Simple Traversal of a Subdivision Without Extra Storage*

Mark de Berg, Marc van Kreveld,
René van Oostrum and Mark Overmars

Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB, The Netherlands
Email: {markdb,marc,rene,markov}@cs.ruu.nl

Abstract

In this paper we show how to traverse a subdivision and to report all cells, edges and vertices, without making use of mark bits in the structure or a stack. We do this by performing a depth-first search on the subdivision, using local criteria for deciding what is the next cell to visit. Our method is extremely simple and provably correct. The algorithm has applications in the field of Geographic Information Systems (GIS), where traversing subdivisions is a common operation, but modifying the database is unwanted or impossible. We show how to adapt our algorithm to answer related queries, such as windowing queries and reporting connected subsets of cells that have a common attribute. Finally, we show how to extend our algorithm such that it can handle convex 3-dimensional subdivisions.

Keywords: Subdivisions, traversal algorithms, topological data structure, windowing, three dimensions.

1 Introduction

The basic spatial vector data structure in any geographic information system is the one that stores the zero-, one- and two-dimensional features of planar subdivisions. Contemporary GISs like ARC/INFO [15] and the DIME file [16] use polygon structures that store the explicit topology as well. This means that from any feature, it is possible to access adjacent features efficiently. Essentially, these structures are similar to the doubly connected edge list and quad edge structures used in computational geometry.

*This research is supported by a PIONIER project of the Dutch Organization for Scientific Research N.W.O.

A basic algorithmic routine that can be applied to any planar subdivision is its traversal. The objective of such a traversal can be to report all boundary segments on a map, or to find a specific one. Unfortunately, to traverse a topological polygon structure one needs to record what features have been visited, to avoid continuing forever. This means that the zero-, one- and two-dimensional features must have a mark bit to capture this information. This is highly undesirable, because mark bits require extra storage space, or even worse, the data structure at hand may not have such mark bits with the features. Another drawback of mark bits in concurrent situations is that subdivision traversal cannot be performed by two users at the same time.

An algorithm for the traversal of *triangulated subdivisions,* or *triangulated irregular networks,* that did not require mark bits to record what triangles had been visited and which had not was developed by Gold et al. [11] (see also Gold and Maydell [12] and Gold and Cormack[10]). Their method involved choosing one starting point, and defining for each triangle exactly one incident edge through which the triangle could be entered. With the correct definitions and choices one can make sure that every triangle in the subdivision is reported exactly once. In fact, an order is defined on the triangles and the triangles are reported according to this order. Other work on ordering of triangles was done by De Floriani et al. [8], who used the order for visibility purposes.

In the field of computational geometry, efficient traversal algorithms have also been studied. Avis and Fukuda [1] gave an algorithm to report the vertices of an *arrangement* or a *convex polytope* without using mark bits in the data structure that represents the arrangement or polytope. Their description is a rather abstract one and does not address non-convex subdivisions; see also Fukuda and Rosta [9]. A generic algorithm for traversing graph-like data structures without storing any information about the visited parts of the data structure was developed by Avis and Fukuda [2].

In this paper we extend the result of Gold et al. [11] and Edelsbrunner et al. [7] to traverse subdivisions without using mark bits. Our ideas are similar to those of Avis and Fukuda [2] and of Edelsbrunner et al. [7]. Unlike their methods, our algorithm applies to any subdivision of which the vertices and edges form a connected graph. Furthermore, we provide various practical extensions for windowing and reporting parts of subdivisions. We prove the correctness of our algorithms for both convex and non-convex subdivisions. The algorithms are extremely simple; we implemented the algorithm for straight-line subdivisions in about 100 lines of C-code and it works fine. We also extend the results to subdivisions with curved arcs, and we give extensions for the traversal of a connected part of a subdivision, or the part of a subdivision that lies inside a specified window. These operations are commonly used by geographic information systems. One usually doesn't need the whole subdivision to be traversed, but just some subregion in which the user is interested. We also address the traversal of the surface of a convex polyhedron in 3-dimensional space, and the traversal of a convex subdivision in 3-dimensional space. In all cases, no mark bits are required in the data structure. Some of our results have been obtained by Snoeyink [18] simultaneously.

We present our algorithms using the *doubly-connected edge list structure* [4, 14, 17], a

standard data structure used in computational geometry that stores topology explicitly. This is not a restriction; simple adaptations to the algorithms can be done so that they apply to the quad edge structure [13], the fully topological network structure [3], the ARC/INFO structure [15], the DIME file [16], or any other vector data structure that stores the topology explicitly.

In the next section we give a brief description of the doubly-connected edge list structure. Then we proceed with the simple traversal algorithm for subdivisions embedded in the Euclidean plane $\mathbb{E}^2$ and prove its correctness. In Section 2.4 we show how to adapt the algorithm such that it can handle TINs and surfaces of 3-dimensional polyhedra and subdivisions with curved edges. We address the traversal of connected subsets of the cells in a subdivision and windowing queries in Section 2.5. Finally, in Section 3 we extend our results to connected subdivisions in three dimensions.

2 Traversing a planar subdivision

2.1 Preliminaries

A planar subdivision S is a partition of a two-dimensional manifold M into three finite collections of disjoint parts, the vertices, the edges and the cells. A subdivision is *connected* if its edges and vertices form a connected set. This implies that all bounded cells are simple polygons without holes.

The data structure our algorithm operates on is the *Doubly Connected Edge List (DCEL)*. In the DCEL-structure, edges in the subdivision are represented by two directed half-edges, such that one of the half-edges bounds one cell incident to the edge and the other half-edge bounds the other cell. If $\vec{e}$ is a half-edge, then $twin(\vec{e})$ denotes the half-edge that is part of the same edge. Every half-edge is oriented in such a way that the cell to which it is incident lies to its left. In this way, for all bounded cells, all incident half-edges form a counterclockwise cycle around the cell. The half-edges that bound the one unbounded cell (the *outer cell*) form a clockwise cycle. This is illustrated in Figure 1.

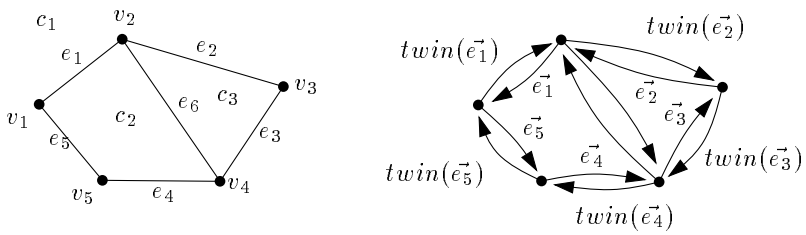

Figure 1: A subdivision and the corresponding orientation of half-edges.

The DCEL consists of a table of half-edge records, a table of vertex records, and a table of cell-records.

Any vertex record stores the coordinates of the vertex and a single reference to a half-edge record of a half-edge that has the vertex as its origin; the other half-edges incident to this vertex can be accessed by first accessing the referenced half-edge. The primitives *Coordinates*(v) and *Incident-edge*(v) are used for the values of these fields for a vertex v. Also, non-geometric information may be stored. For the subdivision in Figure 1 the vertex table will look something like this:

Vertex	*Coordinates*	*Incident-edge*
v_1	$(\cdots, \cdots)$	$twin(\vec{e_1})$
v_2	$(\cdots, \cdots)$	$\vec{e_1}$
$\cdots$	$(\cdots, \cdots)$	$\cdots$

If the edges and vertices of the subdivision form a connected subset of the plane, then every cell is incident to one cycle of half-edges. In this case, a cell record stores one reference to a half-edge record of a half-edge incident to that cell, and possibly some non geometric information. The primitive *Outer-incident-edge*(c) is used to access the half-edge for a bounded cell c, and the primitive *Inner-incident-edge*(c') is used for the unbounded cell c'. The table below shows the cell table for the subdivision in Figure 1.

Cell	*Outer-incident-edge*	*Inner-incident-edge*
c_1		$twin(\vec{e_4})$
c_2	$\vec{e_1}$	
c_3	$\vec{e_2}$	

For any half-edge $\vec{e}$, its record stores one reference to the record of the vertex at which it originates, a reference to the record of the incident cell c (which lies to its left), a reference to the half-edge that precedes $\vec{e}$ in the cycle of edges around c, a reference to the record of the half-edge that follows after $\vec{e}$ in this cycle, and a reference to the record of the complementary half-edge $twin(\vec{e})$:

Half-edge	*Origin*	*Incident-cell*	*Prev*	*Next*	*twin*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
e_5	v_1	c_2	$\vec{e_1}$	$\vec{e_4}$	$twin(\vec{e_5})$
$twin(\vec{e_5})$	v_5	c_1	$twin(\vec{e_4})$	$twin(\vec{e_1})$	$\vec{e_5}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

The DCEL-primitives can be combined to answer more complex queries. For example, suppose that we want to know what is the first outgoing half-edge of vertex v_4 that we encounter if we rotate counterclockwise around v_4 and start at half-edge $\vec{e_3}$ (Figure 1). This query is answered by $twin(prev(\vec{e_3}))$.

2.2 The local method

We assume in this section and in Section 2.3 that the subdivision our algorithm operates on is embedded in the Euclidean plane $\mathbb{E}^2$, that its edges are straight line segments, and that the subdivision is connected. In Section 2.4 we will show how to overcome the first two restrictions.

The idea behind our algorithm is to define an order on the cells of the subdivision and to visit the cells in this order. For every cell we define a unique predecessor, such that the predecessor relationship imposes a directed graph $G(V, E)$ on the subdivision, with $V = \{c \mid c \in S\}$ and $E = \{(c', c) \mid c' \textit{ is the predecessor of } c\}$. Our algorithm reports the cells of S in depth-first order, that is, in the order corresponding to a depth-first order in the graph G. To facilitate this, the predecessor relationship on the cells of S needs to be defined such that G is a tree, rooted at the node of the starting cell c_{start}, and all arcs are directed away from the root. Graph G never is explicitly determined or stored, but it is used to prove the correctness of the algorithm. The most important step of the proof is showing that G is a tree. Analogous to graph terminology, a path π in the subdivision from one cell c' to another cell c is a sequence of cells $(c' = c_0, c_1, \ldots, c_k = c)$ such that c_{j-1} is the predecessor of c_j for each $j, 1 \leq j \leq k$. To show that G is a directed rooted tree, we must show that for every cell c there is exactly one path π from c_{start} to c.

The predecessor relationship is defined by using "local information" only: we identify for each cell c a special half-edge, $entry(c)$. The cell c' incident to $twin(entry(c))$ is defined as the predecessor of c. We define the entries of the cells by first choosing an arbitrary point p in the starting cell c_{start} and then taking the following steps:

- For every cell c in S except for c_{start}, we calculate the Euclidean distance between p and the closures of all the half-edges of c. We define the half-edge $\vec{e}$ of c that has minimum distance to be the entry of c. In some cases ties have to be broken. Consider the circle C centered at p that intersects the boundary of the cell c and with minimum radius. If C intersects the boundary of c in more than one point, we choose the first of these points on C, clockwise around p, starting in some fixed direction θ (Figure 2). Let this point be p'.

  - If p' lies in the interior of a half-edge $\vec{e}$ incident to c, then $\vec{e}$ is the entry of c.
  - If p' lies on a vertex v of the boundary of c, then we must choose between the two half-edges $\vec{e}$ and $\vec{e'}$ incident to c and have v as destination and source, respectively. If $\vec{e}$ is exposed to p (i.e., the directed line $\vec{\ell}$ induced by $\vec{e}$ has p strictly to the right), we choose $\vec{e}$ as the entry of c; otherwise, we choose $\vec{e'}$. It is straightforward to verify that at least one of $\vec{e}$ and $\vec{e'}$ is exposed to p.

Testing whether a half-edge $\vec{e}$ is the entry of its incident cell takes $O(1)$ time for convex subdivisions; we simply compare the distance between p and the closure of $\vec{e}$ with the distance between p and the closure of the successor of $\vec{e}$ and distance between p and the closure of the predecessor of $\vec{e}$, respectively. If the distance between p and the closure of $\vec{e}$

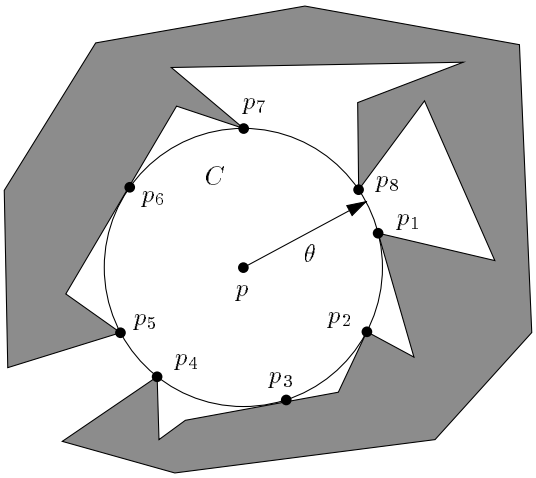

Figure 2: p_1 is the first point on C, starting in direction θ.

is smaller than the other two distances, then $\vec{e}$ is the entry of its incident cell. It can also be the entry if the source or destination of $\vec{e}$ is closest to p, in which case a few extra tests need be done. The tests can be performed in $O(1)$ time.

For non-convex subdivisions, determining whether a half-edge $\vec{e}$ is the entry of its incident cell c involves the comparison of the distance between p and the closure of $\vec{e}$ with the distance between p and the closures of all other half-edges incident to c; this takes time linear in the number of half-edges incident to c.

Lemma 1 *With the predecessor relationship defined as above, for every cell c in S there is exactly one path π from c_{start} to c.*

Proof: It is easy to verify that there cannot be more than one path from c_{start} to a cell c, since every cell has exactly one entry. Now suppose that there exists a non-empty set $S' = \{c \mid c \in S \text{ and no path } \pi \text{ exists from } c_{start} \text{ to } c\}$. Define R' to be the closure of the cells in S'. Let C be a circle centered at the point p as defined above, and let C have radius such that it intersects the boundary of R', but not its interior. C intersects the boundary of at least one cell c in S'.

Let $p' \in R'$ be the first point on C, clockwise around p, starting in direction θ. Choose one of the cells in S' that has p' on its boundary (observe that there is at least one such cell), and let this cell be c. We will show that there is a path from c_{start} to this cell c, thus deriving a contradiction, which will prove the lemma.

For the cell c, $entry(c)$ is defined as above. Observe that p' lies in the closure of $entry(c)$. If p' lies in the interior of $entry(c)$, then the predecessor c' of c, i.e. the cell incident to $twin(entry(c))$, intersects C, and a path π' from c_{start} to c' exists. Since c' is the predecessor of c, there is a path π from c_{start} to c via c' (Figure 3).

On the other hand, suppose that p' is one of the endpoints of $entry(c)$. Assume p' is the destination vertex of $entry(c)$; the case where p' is the source vertex of $entry(c)$ is

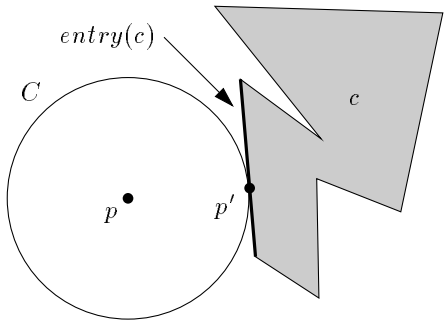

Figure 3: The cell incident to $twin(entry(c))$ intersects C.

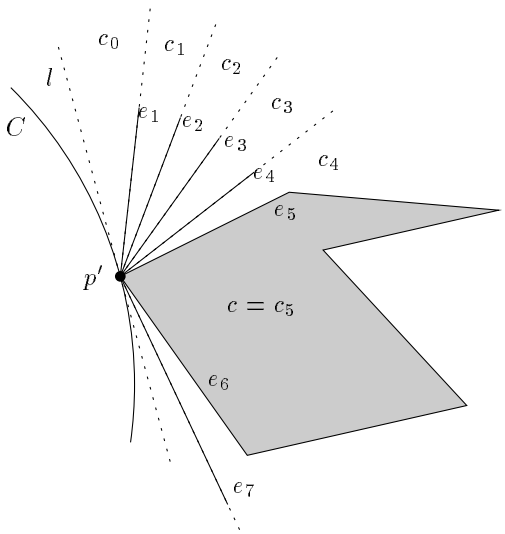

Figure 4: Half-edges having p' as their destination.

analogous. Let $e\vec{l}l$ be the directed line tangent to C at p' and with p to its right. Consider all half-edges that have p' as their destination and lie to the left of $\vec{\ell}$ (or in the line), as illustrated in Figure 4. Let these half-edges be $\vec{e_1}, \ldots, \vec{e_k} = entry(c), \vec{e_{k+1}}, \ldots$, in cyclic order as in Figure 4. The cell incident to $\vec{e_i}$ is labeled c_i for $1 \leq i \leq k$, and the cell incident to $twin(\vec{e_1})$ is labeled c_0. Observe that c_{i-1} is incident to $twin(\vec{e_i})$ for $1 \leq i \leq k$. Also note that a single cell can have more than one label, as illustrated in Figure 5. Using induction on i, we will show that there is a path π_i from c_{start} to c_i. Since $c_k = c \in S'$, this again leads to a contradiction.

- Since no edges lie between $\vec{e_1}$ and the tangent to C, c_0 must intersect C. It follows that there is a path π_0 from c_{start} to c_0.
- Assume there is a path π_i from c_{start} to all cells $c_0, c_1, \ldots, c_i$ for some $0 \leq i < k$. We can distinguish three cases, which together cover all possibilities:

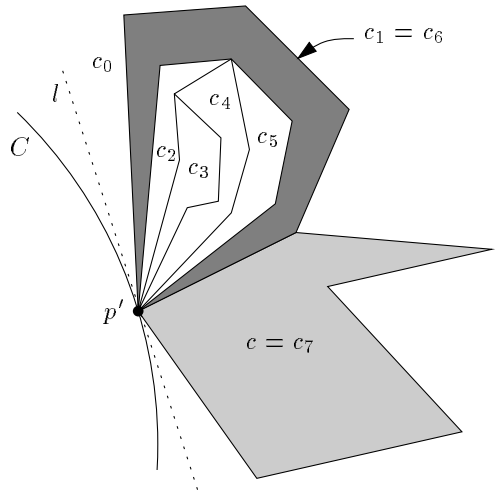

Figure 5: Cells can have multiple labels.

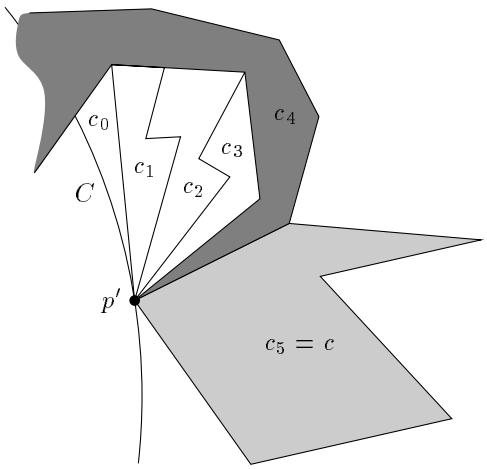

Figure 6: Cells intersecting the circle C are reachable from c_{start}.

- If $c_{i+1} \notin S'$ then by definition of S' there is a path from c_{start} to c_{i+1}. Otherwise, we can conclude by definition of C that the cell c_{i+1} doesn't intersect the interior of C, as illustrated in Figure 6. Nor can its boundary have a point on C before p', starting clockwise from θ.
- c_{i+1} has another label c_j, with $j < i$, as illustrated in Figure 5. In this case, there is a path $\pi_{i+1} = \pi_j$ from c_{start} to $c_j = c_{i+1}$ by the induction hypothesis.
- $c_{i+1} \in S'$ and it has no other label c_j with $j < i$, In this case it's straightforward to verify that $entry(c_{i+1}) = \vec{e}_i$: C is the smallest circle centered at p intersecting the boundary of c_{i+1}, and p' is the first point on C and the boundary of c_{i+1} that is encountered when we rotate clockwise around p, starting in direction θ.

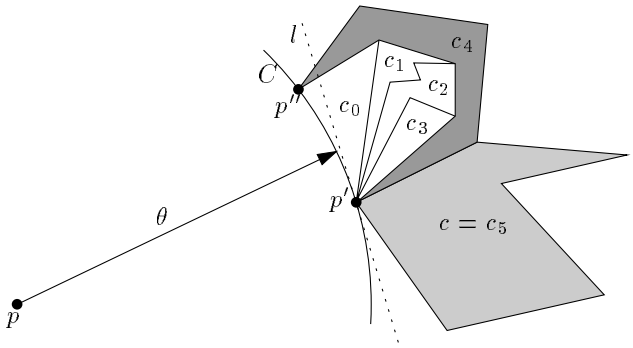

Figure 7: By choice of p', $entry(c_4)$ is incident to p'.

Also note that $\vec{e}_i$ is exposed because $\vec{e_k}$ is.

Since there is a path π_i from c_{start} to cell c_i by the induction hypothesis and c_i is the predecessor of c_{i+1}, there is a path π_{i+1} from c_{start} to c_{i+1} via c_i.

Since in all cases there is a path to c_{i+1}, we conclude by induction that there is a path to $c_k = c$. □

2.3 The algorithm

Using the methods of the previous section for finding the entry of a cell c, we can develop a very simple algorithm that traverses a subdivision S in a depth-first manner without using a stack and reports all cells of S:

Algorithm *Traverse*(S, c)
Input: A planar subdivision S of which the edges and vertices form a connected set, and a bounded starting cell $c \in S$

1. $\vec{e} \leftarrow$ *Outer-incident-edge*$[c]$
2. Report *Incident-cell*$[\vec{e}]$
3. **repeat if** $\vec{e} =$ *entry*(*Incident-cell*$[\vec{e}]$)
4. **then** $\vec{e} \leftarrow next[twin[\vec{e}]]$ (∗ return from cell ∗)
5. **else if** $twin[\vec{e}] =$ *entry*(*Incident-cell*$[twin[\vec{e}]]$)
6. **then** Report *Incident-cell*$[twin[\vec{e}]]$
7. $\vec{e} \leftarrow next[twin[\vec{e}]]$ (∗ explore new cell ∗)
8. **else** $\vec{e} \leftarrow next[\vec{e}]$ (∗ next half-edge along current cell ∗)
9. **until** $\vec{e} =$ *Outer-incident-edge*$[c]$

Of every half-edge of the counterclockwise cycle of edges around a cell c in S, algorithm *Traverse* inspects the corresponding *twin*-edge of the neighboring cell. If this half-edge is the entry of the neighboring cell, the algorithm continues in a depth-first manner in this cell after reporting it. Note that no stack or other memory resources are needed.

Since we start with the successor of the entry-edge of a cell the algorithm is finished with the counterclockwise cycle of edges of that cell when it encounters the entry of the cell (line 3).

Let n be the number of edges in the subdivision. Since the subdivision is planar, both the number of vertices and cells are also $O(n)$. In Algorithm *Traverse* the function *entry* is called at most $4n$ times, namely at most twice for each half-edge. For convex subdivisions we can determine in $O(1)$ time whether a half-edge is the entry of its incident cell; it follows that the running time of Algorithm *Traverse* is $O(n)$ for convex subdivisions. For non-convex subdivisions determining whether a half-edge is the entry of its incident cell takes time linear in the number of edges of that cell. If the cells have constant complexity, then the running time of our algorithm is $O(n)$; if the cells are non-convex polygons of which the complexity is not bounded by a constant, then the running time is bounded by the sum of the squares of the complexities of all cells, which is $O(n^2)$ in the worst case.

Algorithm *Traverse* can easily be adapted to report the edges and the vertices of the subdivision as well: when a cell is reported in line 6 of Algorithm *Traverse*, we list all its half-edges. To prevent a half-edge pair from being reported twice, we only report the one that has its direction in the range $[0, \pi)$. Of every half-edge $\vec{e}$ that is reported, we report its source vertex v if and only if $\vec{e}$ is the first half-edge with v as the source, in a cyclic order starting in some fixed direction θ. These tests can be performed in constant time using standard DCEL-operations. The (asymptotic) running time of Algorithm *Traverse* is not affected by these adaptations.

2.4 Overcoming the restrictions

In Sections 2.2 and 2.3 we made the assumptions that the subdivision our algorithm operates on is embedded in $\mathbb{E}^2$ and that its edges are straight line segments. In this section we will show how to adapt the the algorithm such that it can handle polyhedral terrains represented by TINs, surfaces of 3-dimensional convex polyhedra, and subdivisions with curved arcs.

Adapting our algorithm so that it can handle a polyhedral terrain is quite simple, if it is represented by a *Triangulated Irregular Network* (TIN). A terrain is a two-dimensional surface in three-dimensional space, with the special property that every vertical line intersects it in a point, or not at all. This means that a point p with coordinates (x, y, z) on the terrain can be mapped to a point $p' \in \mathbb{E}^2$ with coordinates (x, y), and that the mapping of all points of the terrain to $\mathbb{E}^2$ is injective. Calculating the *entry* of a triangle in a polyhedral terrain can be done with the method of Section 2.2 if we project every edge and vertex of the triangles on $\mathbb{E}^2$ when it is examined; algorithm *Traverse* needs no further adaptations. Since every projection takes $O(1)$ time, the asymptotic running time of the algorithm is not affected by the projections. For TINs our algorithm is similar to the algorithm of Gold and Cormack [10].

Surfaces of 3-dimensional convex polyhedra can be dealt with in the same way, although mapping the vertices and edges to $\mathbb{E}^2$ requires a little more effort for a convex polyhedron than for a polyhedral terrain (see Figure 8). We add an extra vertex v to the polyhedron, such that the new polyhedron is still convex, and all vertices and edges of the original polyhedron are vertices and edges of the new polyhedron. Seen from the new vertex v, only one face f of the original polyhedron is visible (the gray face in Figure 8). A point p on the original polyhedron is projected on this face f by taking the intersection of f and the line through v and p as the projection p' of p.

Finding a suitable vertex v involves taking a face f of the polyhedron, choosing a point in the interior of f, and translating it along the normal of f to the outside of the polyhedron. The faces incident to the *twin*-edges of the half-edges bounding f determine how little we should translate the point to the outside of the polyhedron. Testing this takes time linear in the complexity of f. After that, projections take $O(1)$ time for each vertex and edge of the polyhedron, and we obtain a planar convex subdivision of f.

We now can run Algorithm *Traverse* on the polyhedron; determining whether an edge is the entry of a face or not is done by performing the calculations described in Section 2.2 on the projected version of the edge. Again, the asymptotic running time of the algorithm is not affected by the projections.

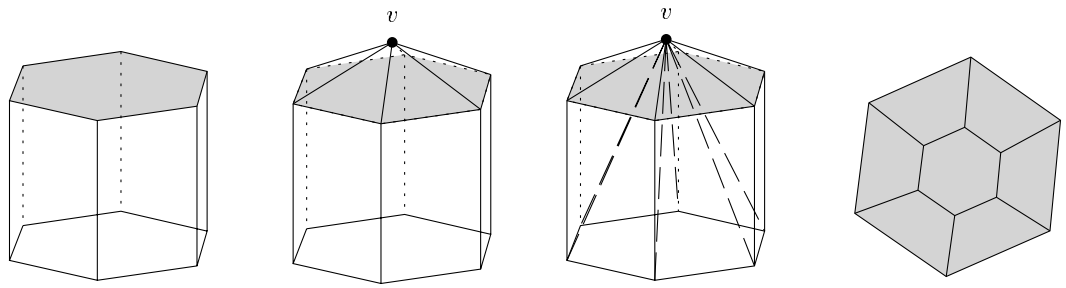

Figure 8: Projecting the edges and vertices of a convex polyhedron to one of its faces by adding a vertex.

Algorithm *Traverse* can also be used to report the cells, arcs and vertices of subdivisions with curved arcs instead of straight edges, provided that the arcs have constant description size and that we can calculate the minimum distance from a point to an arc. Furthermore, when we are to determine the *entry* of a cell, we need to adapt the notion of *exposed*. Intuitively, a curved half-arc $\vec{a}$ is exposed to a point p with respect to vertex v if close to v, the arc a has the cell to the one side and faces the point p to the other side. More formally, for all sufficiently small positive real values ϵ there is a point $q \neq v$ on $\vec{a}$ in an ϵ-neighborhood of v such that the interior of the segment $\overline{pq}$ does not intersect the cell c incident to $\vec{a}$ in that ϵ-neighborhood of v (see Figure 9).

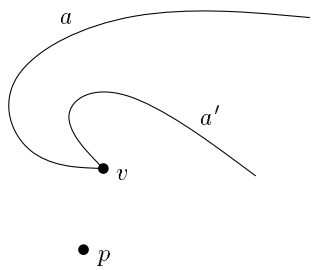

Figure 9: Arc a is exposed to p, whereas a' is not.

2.5 Related queries

Sometimes we don't want the whole subdivision S to be reported, but just some connected subset $S' \subseteq S$, such that all cells in S' have the same attribute as the starting cell c_{start} (Figure 10). For example, suppose that the starting cell lies in a forest; we then may ask "report all cells that lie in the same forest". We will show how to adapt Algorithm *Traverse* such that these queries can answered efficiently as well.

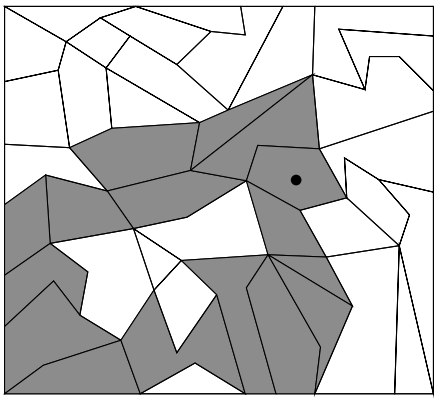

Figure 10: A connected set of cells with the same attribute as c_{start} (the cell containing the dot).

As Figure 10 indicates, the connected subset S' of S can contain holes consisting of cells that do not have the same attribute as the starting cell c_{start}. Some of the cells in these holes can be predecessors of cells in S'. However, we don't want to visit cells or half-edges in the holes. Instead, we rather consider each hole as a single cell. To do this, we need a way to traverse the counterclockwise cycle of half-edges that bound each hole, without ever visiting half-edges $\vec{e}$ for which $\vec{e}$ nor $twin(\vec{e})$ bounds a cell of S'. If we can achieve this, we can express the running time of the traversal algorithm in the combinatorial complexity of S', and not in the complexity of the whole subdivision S. To treat a hole as one single cell we do the following (Figure 11): suppose that half-edge $\vec{e}$ with destination vertex v is a half-edge of the boundary of a hole. We find its successor

in the counterclockwise cycle of half-edges that bound this hole by inspecting all outgoing half-edges of v in counterclockwise order, starting with $twin(\vec{e})$, until we find an edge $\vec{e'}$ that is incident to the hole. This half-edge $\vec{e'}$ is the successor of $\vec{e}$. In this way, we can treat the holes as ordinary cells of the subdivision, except that they are not reported. Notice that the cells of $S \setminus S'$, not enclosed by S', are handled correctly too. The algorithm won't even notice the difference between a hole and these outer cells.

What is the effect on the running time of Algorithm *Traverse*? Let k be the number of edges in S'. Then the total number of edges on the boundary of all holes is at most $O(k)$. Each edge of each hole is tested once for being the entry of the hole. Testing an edge of a hole involves traversing all edges that bound that hole. If we do this as described above, this takes time linear in the number of edges in S' of which the source vertex lies on the boundary of the hole; this number is $O(k)$ for all holes together. Testing each hole once takes $O(k)$ time; since each hole is tested at most $O(k)$ times, all these tests together take at most $O(k^2)$ time. If we combine this with the analysis in Section 2.3 we derive a running time of $O(k^2)$: the running time depends only on the complexity of the reported cells and not on the complexity of the whole subdivision.

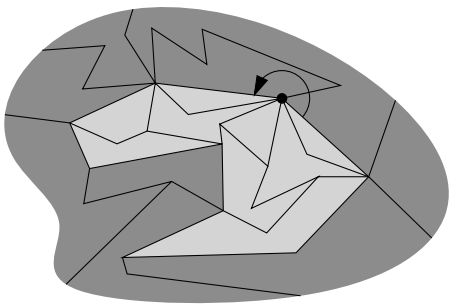

Figure 11: Finding the next edge of the (lightly shaded) hole.

Another query that arises often in practice is "given a subdivision S and a rectangular window W, report all cells in S that intersect W (Figure 12).

We solve this query as follows. Normally when we test a half-edge e for being the entry, we traverse the cycle of half-edges around c, keeping track of the edge that has minimum distance to a predefined point p in the starting cell c (breaking ties as described in Section 2.2). We make a little adaptation here: before calculating the distance of an edge to the point p, we clip the edge to the window W and perform our calculations on the clipped version of the edge (Figure 13). Edges that don't intersect W disappear; we consider their distance to p to be infinite. We also make a small adaptation in Algorithm *Traverse*: in line 5 we only test $twin[\vec{e}]$ on being the entry of its incident cell if $twin[\vec{e}]$ intersects the window W: if it doesn't it can't be the entry of its incident cell anyway, and omitting the test prohibits cells that don't intersect W to influence the running time of the algorithm.

It is straightforward to verify that with these adaptations our algorithm is still correct. Clipping the edges has the effect that cells that intersect the window W possibly fall apart

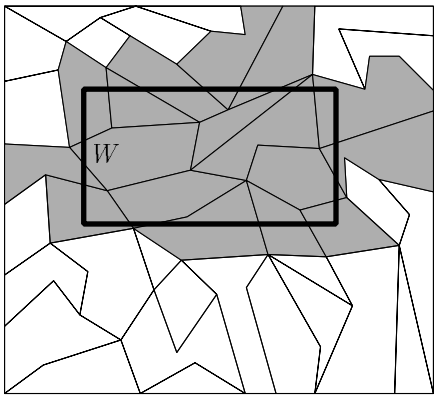

Figure 12: Reporting all cells that intersect a rectangular window.

into two or more pieces. Each of this pieces has a well-defined entry, and the piece of which the entry has the minimum distance to p determines the entry of the original (unclipped) cell.

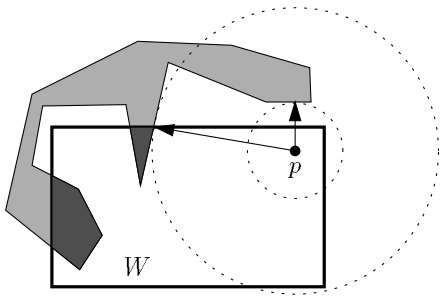

Figure 13: Clipping the edges of a cell to the window.

Clipping to the window W can be done in $O(1)$ time for each half-edge. Convex cells can have at most one piece inside a rectangular window; testing a half-edge of a convex cell on being the entry involves only clipping and comparing its distance to p with the distances of its predecessor and successor to p. Combining this with the analysis in Section 2.3, we derive a running time of $O(k)$ if the cells are convex polygons or non-convex polygons of constant complexity, where k is the total complexity of the cells that intersect the window W. If the cells are non-convex polygons of which the complexity is not bounded by a constant, then the running time is bounded by $O(k^2)$.

3 Extension to three dimensions

Extending our algorithm such that it traverses convex subdivisions in three dimensions and reports all (three-dimensional) cells, (two-dimensional) faces, edges and vertices is

straightforward. Dobkin and Laszlo [5] have developed data structures and operations for handling three-dimensional subdivisions; these are comparable with the DCEL data structure and operations which we used in the two-dimensional setting.

Again, we need to determine for each cell in the subdivision which one of its neighbors is its predecessor in the traversal. The entry of a cell is that face of the cell that is incident to the predecessor of the cell. Once we are able to determine for each cell which of its faces is its entry, we can apply the simple scheme of Algorithm *Traverse* again: we enter a cell c through its entry, and traverse all its faces (which we can do with the method described in Section 2.4, since the cells are convex). During this traversal of the faces, we test every face on being the entry of the other incident cell. If this is the case we continue in a depth-first manner on this cell, again without using a stack, as described in Section 2.3. After returning from this cell, we proceed with the next face of the cell c, until we are back at the entry of c again; then we return to the predecessor of c.

Determining the entry of a cell c of a three-dimensional subdivision is analogous to the two-dimensional case described in Section 2.2: we choose an arbitrary point p in the starting cell c_{start}, and define some face f that has minimum distance to p to be the entry of c, for any cell c except for the starting cell. Let p' be the unique point in the boundary of c that realizes the minimum distance.

If p' lies in the interior of a face f of c, then f is the entry of c. If p' lies on an edge or a vertex of c, we have to make a choice between the faces of c that are exposed to p. We choose the one that has the smallest angle with the plane that is tangent to the sphere C in the point p'. Ties can be broken in various ways, as long as it is done consistently.

4 Conclusions and open problems

We have developed a simple, flexible and efficient algorithm for traversing various kinds of planar subdivisions without using mark bits in the structure or a stack. Our algorithm reports each cell, edge and vertex exactly once. The algorithm can handle subdivisions embedded in the Euclidean plane $\mathbb{E}^2$, as well as polyhedral terrains, and surfaces of convex 3-dimensional polyhedra. It can easily be adapted to report a connected subset of the cells in the subdivision, or to answer windowing queries; both adaptations result in an output sensitive algorithm. Extending the algorithm to handle convex subdivisions in three dimensions is straightforward. An implementation of the algorithm for planar subdivisions with straight edges took about 100 lines of C-code.

A number of problems remains to be solved. We have looked at non-connected subdivisions, that is, subdivisions of which the edge and vertex set is unconnected (Figure 14), but at this moment it is unclear if these can be handled without making use of a stack or other memory resources to keep track of the components in the subdivision that have been visited.

Also unsolved is the problem of traversing the surface of non-convex polyhedra; although these are topologically equivalent to a sphere, which means that they can be projected to

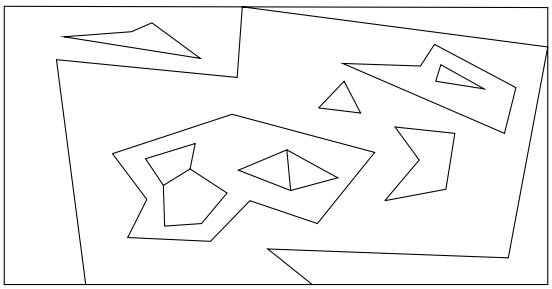

Figure 14: An unconnected subdivision.

$\mathbb{E}^2$, there is no way to determine the projection of a vertex of the surface of a non-convex polyhedron if we are only allowed to use local information: we need to know the geometry of the whole polyhedron. In other words: to traverse the surface of a non-convex polyhedron without using mark bits or a stack, we have to traverse its surface using mark bits or a stack, which may be regarded as cheating.

Since traversing non-convex 3-dimensional subdivisions involves traversing the surfaces of its cells which are non-convex polyhedra, there is no use in attacking this problem before the former problem has been solved.

In many practical situations such as those arising in GISs, the cells in a subdivision represent geographical entities like countries. It may well be that two adjacent cells have long chains of edges on their common boundaries. If one of the cells is the predecessor of the other, then only one edge in the chain is the entry. It would be interesting to find an elegant way to represent chains of edges by "pseudo-edges" between vertices of degree three and higher, in order to avoid the traversal of all the edges in the chain. This, of course, would involve modifying the data structures and adapting the definition of the *entry* of a cell, and we have not yet succeeded in doing this correctly.

References

[1] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 98–104, 1991.

[2] D. Avis and K. Fukuda. Reverse search for enumeration. Technical Report SOCS-92.21, School of Computer Science, McGill University, 1992.

[3] P.A. Burrough. *Principles of Geographical Information Systems for Land Resources Assessment.* Number 12 in Monographs on Soil and Resources Survey. Clarendon Press, Oxford, 1986.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry by Example*. 1996. manuscript.

[5] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 86–99, 1987.

[6] H. Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 145–151, 1989.

[7] H. Edelsbrunner, L.J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Computing*, 15:317–340, 1986.

[8] L. De Floriani, B. Faldicieno, G. Nagy, and C. Pienovi. On sorting triangles in a delaunay tessalation. *Algorithmica*, 6:522–532, 1991.

[9] Komei Fukuda and Vera Rosta. Combinatorial face enumeration in convex polytopes. *Computational Geometry, Theory and Applications*, 6:191–198, 1994.

[10] C. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. In *Proc. 2nd Int. Sympos. Spatial Data Handling*, pages 74–85, 1986.

[11] C. M. Gold, T. D. Charters, and J. Ramsden. Automated contour mapping using triangular element data structures and an interpolant over each irregular triangular domain. *Computer Graphics*, 11(2):170–175, 1977.

[12] C.M. Gold and U. Maydell. Triangulation and spatial ordering in computer cartography. In *Proc. Canad. Cartographic Association Annual Meeting*, pages 69–81, 1978.

[13] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[14] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[15] D.J. Peuquet and D.F. Marble (Eds.). ARC/INFO: an example of a contemporary geographic information system. In *Introductory Readings in Geographic Information Systems*, pages 90–99. Taylor & Francis, 1990.

[16] D.J. Peuquet and D.F. Marble (Eds.). Technical description of the DIME system. In *Introductory Readings in Geographic Information Systems*, pages 100–111. Taylor & Francis, 1990.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, New York, NY, 1985.

[18] J. Snoeyink, 1995. Personal Communication.