is taken for rays that pass through the vertices of a polygon, horizontal edges are ignored, and if the ray intersects a vertex, and the vertex has the largest ordinate of the edge the intersection is counted, otherwise it is ignored.

Connected adjacent leaf nodes void of geometry are grouped and the inside/outside status of one of the cells in the group is calculated with rule 1, the result is then copied to all the other elements in the group using following rule:

Rule 2 – Rule 1 can be generalized to the outline of triangle strips void of geometry or more arbitrary connected contour shapes, as long as the single ray being fired does indeed traverse to infinity or beyond the known pre-calculated bounds of the polygons being tested.

We use a flood-fill algorithm to group connected adjacent triangles (Fig.9-left)) as follows: we build an array of pointers to all the leaf nodes void of geometry in the tree, and we zero a group counter and the group attribute of each leaf. We sort the array in increasing triangle size (a small triangle will always have only one adjacent neighbor on one side, rather than two or more if starting with a larger triangle first), and visit each element of the array that has no group assigned, we increment the group counter each time we find a triangle that has not had a group ID assigned and assign the current group number to the triangle, we then retrieve the three adjacent triangle neighbors (left, right and vertical) of the triangle. If one of the neighbors has no geometry and has not been assigned a group, we assign the current group counter value to it and push it on to a stack. While the stack is not zero we keep removing the last element of the stack before proceeding with the next element of the sorted array. Once all leaf triangles of the array have been visited, we sort the array again this time on group ID, the first leaf node of a group found in the array is then ray tested as described below and the result is copied to all the array element of the same group. As mentioned earlier, each connected group has one leaf node void of geometry ray tested, a horizontal ray from the middle of a triangle is fired towards a point outside the tree. If the ray passes a triangle that has polygon vertices or flagged geometry, tests for intersection are carried out on polygon edges. Specifically each vertex contained or flagged in a triangle is looked up, and two edges are built by retrieving the two other connected vertices. These connected vertices can lie arbitrarily far from the ray path, and the ray might only intersect the edges within adjacent triangles. In order to not test an edge more than once during the course of the ray path and to keep track of whether an edge has already been counted as a hit, each vertex of our polygon set has two flags reflecting the intersection status of the two edges that connect to it. These flags can have a mark of 0, 1 or 2, depending on whether the edge has not been tested (0), tested and hence not requiring further tests (1), or tested and with a hit that has been counted (2). If a ray crosses a triangle with no geometry, that triangle is skipped, and the next adjacent triangle in the ray path is retrieved, until the ray exited the tree. All the nodes that were visited have the flags of their vertices

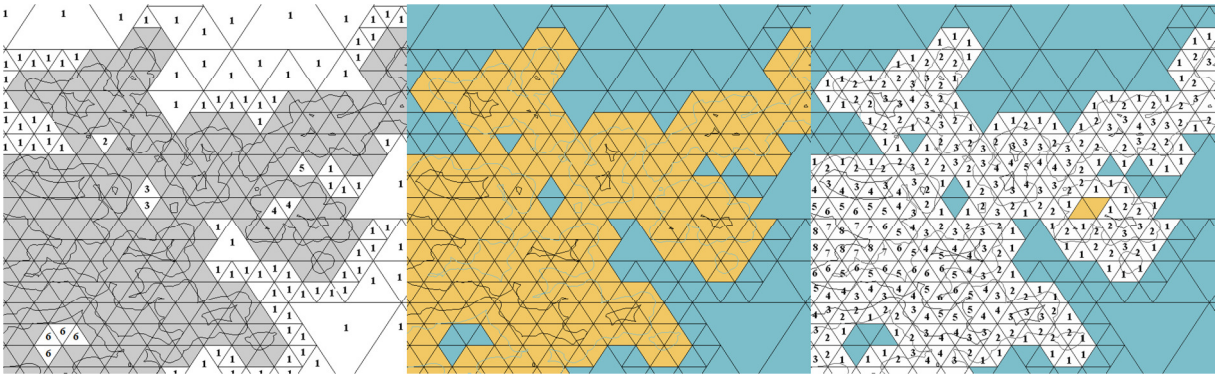

Figure 9. Left: Flood fill grouping of triangles void of geometry, numbers represent each connected group. Center: Interior/exterior extraction results. Right: Pre-computation for point buffer creation, numbers represent the smallest number of triangles that need to be traversed to find an adjacent triangle void of geometry, this will be used to establish the shortest route for a ray when determining the point buffer interior/exterior status.

zeroed, before proceeding with any more triangle/polygon inclusion tests. During ray tests a separate intersection count is kept for each polygon ID the ray crosses. If the final intersection count for a particular polygon ID is odd, the starting selected triangle is deemed to be inside that polygon, and the ID of that polygon is registered in the triangle. If the count is even, the triangle is outside of that polygon, and the polygon ID for that count is not stored. Note that it is possible for a triangle to be inside more than one polygon, or to be outside a tested polygon but inside another polygon. Fig.1-c) and Fig.9-centre) shows extracted landmasses in brown that are all within a sea polygon in dark blue. It is interesting to note, that since we always retrieve the two edges that are connected to a vertex our ray intersection is not adversely affected in the presence of clockwise and counter clockwise polygons. Any duplicate polygon would have different polygon IDs, and hence does not inadvertently affect the inside/outside counting test.Table 3 shows spatial and memory statistics of Trixel Buffers. We note that the average vertex spacing of our 270,000 polygon edge data set is 136 meters and that this spacing is almost the same everywhere. A strategy whose run-time performance relies exclusively on minimizing the number of geometric primitives to test on cells with geometry would require a tree of depth 16 with leaf edges of 100 meters to have only one vertex to test at run-time. We show that by computing point buffers inside leaf nodes of depth 12 and applying our rules described below, our algorithm is already I/O bound.

Point buffer creation and point in N-polygon inclusion query

At run-time, when testing the polygon inclusion status for point queries lying inside a node with geometry, it is not optimal to carry the same procedure of determining the polygon inclusion of a group, following a ray to the outside of the quadtree each time. One could stop counting intersections when the ray reached the bounding box limits of the polygon of interest, however this can involve traversing several triangles with and without geometry. A faster strategy is to stop the ray when the ray enters a stable node (void of geometry) and apply the following additional rules (3,4,5,6,7) to infer the polygon inclusion status (please refer to Fig. 10 bottom):

Rule 3 – Given an even or zero number of intersections for a polygon ID and the stable node entered is classified as being outside that polygon, the point query is deemed to be outside that polygon (rays starting from red points in node B, reach the outside stable node C).

Rule 4 – Given an odd number of intersections for a polygon ID and the stable node entered is classified as being outside that polygon, the point query is deemed to be inside that polygon (rays starting from blue points in node B, reach the outside stable node C).

Rule 5 – Given an even or zero number of intersections for a polygon ID and the stable node entered is classified as being inside that polygon, the point query is deemed to be inside that polygon (for illustration purposes, rays fired towards the left and starting from blue points in node B, test geometry in B and traverse 3 subnodes of the same size with geometry to reach a large inside stable node to the left of node E).

Rule 6 – Given an odd number of intersections for a polygon ID and the stable node entered is classified as being in that polygon, the point query is deemed to be outside that polygon (rays starting from red points in node B, reach a large inside stable node to the left of node E).

We have so far considered polylines whose vertex spacing is similar to the resolution of the tree. During the landmass extraction process, our horizontal rays were guaranteed to traverse the quadtree to a point outside it, therefore intersecting any polygons in its path. For shorter rays in our point queries, we need to cater for polylines with a vertex spacing that is greater than the edge length of the leaf nodes of the tree (Fig.10-top).

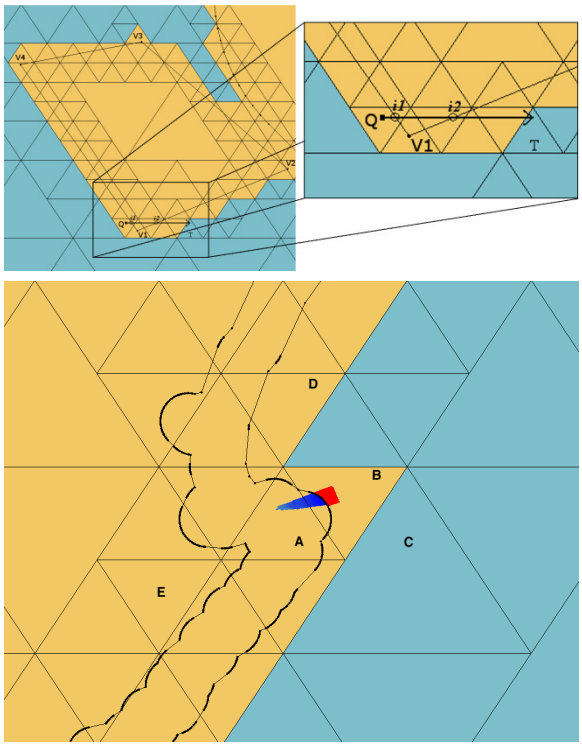

Figure 10. Top: Rule 7. Bottom: Rules 3-7.

Specifically the ray test retrieves the edge information of a node flagged with geometry or containing a vertex, and tests the edge for intersection. Since the ray is no longer guaranteed to traverse to the outside of the tree, the intersection point with the edge is tested to see if it is inside the triangle with the ray segment being tested. If the intersection is outside, it is not counted, and is instead only counted in the triangle where the ray intersects the edge. This then allows one to use the following rule to determine the polygon inclusion set of the query:

Rule 7 – the polygon inclusion set of a pre-calculated node whose ray traversed to the outside of the quadtree, can be re-used in the counting of any horizontal ray test that shares its original ray path.

Still, it is not optimal to cast a ray from a point query towards a node void of geometry, as adjacent triangles still need to be retrieved along the ray path. Instead we pre-compute the multiple polygon inclusion of the center position (point buffer) of every leaf triangle with geometry. At run-time point queries inside leaf nodes with geometry simply cast a ray from their position towards their triangle center position and rules 3-7 are applied to infer the polygon inclusion status. When pre-computing the inclusion status of a point buffer we apply rule 8 to allow the ray to follow the shortest path (Fig.9-right) to a stable node with known inclusion status and apply rule 9. We note that the presented rules for rays have similarities with the process of finding the topological genus of an object in that they require the object to be bended in space to test for equivalence.

Rule 8 – the polygon inclusion set of a pre-calculated node (buffer status) void of geometry whose ray traversed to the outside of the quadtree, can be re-used in the counting of any ray from any direction, in other words ray segments can change direction, and the counting of local intersections enclosed in each traversed triangle that the buffer status can still be used in the end to infer inclusion for the ray test as a whole.

Rule 9 – In the limit of the subdivision, nodes void of geometry are point buffers with surrounding triangles void of geometry.

Figure 9-right shows the smallest number of triangles that need to be traversed from that triangle to find an adjacent triangle void of geometry. These numbers are computed as follows, we create an array only with pointers to leaf nodes with geometry, and access each node (A) in the array and retrieve the triangle´s three adjacent neighbors, if one of them is void of geometry (B) then we mark A with the distance 1, and we record in A which of its three edges leads to B (this is the ID from 1-3 that is used for retrieving the adjacent triangle that is the shortest ray path later in the point-buffer status calculation), if all neighbors had geometry we leave A untouched, we do the same with the remaining nodes in the array. We then increment the distance to 2 and look in the array to find any unmarked node (A) that has an adjacent neighbor (B) with a marked distance of 1, if there is we mark A with a distance of 2, hence working backwards and inwards. The process ends when there was no change made when iterating through the array.

4. RESULTS

Timings for the construction of the various phases of Trixel Buffers are given in Table 2. Memory statistics are given in Table 3. We tested Trixel Buffers with 3 different bathymetric datasets of different sizes (Table4). All the results in our article were carried out on a 2.5Ghz Duocore machine with 2GBytes of RAM, except if indicated otherwise. Figure 1, shows the Point in N-polygons

level	Subdivision	Subdivision/ leaf size refinement	Polygon line extraction	Refinement/ zero filling	Automatic Interior/ Exterior extraction	Calculate minimum distance 2 void node for point buffers	Create point buffers	Total time
1	<1s	<1s	<1s	<1s	<1s	<1s	<1s	1s
2	<1s	1s	<1s	<1s	<1s	<1s	<1s	2s
3	<1s	1s	1s	<1s	<1s	<1s	<1s	2s
4	1s	1s	1s	<1s	<1s	<1s	<1s	3s
5	1s	2s	1s	<1s	<1s	<1s	<1s	4s
6	1s	2s	1s	<1s	<1s	<1s	<1s	4s
7	2s	1s	2s	<1s	<1s	<1s	1s/	5s
8	2s	2s	1s	<1s	<1s	<1s	1s	6s
9	1s	2s	2s	<1s	<1s	<1s	1s	6s
10	2s	1s	2s	<1s	<1s	<1s	1s	7s
11	2s	2s	2s	<1s	2s	<1s	2s	10s
12	2s	2s	3s	1s	6s	2s	2s	17s
13	3s	2s	3s	1s	12s	3s	6s	30s
14	3s	2s	5s	1s	29s	5s	12s	58s
15	3s	2s	8s	1s	1m07s	12s	24s	1m58s
16	3s	2s	15s	1s	2m26s	26s	48s	4m04s
17	3s	2s	28s	3s	**5m24s**	55s	1m37s	8m40s

Table 2. Trixel Buffers construction timings.

inclusion queries results using a combined polygon set of 270 000 vertices (convex and concave with av. spacing of 136 meters). left: result of 1 757 billion bathymetric point queries (Solent) in an area of 9.7x6.8 km, the point in N polygons inclusion test took 1h 36 min (I/O bound) with a new spatial tree depth 12 (leaf node size 1600 meters) rather than depth 16 (100 meters) for equivalent performance of competing methods and assuming only convex polygons in the later; center: 61 million bathymetric points (Kirkwall) in an area of

Sub-division Level	# non-leafs	# leafs	# leafs with geometry	# leads without geometry	Node width corresponding geodetic (Km)	size RAM (MB)
0	0	1	1	0	6026.067	<1
1	1	4	2	2	3225.863	<1
2	3	10	6	4	1637.735	<1
3	9	28	14	14	821.909	<1
4	23	70	33	37	411.333	<1
5	56	169	83	86	205.713	<1
6	139	418	212	206	102.862	<1
7	351	1,054	520	534	51.432	<1
8	871	2,614	1,323	1,291	25.716	<1
9	2,194	6,583	3,415	3,168	12.858	1
10	5,609	16,828	8,222	8,606	6.429	2.4
11	13,831	41,494	20,143	21,351	3.214	6
12	33,974	101,923	48,327	53,596	**1.607**	14
13	82,301	246,904	100,744	146,16	0.803	35
14	183,045	549,136	206,843	342,293	0.401	79
15	389,888	1,169,665	419,403	750,262	0.200	169
16	809,291	2,427,874	844,367	1,583,507	0.100	352
17	1,653,658	4,960,975	1,694,424	3,266,551	0.050	721

Table 3. Trixel Buffers memory consumption.

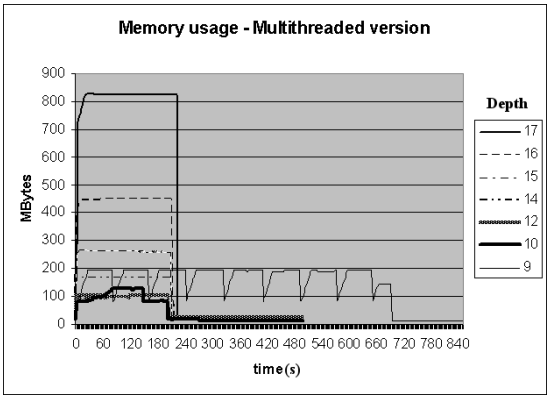

Figure 12. MT(read&queries)performance using a fixed 120MB read buffer and different depth tree.

19x16 km, the point in N polygons inclusion test took ~3.36 mins with the same tree; right: the root of the quadtree in light blue project is one of

example1 - 170315pts **(4.7KB)**					**Kirkwall** - 61068103pts **(1.8GB)**				**Solent** - 1757890806pts **(54GB)**			
valid/inv. 104391v 65924i					23873984v 37194119i				669641232v 1088249574i			
	Time (MT)	**Time (ST)**	**buff.**	**unbuff.**	**Time (MT)**	**Time (ST)**	**buff.**	**unbuff.**	**Time (MT)**	**Time (ST)**	**buff.**	**unbuff.**
read	*1s*	*1s*			*3m34s*	*8m29s*			*1h35m27s*	*3h53m57s*		
level												
1	14m32		0	170315			0	0			0	0
2	14m25		0	170315			0	0			0	0
3	12m13	54m8	0	170315			0	0			0	0
4	1m10	3m18	0	170315			0	0			0	0
5	24s	1m03	0	170315			0	0			0	0
6	20s	53s	0	170315			0	0			0	0
7	4s	12s	0	170315	2h24m		0	61068103			0	0
8	2s	8s	0	170315	47m52		0	61068103	14h7m18		0	1757890806
9	2s	6s	0	170315	12m11		0	61068103	4h14m07		0	1757890806
10	1s	4s	0	170315	3m36		247663	60820440	1h46m26		291548	1757599258
11	<1s	2s	0	170315	3m37		8591533	52476570	1h44m11		291548	1757599258
12	**<1s**	1s	0	170315	**3m37**		25101795	35966308	**1h36m42**		129667487	1628223319
13	<1s	1s	69190	101125	3m37		40185465	20882638	1h36m47		966674193	791216613
14	<1s	1s	88307	82008	3m37		49803339	11264764	1h36m47		1331590546	426300260
15	<1s	**1s**	137395	32920	3m36	**8m39**	55228057	5840046	1h36m39	**3h56m57**	1527122289	230768517
16	<1s	1s	155748	14567	3m36	8m37	58098312	2969791	1h36m33	3h55m05	1634977067	122913739
17	<1s	1s	162400	7915	3m36	8m40	59571359	1496744	1h36m30	3h54m56	1695689619	62201187

Table 4. Trixel Buffer query performance.

potentially 20 icosahedron triangles covering the Earth, the darker blue polygon/the project limit polygon spans 1595x962km, the centre image [A] and the image on the left [B] cover areas smaller than the displayed characters A and B.

In an earlier version of our system, before we introduced the "shortest triangle path information" or "point buffers at run-time", we casted a horizontal ray from the point query to a stable node void of geometry to infer the polygon status. This algorithm proved to be twice as fast as strategies that trace a horizontal ray out of the spatial tree using a tree of depth 16, and still significantly faster than a ray strategy that stops a ray at the polygon limits of the polygons in question. The automatic interior/exterior extraction of a tree of depth 17 used to take 49 minutes to compute with a 3GHz processor rather than 5 minutes (Table 2) because the adjacent triangles were grouped into several individual horizontal triangle strips of the same height, rather than just 1 test made and copied for the whole group. Point validity results were inspected visually and matched the visual results of the [Tay94a] algorithm that does not use spatial databases, this algorithm which uses a standard ray test took over 4 days to process the Kirkwall data set on the 3GHz machine. Our algorithm later became I/O bound using point buffers, with multithreading/MT (tree of level 12) and also with a single thread/ST (tree of level 15). Timings for just reading point queries from disk are given in the row labeled read in Table 4. For the multithreaded version we read blocks of 1000 points and inserted them into the back of a queue, blocks in the front of the queue were removed for processing while reading, if the queue reached its full capacity (5000 blocks, 120Mb) reading was stopped until all blocks in the queue were processed, this did not happen in the I/O bound cases (Fig.12).

5. DISCUSSION

Although the performance of our Point-in N polygon inclusion tests is I/O bound, it takes over a minute to render a binary visualization file of results (Figure 1, center), it would be nice to organize the file spatially during the queries so as to enable frustum culling&sampling according to zoom level for interactive rendering.

6. CONCLUSIONS

We developed Trixel Buffers and nine rules that extend the Gauss-Jordan theorem to be used efficiently with hierarchical spatial databases. The concept can be used for determining the inclusion in 3D/octrees, and extend other quadtree methods.

7. ACKNOWLEDGMENTS

The authors wish to thank the United Kingdom´s Hydrographic Office for providing the Bathymetric data. Funding by UKHO CONTRACT No HA294/024/009.

8. REFERENCES

[Berg97a] Berg, M., Kreveld, M., Overmars, M. and Schwarzkopf, O. Computational Geometry Algorithms and Applications. Springer-Verlag, New York, ISBN 3-540-61270-X, 1997.

[Fek90a] Fekete, G. Rendering and managing spherical data with Sphere Quadtrees. Proc. of IEEE Visualization, 14:176-186, 1990.

[Hai94a] Haines, E. Point in Polygon Strategies. Graphics Gems IV, pp.24-46, 1994.

[Edel86a] Edelsbrunner, H., Guibas, L. J., and Stolfi, J. Optimal point location in a monotone subdivision. SIAM J. Comput, 1986.

[Ili06a] Iliffe, J., Ziebart, M., Turner, J., Oliveira, J. F. and Adams, R. The VORF project – Joining up Land and Marine Data. GIS Professional, Issue 13, November/December, 2006, pp.24-26.

[Kir83a] Kirkpatrick, D. G. Optimal search in planar subdivisions. SIAM J. Comput, 1983.

[Oli06a] Oliveira, J. F., and Buxton, B. F. PNORMS: Platonic derived Normals for error bound compression. ACM VRST, 2006.

[Pat06a] Patrascu, M., Planar Point Location in Sublogarithmic Time. Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS´06), 325-332, 2006.

[Pov04a] Poveda, J., Gould M., and Oliveira, A. A New Quick Point Location Algorithm. Springer´s Lecture Notes in Comp.Sci.3289.

[Prep85a] Preparata, F. P., and Shamos, M. I. Computational Geometry an Introduction. Springer-Verlag,, ISBN 0-387-96131-3, 1985.

[Ran02a] Randall, D. A., Ringler, T. D., Heikes, R. P., Jones, P., and Baumgardner, J. Climate Modeling With Spherical Geodesic Grids, Computing in Science & Engineering, 2002.

[Sam90a] Samet, H. Applications of Spatial Data Structures. Springer-Verlag, Addison Wesley, Reading, MA, ISBN 0-201-50300-X, 1990.

[Sar86a] Sarnak, N., Tarjan, R. E. Planar Point Location Using Persistent Search Trees. Communications of the ACM, 29(7), 1986.

[Ham13a] Hamilton, C.J. Views of the Solar System: http://www.solarviews.com

[Tay94a] Taylor, G. E. Point in polygon test. Survey Review, 32(254): 479-484, 1994.

[Xu03a] Xu, Z. S., and Tang, L. An efficient rejection test for ray/triangle mesh intersection. Journal of Software, 14(10): 1787-1795, 2003.