GENERATION OF POLYGONAL MESHES IN COMPACT SPACE

Sergio Salinas-Fernández[1] José Fuentes-Sepúlveda[2] Nancy Hitschfeld-Kahler[3]

[1] *University of Chile, Santiago, CL-RM, Chile. ssalinas@dcc.uchile.cl*
[2] *University of Concepción. Concepción, CL-BI, Chile. jfuentess@inf.udec.cl*
[3] *University of Chile, Santiago, CL-RM, Chile. nancy@dcc.uchile.cl*

ABSTRACT

We present a new compact half-edge data structure for storing polygonal meshes. This data structure allows us to reduce the memory usage for the topological information of the mesh in a 99% with respect to a non compact half-edge. The compact data structure works for any kind of planar graph. To test this compact data structure, we have implemented a new version of the polygonal mesh generator Polylla using the compact half-edge data structure. We tested the mesh generator using two implementations of the half-edge data structure: the first one (non-compact) using an array of structures and the second one (compact) using PEMB, a modification to the Turán's graph representation such that it supports fast navigation. Finally, we show preliminary experiments to compare the performance of compact Polylla versus the non-compact version.

Keywords: Polygonal mesh, compact data structures, arbitrary shape polygon

1. INTRODUCTION

Polygonal mesh generation is a research area broadly studied, with applications in many fields such as computer graphics [1], geographic information systems [2], Finite Element Methods (FEM) [3], among others. In the particular case of FEM, the polygons composing a mesh has to fulfill some quality shape criteria. Typical meshes tend to contain only triangles or quadrilaterals, except for Voronoi meshes that contain convex polygons as basic cells [4]. In recent years, the Virtual Element Method (VEM) [5] has shown that mesh generation can be based not only on convex but also non-convex polygons [6, 7], opening a new research line to generate quality meshes for VEM [8, 9].

There are several approaches to generate spatial discretizations, usually composed of triangles, quadrilateral, or both cell types [10, 11]. In general, mesh algorithms can be classified into two groups [12, 13]: *(i)* direct algorithms: meshes are generated from the input geometry, and *(ii)* indirect algorithms: meshes are generated starting from an input mesh, typically an initial triangle mesh. By joining triangles, several algorithms have been developed to generate quad meshes [14, 15, 16]. Such kind of mesh generators are also known as tri-to-polygon mesh generators. An advantage of using indirect methods is that the automatic generation of triangular meshes is a well-studied problem and several efficient and robust tools are available for free to generate triangulations [17, 18, 19].

Huge simulations such as hydrological modeling on the earth's surface, earthquakes, and climate modeling, among other applications, require solving numerical methods using meshes with millions of points and faces. A way to handle those kinds of applications is by using GPU parallel programming, but GPU solutions are more limited in memory than CPU solutions. An approach to face this kind of memory problem is using *compact data structures*. Compact data structures store information using a compact representation of it while supporting operations directly over such a representation. Examples of compact data structures in-

Figure 1: Polylla mesh of the football team Club Universidad de Chile's logo. The mesh contains 410 polygons and 1039 edges. White spaces represent holes.

clude integer vectors, trees, graphs, text indexes, etc. (see [20] for a thorough list). Of particular interest for this work are the results of *Ferres et al.* [21] to represent planar graph embeddings in compact space. From now on, we will refer to their compact data structure as PEMB. Interestingly, PEMB can be seen as a compact version of the half-edge data structure [22], using around 5 bits per edge. Given a planar graph τ, and an arbitrary edge e and face f of τ, PEMB accomplishes:

- Edge e has an orientation
- Edge e has a twin edge with opposite orientation
- Edge e is accessible by random access
- All edges of face f have the same orientation

In this work we show how to use PEMB as a compact half-edge data structure to implement *compact Polylla*, a compact version of the polygonal mesh generator *Polylla* [23], a mesh generator based on terminal-edge regions. The original version of Polylla does not use the half-edge data structure, so as a byproduct we will provide a new non-compact version based on the half-edge data structure. An example of a Polylla mesh is shown in Figure 1. Both PEMB and Polylla are aimed to work on planar 2D polygonal meshes. Thus, all the results of this work are limited to 2d geometries defined by PSLGs.

The paper is organized as follows: Section 2 explains the concepts necessary to understand this paper. Section 3 explains the implementation of the compact and the non-compact half-edge data structure. Section 4 shows a half-edge version of Polylla. Section 5 shows the experiments of time and memory, and Section 6 shows the conclusions and future work for Polylla and PEMB.

2. BACKGROUND

This section explains the half-edge data structure, how the PEMB data structure works, and the Polylla algorithm.

2.1 Compact representation of a planar graph

A way to address the memory problem of representing large data sets is using lossless compression, that is, reducing the number of bits as much as possible without losing information. However, in general, a disadvantage of this alternative is that it only allows fast navigation with decompressing the data.

An alternative is to represent the data using a compact representation and build data structures on top of it to support fast operations. For instance, Tutte [24] showed that $3.58m$ bits suffice to represent a planar graph embedding with m edges in the worst case. In the case of triangular meshes, works exist that try to reach such a bound. For example, the compact data structure showed in [25] reduces the representation of a graph by assigning a unique id to each half-edge and stores only the correspondence between adjacent half-edges and a mapping from each vertex to any of its incident half-edges. Catalog representation [26] gathers the triangles of a triangulation into patches to reduce the number of references to the elements in the triangulation. The Star-Vertex data structure [27] stores the geometrical position of the vertices and a list of their neighbor's vertices to represent a planar mesh. More compact data structures to represent planar graphs can be seen in [28, 29]. The space consumption of the previous works is $O(m)$ references, equivalent to $O(m \log m)$ bits. For a more detailed analysis, see [28, Section I].

The space consumption can be improved to $O(m)$ bits using succinct data structures. In this paper, we use the work of Ferres *et al.* [30], a succinct data structure to represent planar embeddings (see Section 2.3). A succinct data structure [20, Foreword] is a more restricted version of a compact data structure, where a combinatorial object, such as a graph or a tree, is represented using space closed to its information theory lower bound and add only a lower-order term to support fast queries. Aleardi and Devillers showed a

similar result [31] that uses the succinct representation of the Schnyder wood triangulation to get similar queries of the winged-edge data structure [32]. However, their solution is limited to triangulations, while Ferres *et al.* works for any planar graph embedding.

2.2 Half-edge data structure

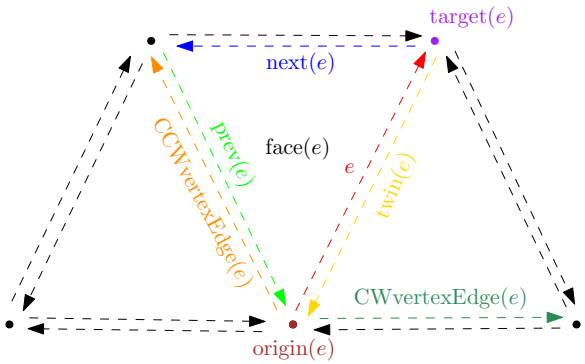

Figure 2: Visual representation of the queries that can be applied to the half-edge e.

The half-edge data structure, also known as doubly connected edge list (DCEL) [22], is an edge-based data structure where each edge of a polygonal mesh is represented as two half-edges of opposite orientation. Each half-edge contains information about its orientation and adjacent elements, allowing easy navigation inside the mesh. Given a half-edge e, the primitive queries [33, Chapter 2] supported by the data structure are the following (see Figure 2):

- TWIN(e): return the opposite half-edge of e, sharing the same endpoints.
- NEXT(e): return the half-edge next to e inside the same face in counter-clockwise order.
- PREV(e): return the half-edge previous to e inside the same face in counter-clockwise order.
- ORIGIN(e): return the source vertex of the half-edge e.
- TARGET(e): return the target vertex of the half-edge e.
- FACE(e): return the index of the incident face to the half-edge e.

Based on the primitive queries, more complex queries can be defined. Given a half-edge e, vertex v, and face f of a mesh, we define the following complex queries that will be used later:

- CCWVERTEXEDGE(e): return the half-edge with source vertex $origin(e)$ and next to e in counter-clockwise.
- CWVERTEXEDGE(e): return the half-edge with source vertex $origin(e)$ and previous to e in clockwise.
- EDGEOFVERTEX(v): return an arbitrary half-edge with source vertex v.
- INCIDENTHALFEDGE(f): return an arbitrary half-edge delimiting face f.
- ISBORDER(e): return true if half-edge e is incident to the outer face of the mesh.
- LENGTH(e): return the length of the half-edge e.
- DEGREE(v): return the number of the edges incident to with source vertex v.

2.3 PEMB data structure

PEMB [30] is a compact data structure designed to represent planar graph embeddings. Given a planar graph embedding $\tau = (V, E)$, PEMB represents τ in $4|E| + o(|E|)$ bits and support navigational operations in near-optimal time [34]. To construct PEMB, τ is decomposed into two spanning trees: an arbitrary spanning tree T for τ and the complementary spanning tree T' of the dual graph of τ. Thus, navigational operations over τ are mapped to navigational operations over the spanning trees. T is traversed in counter-clockwise order in a depth-first manner, starting at an edge incident to the outer face, generating a balanced parenthesis representation of T, where open/close parentheses are represented with 0/1 bits, respectively. During the traversal of T, a clockwise depth-first traversal of T' is induced, generating a balanced parenthesis representation of T' and a bitvector representing how both spanning trees are intertwined. The balanced parenthesis representations of both trees are stored as compact trees [20, Chapter 8], while the bitvector is stored as a compact bitvector [20, Chapter 4]. After the construction of PEMB, the vertices are referred to by their rank in the depth-first traversal of T. Thus, the first visited vertex in the traversal has id 0 and the last id $|V| - 1$. A planar graph $\tau = (V, E)$ is represented as three bitvectors:

- a bitvector $A[1..2|E|]$ in which $A[i] = 1$ if and only if the i-th edge we process in the traversal of τ is in T, and $A[i] = 0$ otherwise.
- a bitvector $B[1..2(|V| - 1)]$ in which $B[i] = 0$ if and only if the i-th time we process an edge in T during the traversal, is the first time we process that edge, and $B[i] = 1$ otherwise.
- a bitvector $B^*[1..2(|E|-|V|+1)]$ in which $B^*[i] = 0$ if and only if the i-th time we process an edge in T' during the traversal, is the first time we process that edge, and $B^*[i] = 1$ otherwise.

Figure 3 shows an example of the decomposition of a triangulation into two intertwined spanning trees. Its representation is stored as:

$A[0..48] = 01100011000110000011010101000000110101001111111 0$
$B[0..22] = 0010101000010001111111$
$B^*[0..26] = 00001000000011111100011111$

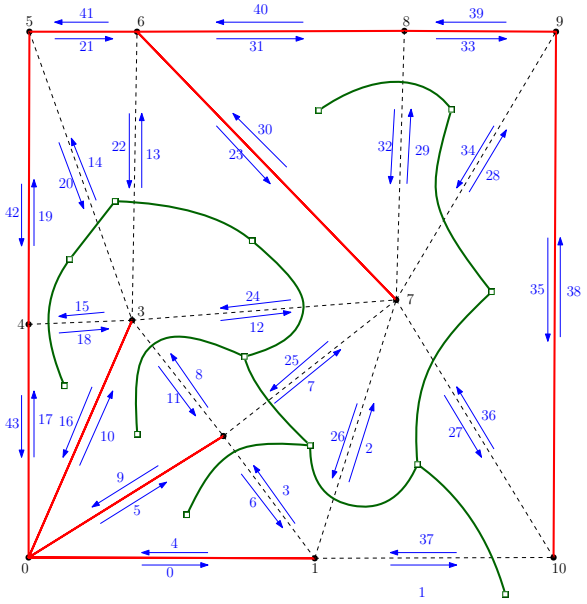

Figure 3: Representation of a triangulation as the decomposition into two spanning trees. Thick edges represent the edges of the spanning trees, red for the spanning tree of the triangulation, T, and green for the spanning tree of the dual, T'. For each edge of the triangulation, its orientation and rank after the traversal T are shown.

Some of the operations supported by PEMB that we will use in our compact half-edge representation, are:

- PEMB_VERTEX(i): return the id of source vertex of the i-th visited edge.
- PEMB_FIRST(v): return i such that when visiting the i-th edge during the traversal of T, it is the first edge whose source vertex is v.
- PEMB_LAST(v): return i such that when visiting the i-th edge during the traversal of T, it is the last edge whose source vertex is v.
- PEMB_NEXT(i): return j such that the j-th visited edge is next to the i-th edge, in counter-clockwise, of the visited edges of pemb_vertex(i) during the traversal of T. If the i-th edge corresponds to the last visited edge of pemb_vertex(i), then return PEMB_FIRST(v).
- PEMB_PREV(i): return j such that the j-th visited edge is previous to the i-th edge, in counter-clockwise, of the visited edges of pemb_vertex(i) during the traversal of T. If the i-th edge corresponds to the first visited edge of pemb_vertex(i), then return PEMB_LAST(v).
- PEMB_MATE(i): return j such that we process the same edge i-th and j-th during the traversal of T;
- PEMB_DEGREE(v): return the number of edges incident to vertex v.
- PEMB_FIRST_DUAL(f): return the position of the first visited edge incident to face f during the traversal of T.
- PEMB_GET_FACE(e): return the id of the face incident to edge e.

2.4 The Polylla algorithm

The Polylla mesh generator [23] takes an initial triangulation as input $\tau = (V, E)$ to generate a polygonal mesh $\tau' = (V, E')$. Any triangulation works. The algorithm merges triangles to generate polygons of arbitrary shape (convex and non-convex shapes). To understand how the algorithm works, we must introduce the longest-edge propagation path and terminal-edge regions.

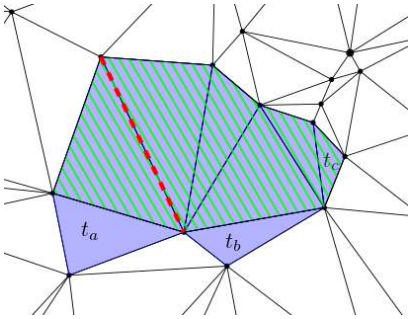

Figure 4: Example of a longest-edge propagation path of a triangle and a terminal-edge region. Dashed edges are the terminal-edge. The marked polygon is a terminal-edge region formed by the union of the triangles belonging to the Lepp(t_a), Lepp(t_b), and Lepp(t_c). The triangles with the line pattern correspond to Lepp(t_c).

Definition 1 ***Longest-edge propagation path*** *[35] For any triangle t_0 of any conforming triangulation τ, the Longest-Edge Propagation Path of t_0 (Lepp(t_0)) is the ordered list of all the triangles $t_0, t_1, t_2, ..., t_{n-1}$, such that t_i is the neighbor triangle of t_{i-1} by the longest edge of t_{i-1}, for $i = 1, 2, ..., n$. The longest-edge adjacent to t_n and t_{n-1} is called* ***terminal-edge****.*

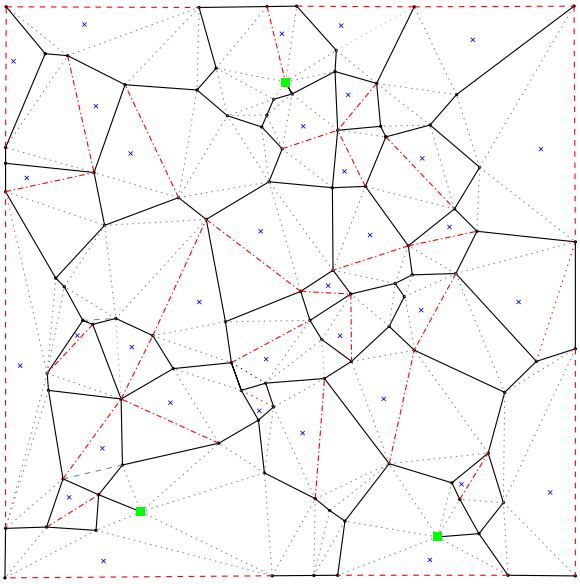

Figure 5: The output of the label phase to generate terminal-edge regions. Black lines are frontier-edges, and dotted gray lines are internal-edges. Terminal-edges are red dashed lines. Since terminal-edges can be inside or at the boundary of the geometric domain, dashed lines are border terminal-edges, and dotted dashed lines are internal terminal-edges. Barrier-edge tips are green squared vertices and seed triangles with a blue cross.

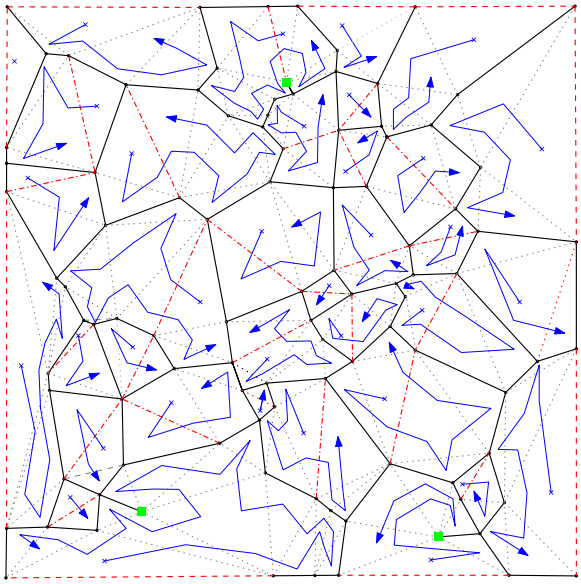

Figure 6: Traversal phase example: arrows inside terminal-regions are the paths of the algorithm during the conversion from a terminal-edge region to a polygon. The path starts at a triangle labeled as a seed triangle. Each terminal-edge region has only one seed triangle.

Definition 2 ***Terminal-edge region*** *[36] A terminal-edge region R is a region formed by the union of all triangles t_i such that Lepp(t_i) has the same terminal-edge.*

An example of both concepts is shown in Figure 4.

To convert terminal-edge regions into polygons, the Polylla algorithm works in three main phases:

i) *Label phase*: Each edge $e \in E$, adjacent to triangles t_1 and t_2, is labelled according its length as terminal-edge, internal-edge or frontier-edge:

- Internal-edge: e is the longest edge of t_1 or t_2, but not of both.
- Frontier-edge [37]: e is neither the longest-edge of t_1 nor t_2. If $t_2 = null$, e is also a frontier-edge.

Frontier-edges are the border of terminal-edge regions and so the edges of the polygons in the final mesh. A particular case of frontier-edges is barrier-edges where t_1 and t_2 belong to the same terminal-edge region. An endpoint of a barrier-edge belonging to only one frontier-edge is called a barrier-edge tip.

Figure 5 shows a triangulation with labeled edges and triangles. The labeled triangles are terminal triangles, i.e., triangles that share a terminal-edge. In the next phase, one terminal triangle per each terminal-edge is labeled as *seed triangle*.

ii) *Traversal phase*: In this phase, polygons are generated from seed triangles.For each seed triangle, the vertices of frontier-edges are traversed and stored in counter-clockwise order, delimiting the frontier of the terminal-edge region. During the traversal, some non-simple polygons with barrier-edges can be generated. Those polygons are processed later in the next phase. An example of this phase is shown in Figure 6.

iii) *Repair phase*: Non-simple polygons with barrier-edges (a polygon with dangling interior edges) are partitioned into simple polygons. Interior edges with a barrier-edge tip as an endpoint are used to split it into two new polygons, and per each new polygon, a triangle is labeled as a seed. The final output is a polygonal mesh composed of simple polygons after applying the Traversal phase to the new polygons. An example of this phase is shown in Figure 7.

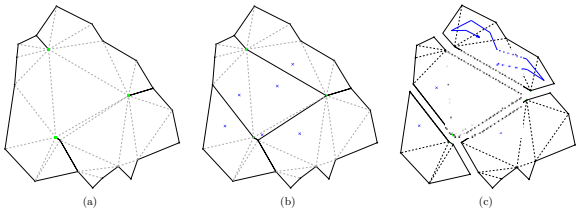

Figure 7: Example of a non-simple polygon split using interior edges with barrier-edge tips as endpoints. (a) Non-simple polygon. (b) Middle interior edges incident to barrier-edge tips are labeled as frontier-edges (solid lines), and cross-labelled triangles are stored as seed triangles. (c) The algorithm repeats the travel phase using a new seed triangle but avoiding generating the same polygon again. Source: [23].

3. HALF-EDGE DATA STRUCTURE IMPLEMENTATION

This section shows how to implement the non-compact and compact versions of the half-edge data structure. Additionally, we introduce some extra data structures needed for the Polylla algorithm.

3.1 Non-compact half-edge: AoS HALF-EDGE

To store the initial triangulation $\tau = (V, E)$ as a set of half-edges, the half-edge data structure is implemented as an array-based adjacency list, storing the vertices in an array of length $|V|$ and the half-edges in an array of length $2|E|$. Each vertex v stores its coordinate ($v.coord$), the index of an arbitrary incident half-edge ($v.hedge$), and a boolean indicating if v is incident to the outer face ($v.is\_border$). For each half-edge e its source ($e.src$) and target ($e.tgt$) vertices, twin ($e.twin$), next ($e.next$) and previous ($e.prev$) half-edges, incident face ($e.face$) and a boolean indicating if e is incident to the outer ($e.is\_border$)face are stored. The three half-edges bordering a face are stored consecutively in the array of half-edges, i.e., the half-edges of face i are stored in the indices $3i$, $3i+1$ and $3i+2$. Thus, most of the half-edge primitive queries are supported in constant time by returning the corresponding field (e.g. EDGEOFVERTEX(v) returns $v.hedge$ and ISBORDER(e) returns $e.is\_border$). More complex queries are supported as follows:

- CCWVERTEXEDGE(e): TWIN(NEXT(e)).
- CWVERTEXEDGE(e): TWIN(PREV(e)).
- INCIDENTHALFEDGE(f): Half-edge at index $3f$ in the array of half-edges.
- LENGTH(e): Euclidean distance of the coordinates of ORIGIN(e) and TARGET(e).
- DEGREE(e): Using the query CCWVERTEXEDGE(e), iterate over the neighbors of ORIGIN(e) until reaching e.

3.2 Compact half-edge

Our compact representation of the half-edge data structure has two components: *(1)* a compact representation of the initial triangulation, using PEMB, and *(2)* a non-compact vector with the coordinates of the vertices.The vertex identifiers in PEMB are not necessarily the same as the input triangulation since PEMB assigns new identifiers according to the traversal of the trees. To simplify the mapping between the components (1) and (2), the coordinate of the vertex with id i, in PEMB, is stored at the entry i of the vector of coordinates. Notice that no extra data structures are needed since PEMB provides all navigational queries to implement the half-edge data structure. Thus, the compact half-edge data structure uses $4|E| + o(|E|)$ bits for the first component and $O(|V| \log |V|)$ bits for the second component, where the $\log |V|$ term comes from the fact that at least $O(\log |V|)$ bits are necessary to represent $O(|V|)$ coordinates.

In PEMB, the edges of a face are oriented clockwise, the opposite orientation of the half-edge data structure (see Figure 3). Additionally, we notice that queries PEMB_NEXT and PEMB_PREV of PEMB have a different meaning than queries NEXT and PREV of the half-edge. The former queries refer to edges incident to a vertex, while the latter refers to edges incident to a face. It is possible to orientate the faces of PEMB counter-clockwise by traversing the primal spanning tree clockwise.

In what follows, we show how to support half-edge queries with PEMB:

- TWIN(e): PEMB_MATE(e).
- NEXT(e): PEMB_PREV(PEMB_MATE(e))
- PREV(e): PEMB_MATE(PEMB_NEXT(e))
- ORIGIN(e): PEMB_VERTEX(PEMB_MATE(e))
- TARGET(e): PEMB_VERTEX(e)
- FACE(e): PEMB_GET_FACE(e)

Additional queries are implemented as follows:

- CCWVERTEXEDGE(e): PEMB_NEXT(e)
- CWVERTEXEDGE(e): PEMB_PREV(e)
- EDGEOFVERTEX(v): PEMB_MATE(PEMB_FIRST(v))
- INCIDENTHALFEDGE(f): PEMB_FIRST_DUAL(f)
- ISBORDER(e): return true if PEMB_GET_FACE(e) returns the id of the outer face. Otherwise, return false

- LENGTH(v): Euclidean distance of the coordinates, stored in the component (2) of compact half-edge, of ORIGIN(e) and TARGET(e).
- DEGREE(v): PEMB_DEGREE(v)

3.3 Additional data structures

Before implementing the Polylla algorithm based on the half-edge data structure, we need some additional temporary data structures. To label each edge of the triangulation, we use two bitvectors, `max-edge` and `frontier-edge`, to mark the longest edge of a triangle and frontier edges, respectively. Both bitvectors are of length $2|E|$, the number of half-edges. For the seed triangles, a vector called `seed-list` stores the indices of the incident terminal-edges.

For the repair phase, we use two auxiliary arrays to avoid the duplication of the polygons, `subseed-list`, that is declared empty, and `usage bitvector`, of length $|E|$.

Finally, the output mesh is stored as a 2-dimensional array called `mesh array`, where each row stores a set of vertices representing a polygon. Notice that we do not return a compact version of the output mesh directly. Instead, after the generation of `mesh array`, we can store it in compact space by constructing its compact half-edge representation.

4. HALF-EDGE POLYLLA ALGORITHM

This section explains how to implement the Polylla algorithm using a half-edge data structure. The algorithm takes a triangulation $\tau(V, E)$ as input and generates a polygonal mesh as output. All the phases of the Polylla mesh are $O(|V|)$.

4.1 Label phase

This phase labels each edge $e \in E$ as a frontier-edge, the longest edge of a face, and/or a seed edge incident to a triangle seed. The pseudo-code of this process is shown in Algorithm 1.

The algorithm iterates over each triangle $t \in \tau$, where the edges delimiting t are obtained with the queries e = INCIDENTHALFEDGE(t), NEXT(e) and PREV(e). The edges of a triangle t are then compared, and the id of the longest one is marked in `max-edge` bitvector (lines 1–3). Afterward, the algorithm iterates over all the half-edges of τ. If a half-edge e or its twin TWIN(e) are at the geometric boundary, i.e. IS_BORDER(e)=TRUE or IS_BORDER(TWIN(e))= TRUE, or both half-edges were not marked in `max-edge`, then e is labelled as a frontier-edge (lines 4–9). Alongside, the algorithm searches for seed edges: if a half-edge e and its TWIN(e) are a terminal-edge or border terminal-edge incident to an interior face, then the algorithm labels any of the half-edges (lines 10–12).

Algorithm 1 Label phase

Input: Half-edge data structure `HalfEdge`
Output: Bitvectors `frontie-edge` and `max-edge`, and vector `seed-list`

```
 1: for all triangle t in HalfEdge do
 2:     Mark the longest edge of t in max-edge
 3: end for
 4: for all half-edge e in HalfEdges do
 5:     if e and TWIN(e) are not in max-edge then
 6:         Mark e in frontier-edge
 7:     else if e or TWIN(e) are border edges then
 8:         Mark e in frontier-edge
 9:     end if
10:     if e is terminal-edge or border terminal-edge then
11:         Store the id of e or TWIN(e) in the seed list
12:     end if
13: end for
```

4.2 Traversal Phase

In the second phase, the algorithm uses seed edges generated in the previous phase to build terminal-edge regions. For each generated region R, its vertices are stored in counter-clockwise order in a set P.

For each seed half-edge e in `seed list`, Algorithm 2 is called. The algorithm iterates in clockwise order around ORIGIN(e) until it finds a frontier-edge e_{init}, an edge that will be part of the final polygonal mesh (lines 1–7). Once a frontier-edge is found, the algorithm iterates, using the query CWVERTEXEDGE(), over the edges of the region R until reaching the next frontier-edge in counter-clockwise order (lines 8–14). Each discovered frontier-edge's source vertex is added to the output polygon (lines 7 and 13). This process ends when all boundary vertices of R are stored in P

Each polygon P is checked if it is a simple or non-simple polygon. The algorithm iterates over all vertices in P, looking for three consecutive vertices, v_i, v_j, and v_k with $v_i == v_k$. If true, then v_j is a barrier-edge tip, and the polygon is a non-simple polygon. If the polygon is simple, it is stored in the `mesh array`. If not, it is sent to the repair phase.

4.3 Repair Phase

The repair phase works similarly to the label and the travel phases but is limited to the triangles of a non-simple terminal-edge region. In summary, the algorithm labels an internal-edge e incident to each barrier-edge tip as frontier-edge and repeats the travel phase using the triangles adjacent to e to generate two new polygons (see Algorithm 3).

Given a non-simple polygon P, for each barrier-edge tip $b \in P$, the algorithm searches for the barrier-edge incident to b (lines 4 – 7). To do that, the algorithm

Algorithm 2 Polygon construction

Input: Seed edge e of a terminal-edge region
Output: Arbitrary shape polygon P
1: $P \leftarrow \emptyset$
2: **while** e is not a frontier-edge **do**
3: $e \leftarrow$ CWvertexEdge(e)
4: **end while**
5: $e_{init} \leftarrow e$
6: $e_{curr} \leftarrow$ next(e)
7: $P \leftarrow P \cup$ ORIGIN(e)
8: **while** $e_{init} \neq e_{curr}$ **do**
9: **while** e_{curr} is not a frontier-edge **do**
10: $e_{curr} \leftarrow$ CWVERTEXEDGE(e)
11: **end while**
12: $e_{curr} \leftarrow$ NEXT(e_{curr})
13: $P \leftarrow P \cup$ ORIGIN(e_{curr})
14: **end while**
15: **return** P

uses the query EDGEOFVERTEX(b to get a starting half-edge e incident to b from where the incident half-edges of b are traversed using CWVERTEXEDGE(e) until getting a frontier-edge of b.

Afterward, the algorithm chooses one of the incident internal-edges of b to split the polygon in two (lines 8 – 10). To choose an internal-edge, the algorithm calculates the number of incident edges as DEGREE(b)−1 (−1 because of the frontier-edge incident to b), and circles around b (DEGREE(b)−1)/2 times to split the polygon evenly. The target internal-edge e is labelled as frontier-edge, marking its two half-edges in the `frontier-edge` bitvector, labelled as `True` in `usage bitvector`, to mark them as visited during this phase, and stored in `subseed-list` to use them later as seed edges to generate a new polygon (lines 11 – 13).

For each half-edge e inside `subseed-list` and `usage bitvector`[e] = `True`, the algorithm repeats the travel phase (line 18) to build a new polygon, marking `usage bitvector`[e] = `False` after building it to avoid the generation of the same polygon more than once.

The final set of simple polygons is returned and stored as part of the mesh in the `mesh array`.

5. EXPERIMENTS

5.1 Implementation

The implementations of the Polylla algorithm and the compact data structures are in C++.[1] The algorithm described in Section 4 was implemented as a class that calls virtual methods of the abstract class `Mesh`. This abstract class contains all the methods of the half-edge data structure, shown in Section 2.2. Those methods were implemented into two child classes: `Triangulation`, that contains the implementations of the functions shown in Section 3.1, and

[1]The source code of our implementations are available at https://github.com/ssalinasfe/Compact-Polylla-Mesh

Algorithm 3 Non-simple polygon reparation

Input: Non-simple polygon P
Output: Set of simple polygons S
1: `subseed list` as L_p and `usage bitarray` as A
2: $S \leftarrow \emptyset$
3: **for all** barrier-edge tip b in P **do**
4: $e \leftarrow$ EDGEOFVERTEX(b)
5: **while** e is not a frontier-edge **do**
6: $e \leftarrow$ CWVERTEXEDGE(e)
7: **end while**
8: **for** 0 **to** (DEGREE(b) - 1)/2 **do**
9: $e \leftarrow$ CWVERTEXEDGE(e)
10: **end for**
11: Label e as frontier-edge
12: Save half-edges h_1 and h_2 of e in L_p
13: $A[h_1] \leftarrow$ `True`, $A[h_2] \leftarrow$ `True`
14: **end for**
15: **for all** half-edge h in L_p **do**
16: **if** $A[h]$ is `True` **then**
17: $A[h] \leftarrow$ `False`
18: Generate new polygon P' starting from h using Algorithm 2.
19: Set as `False` all indices of half-edges in A used to generate P'
20: **end if**
21: $S \leftarrow S \cup P'$
22: **end for**
23: **return** S

`CompactTriangulation`, that contains the implementations of the functions shown in Section 3.2. The `CompactTriangulation` class encapsulates the class `Pemb`, that contains the implementation of PEMB using the library SDSL [38].[2]

5.2 Datasets

To test our implementations, we generated several Delaunay triangulations from random point sets inside a square of dimensions $10,000 \times 10,000$. To see the behaviour of Polylla meshes using another dataset see [23]. For the generation of the triangulations, we used the 2D package of the software CGAL [39]. An example of the generated meshes is shown in Figure 8.

5.3 Experimental setup

To run the experiments, a machine with processor Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz and main memory of 126 GB was used. To measure the memory consumption, the library *malloc count*[3] was used. From this library, we use the function `malloc_count_peak()` to obtain the peak of memory consumption and `malloc_count_current()` to obtain the memory used to store the generated polygonal mesh. The size of PEMB was obtained with the support of the SDSL library. Memory usage and the execution time were calculated using the Chronos library

[2]The original implementation of PEMB is available at https://github.com/jfuentess/sdsl-lite
[3]https://panthema.net/2013/malloc_count/

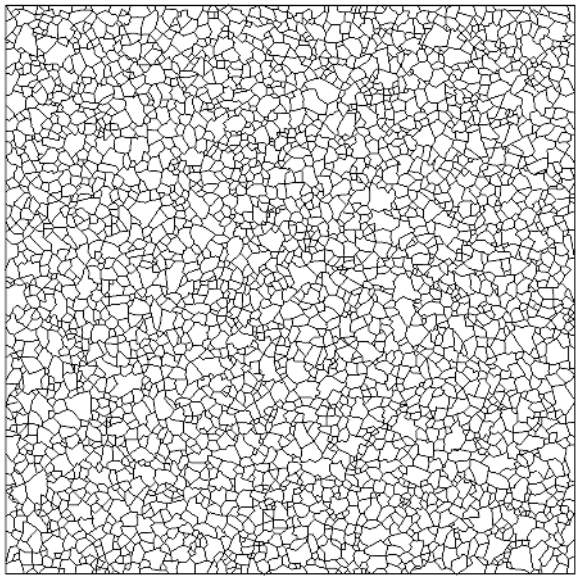

Figure 8: Example of a Polylla mesh generated from a Delaunay triangulation over $10k$ random vertices.

		AoS		Compact	
#V	GDT	GHF	GP	GHF	GP
10M	4.67	0.14	0.58	0.26	23.86
15M	7.19	0.20	0.87	0.39	36.31
20M	9.58	0.27	1.15	0.52	48.82
25M	11.92	0.33	1.43	0.65	60.90
30M	14.70	0.40	1.73	0.78	74.06
35M	16.08	0.47	2.01	0.92	87.77
40M	19.17	0.52	2.25	1.05	98.67

Table 1: Time comparison in minutes of the Delaunay triangulation generation (GDT), the half-edge data structure generation (GHF), and the Polylla mesh generation (GP).

without considering the time to load the input triangulations. Each experiment was run five times, and the average was reported. We generated meshes from 10 million vertices to 40 million.

5.4 Results

Half-edge data structures construction. Table 1 and Figure 9 show the time needed to construct the half-edge data structure (GHF) and to generate the polygonal mesh using the Polylla algorithm (GP). The construction of AoS HALF-EDGE data structure is 1.95x faster than the construction of compact half-edge. In the same line, the generation of a polygonal mesh using AoS HALF-EDGE is 42.7x faster than the generation using the compact half-edge. Additionally,

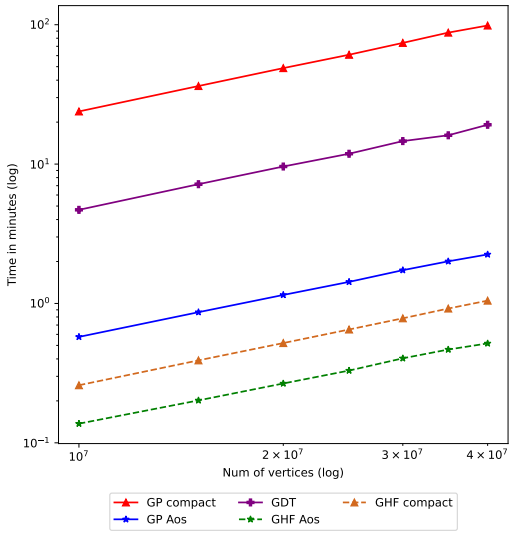

Figure 9: (LogLog plot) Running time, in minutes, to generate the data structures. The continuous line is the time to generate Polylla mesh using compact half-edge and AoS HALF-EDGE (GP compact and GP AoS, respectively), and the Delaunay triangulations (GDT) using CGAL. Dashed lines correspond to the time to generate compact half-edge and AoS HALF-EDGE (GHF compact and GHF AoS, respectively).

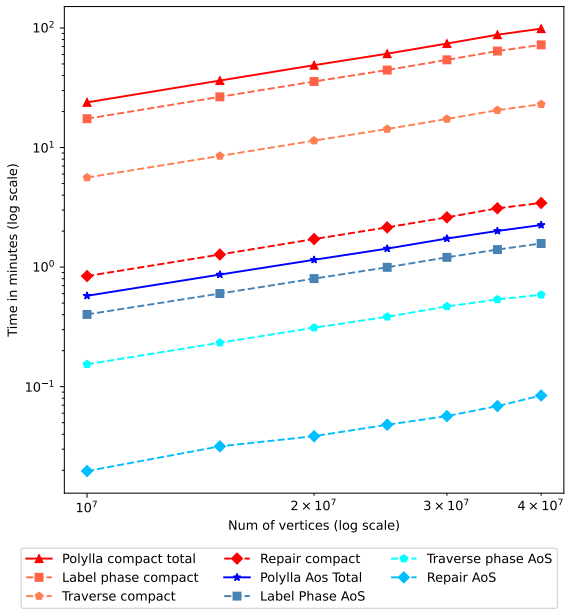

Figure 10: (LogLog plot) Running time, in minutes, that takes each phase of the Polylla algorithm using AoS Half-edge and compact Half-edge. The continuous line is the total sum of the algorithm, while the dashed lines show the time for each phase.

	AoS			Compact					
#V	HF	GHF	GP	HF	Pemb	Coord	GHF	GP	Polylla
10M	1.79	3.02	2.25	0.17	0.02	0.15	1.96	0.63	0.60
15M	2.68	4.53	3.49	0.25	0.03	0.23	2.95	1.06	0.95
20M	3.58	6.03	4.50	0.34	0.04	0.30	3.93	1.26	1.20
25M	4.47	7.54	5.54	0.42	0.05	0.38	4.91	1.49	1.44
30M	5.36	9.05	6.97	0.51	0.06	0.45	5.90	2.12	1.91
35M	6.26	10.56	7.99	0.59	0.07	0.53	6.88	2.32	2.15
40M	7.15	12.07	9.01	0.68	0.08	0.60	7.86	2.53	2.40

Table 2: Memory usage in gigabytes by the algorithm. HF is the memory used to store the half-edge data structure, GHF is the memory cost to generate the triangulation, and GP is the memory cost to generate the Polylla mesh. In the case of the compact HF, HF is also the memory used to store the vertices coordinates (Coord) and the Pemb data structure. The Polylla column is the memory used to store the Polylla mesh.

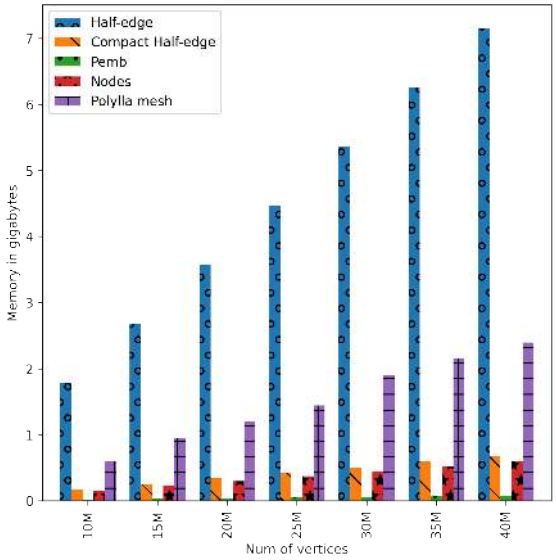

Figure 11: Memory, in gigabytes, used to store the data structures.

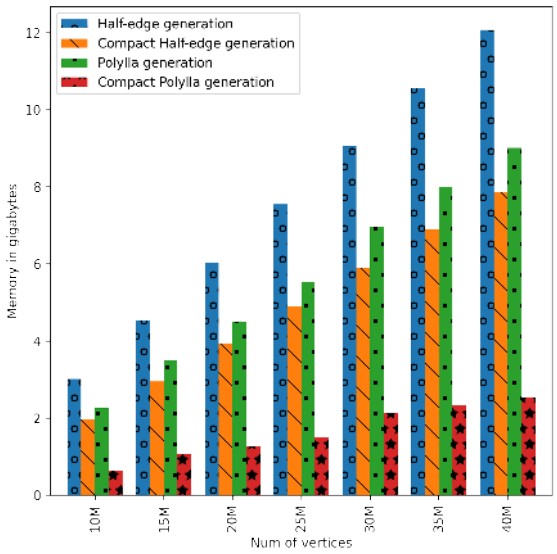

Figure 12: Peaks of memory achieved, in gigabytes, during the generation of the data structures.

as a reference, we include the time needed by CGAL to generate a Delaunay mesh from a random point set.

Phases of the Polylla algorithm. Figure 10 shows the running time to generate polygonal meshes with Polylla. Despite the data structure used to generate the Polylla meshes, all phases of Polylla have the same growth. Remember that all the phases of Polylla have a complexity of $O(|V|)$. Notice that each phase uses different queries. The most costly phase is the label phase, which visits all faces in the triangulation, calculates the length of the edges using the queries `next(·)` and `prev(·)`, and then labels the edges using queries `is_border(·)` and `twin(·)`. The second costly phase is the traversal phase. This phase uses the queries `origin(·)`, `next(·)` and `CWvertexEdge(·)` to generate each polygon. During this phase, all the edges of the triangulation are revisited. The repair phase is the fastest, as it is only used by the 1% of the polygons [23] generated in the traversal phase. One particular query used during the repair phase is the query `degree(·)` to calculate the middle internal-edge.

Memory usage. Results of the memory usage are shown in Table 2. To calculate the memory usage to generate the data structure, we compute the memory peak of the algorithm (the columns with the prefix "G"). The memory usage for the triangulation once the half-edge data structure was created shown without the "G" prefix. It can be observed that the memory usage to generate the polygonal meshes (GP) using the AoS Polylla requires 3.49x more memory than

the compact version (compact GP). In the case of the generation of the data structures (GHF), the AoS half-edge generation (AoS GHF) takes 3.49x more memory than the compact version (compact GHF). The peak of memory usage is shown in Figure 12.

After the half-edge data structure (compact or non-compact) was initialized, the memory usage decreased because several temporal information was not needed anymore. The memory usage during the application of the Polylla phases is shown in Figure 11. The topological information of a triangulation can be compacted by a 99%[4]. respect to AoS HALF-EDGE, that is, without considering the memory usage to store the coordinates.

Most of the memory used by the compact and non-compact half-edge data structure is related to the coordinates of each vertex of the triangulation. The memory to store the compact triangulation is distributed in 88.67% for the point coordinates and 11.33% for the half-edge data structure. As the position of the vertices is float values, they can not be compacted easily.

Despite the fact that compact Half-edge is slower than AoS HALF-EDGE, we argue that in scenarios where the non-compact representation of half-edge does not fit in main memory while compact Half-edge does, the latter will be faster due to the memory hierarchy effect. Empirical evidence for this scenario, but applied on general planar embeddings, can be found in [21].

6. CONCLUSIONS AND FUTURE WORK

We have shown that the succinct data structure known as PEMB is useful for representing polygon meshes and generating tri-to-polygon meshes. Using PEMB, the space usage of a mesh is largely reduced, allowing to process of huge meshes.

One of the advantages of PEMB is that their queries are reduced to simple and fast operations over three static bitvectors. We expect a future development where those operations work in a GPU architecture, as GPU parallelization could take advantage of the low memory usage of PEMB. Additionally, in this work, we used only 7 of the 17 queries supported by PEMB [34]. As future work, we will explore PEMB operations to study the possibility of implementing more queries such as vertex insertion and edge flipping.

In the case of the Polylla algorithm, we showed a new half-edge version that is easier to read and implement in any language. Future work involves taking advantage of this implementation to develop a parallel version of Polylla and extends this work to a 3D using an extension of the half-edge data structure.

[4]Obtained by dividing columns *Pemb* and *AoS HF* in Table 2

7. ACKNOWLEDGMENT

This work was partially funded by ANID doctoral scholarship 21202379 (first author), ANID FONDECYT grant 11220545 (second author) and ANID FONDECYT grant 1211484 (third author).

References

[1] Attene M., Campen M., Kobbelt L. "Polygon Mesh Repairing: An Application Perspective." *ACM Comput. Surv.*, vol. 45, no. 2, Mar. 2013

[2] Huisman O., de By R. *Principles of geographic information systems : an introductory textbook*. Oxford University Press, 01 2009

[3] Ho-Le K. "Finite element mesh generation methods: a review and classification." *Computer-Aided Design*, vol. 20, no. 1, 27–38, 1988

[4] Ghosh S., Mallett R. "Voronoi cell finite elements." *Computers & Structures*, vol. 50, no. 1, 33–46, 1994

[5] Beir L., Brezzi F., Arabia S. "Basic principles of Virtual Element Methods." *Mathematical Models and Methods in Applied Sciences*, vol. 23, 199–214, 2013

[6] Chi H., da Veiga L.B., Paulino G. "Some basic formulations of the virtual element method (VEM) for finite deformations." *Computer Methods in Applied Mechanics and Engineering*, vol. 318, 148–192, 2017

[7] Park K., Chi H., Paulino G.H. "On nonconvex meshes for elastodynamics using virtual element methods with explicit time integration." *Computer Methods in Applied Mechanics and Engineering*, vol. 356, 669–684, 2019

[8] Attene M., Biasotti S., Bertoluzza S., Cabiddu D., Livesu M., Patanè G., Pennacchio M., Prada D., Spagnuolo M. "Benchmarking the geometrical robustness of a Virtual Element Poisson solver." *Mathematics and Computers in Simulation*, vol. 190, 1392–1414, 2021

[9] Sorgente T., Prada D., Cabiddu D., Biasotti S., Patanè G., Pennacchio M., Bertoluzza S., Manzini G., Spagnuolo M. "VEM and the Mesh." *CoRR*, vol. abs/2103.01614, 2021

[10] Bommes D., Lévy B., Pietroni N., Puppo E., Silva C., Tarini M., Zorin D. "Quad-mesh generation and processing: A survey." *Computer Graphics Forum*, vol. 32, pp. 51–76. 2013

[11] Owen S.J., Staten M.L., Canann S.A., Saigal S. "Q-Morph: an indirect approach to advancing front quad meshing." *International journal for numerical methods in engineering*, vol. 44, no. 9, 1317–1340, 1999

[12] Owen S.J. "A survey of unstructured mesh generation technology." *IMR*, vol. 239, 267, 1998

[13] Johnen A. *Indirect quadrangular mesh generation and validation of curved finite elements*. Ph.D. thesis, Université de Liège, Liège, Belgique, 2016

[14] Lee C., Lo S. "A new scheme for the generation of a graded quadrilateral mesh." *Computers Structures*, vol. 52, no. 5, 847–857, 1994

[15] Remacle J.F., Lambrechts J., Seny B., Marchandise E., Johnen A., Geuzainet C. "Blossom-Quad: A non-uniform quadrilateral mesh generator using a minimum-cost perfect-matching algorithm." *International Journal for Numerical Methods in Engineering*, vol. 89, no. 9, 1102–1119, 2012

[16] Merhof D., Grosso R., Tremel U., Greiner G. "Anisotropic quadrilateral mesh generation : an indirect approach." *Advances in Engineering Software*, vol. 38, no. 11/12, 860–867, 2007

[17] Barber C.B., Dobkin D.P., Huhdanpaa H. "The Quickhull algorithm for convex hulls." *Acm Transactions on Mathematical Software*, vol. 22, no. 4, 469–483, 1996

[18] Shewchuk J.R. "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator." M.C. Lin, D. Manocha, editors, *Applied Computational Geometry Towards Geometric Engineering*, pp. 203–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996

[19] Si H. "An Introduction to Unstructured Mesh Generation Methods and Softwares for Scientific Computing." Course, 7 2019

[20] Navarro G. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016

[21] Ferres L., Fuentes-Sepúlveda J., Gagie T., He M., Navarro G. "Fast and compact planar embeddings." *Computational Geometry*, vol. 89, 101630, 2020

[22] Muller D., Preparata F. "Finding the intersection of two convex polyhedra." *Theoretical Computer Science*, vol. 7, no. 2, 217–236, 1978

[23] Salinas-Fernández S., Hitschfeld-Kahler N., Ortiz-Bernardin A., Si H. "POLYLLA: polygonal meshing algorithm based on terminal-edge regions." *Engineering with Computers*, May 2022

[24] Tutte W.T. "A Census of Planar Maps." *Canadian Journal of Mathematics*, vol. 15, 249–271, 1963

[25] Alumbaugh T.J., Jiao X. "Compact Array-Based Mesh Data Structures." B.W. Hanks, editor, *Proceedings of the 14th International Meshing Roundtable*, pp. 485–503. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005

[26] Aleardi L.C., Devillers O., Mebarki A. "Catalog-based representation of 2D triangulations." *International Journal of Computational Geometry & Applications*, vol. 21, no. 04, 393–402, 2011

[27] Kallmann M., Thalmann D. "Star-Vertices: A Compact Representation for Planar Meshes with Adjacency Information." *Journal of Graphics Tools*, vol. 6, no. 1, 7–18, 2001

[28] Aleardi L.C., Devillers O., Rossignac J. "ESQ: Editable SQuad Representation for Triangle Meshes." *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images*, pp. 110–117. 2012

[29] Gurung T., Rossignac J. "SOT: Compact Representation for Tetrahedral Meshes." *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, p. 79–88. Association for Computing Machinery, New York, NY, USA, 2009

[30] Ferres L., Fuentes-Sepúlveda J., Gagie T., He M., Navarro G. "Fast and Compact Planar Embeddings." *WADS*. 2017

[31] Aleardi L.C., Devillers O. "Array-based compact data structures for triangulations: Practical solutions with theoretical guarantees." *Journal of Computational Geometry*, vol. 9, no. 1, 247–289, 2018

[32] Baumgart B.G. "A Polyhedron Representation for Computer Vision." *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, p. 589–596. New York, NY, USA, 1975

[33] Berg M.d., Cheong O., Kreveld M.v., Overmars M. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edn., 2008

[34] Fuentes-Sepúlveda J., Navarro G., Seco D. "Navigating planar topologies in near-optimal space and time." *Computational Geometry*, vol. 109, 101922, 2023

[35] Rivara M.C. "New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations." *International Journal for Numerical Methods in Engineering*, vol. 40, no. 18, 3313–3324, 1997

[36] Alonso R., Ojeda J., Hitschfeld N., Hervías C., Campusano L. "Delaunay based algorithm for finding polygonal voids in planar point sets." *Astronomy and Computing*, vol. 22, 48 – 62, 2018

[37] Hervías C., Hitschfeld-Kahler N., Campusano L.E., Font G. "On Finding Large Polygonal Voids Using Delaunay Triangulation: The Case of Planar Point Sets." *Proceedings of the 22nd International Meshing Roundtable*, pp. 275–292. 2013

[38] Gog S., Beller T., Moffat A., Petri M. "From Theory to Practice: Plug and Play with Succinct Data Structures." *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pp. 326–337. 2014

[39] Yvinec M. "2D Triangulations." *CGAL User and Reference Manual*. CGAL Editorial Board, CGAL project, 5.3.1 edn., 2021