Hierarchical representation for rasterized planar face complexes

Guillaume Damiand, Aldo Gonzalez-Lorenzo, Jarek Rossignac, Florent Dupont

Hierarchical Representation for Rasterized Planar Face Complexes

Guillaume Damiand[a,*], Aldo Gonzalez-Lorenzo[a], Jarek Rossignac[b], Florent Dupont[a]

[a] *Univ Lyon, CNRS, LIRIS, UMR5205, F-69622 France*
[b] *School of Interactive Computing, Georgia Institute of Technology, Atlanta, USA*

ARTICLE INFO

Article history:
Received 15 May 2018

Keywords: Planar polygonal meshes, Irregular representation, Hierarchical representation, Combinatorial maps, Compact representation, Topology preserving rasterization.

ABSTRACT

A useful example of a Planar Face Complex (PFC) is a connected network of streets, each modeled by a zero-thickness curve. The union of these decomposes the plane into faces that may be topologically complex. The previously proposed rasterized representation of the PFC (abbreviated rPFC) (1) uses a fixed resolution pixel grid, (2) quantizes the geometry of the vertices and edges to pixel-resolution, (3) assumes that no street is contained in a single pixel, and (4) encodes the graph connectivity using a small and fixed number of bits per pixel by decomposing the exact topology into per-pixel descriptors. The hierarchical (irregular) version of the rPFC (abbreviated hPFC) proposed here improved on rPFC in several ways: (1) It uses an adaptively constructed tree to eliminate the "no street in a pixel" constraint of the rPFC, hence making it possible to represent exactly any PFC topology and (2) for PFCs of the models tested, and more generally for models with relatively large empty regions, it reduces the storage cost significantly.

1. Introduction

Consider a planar graph, G, that is embedded in the plane and comprises a ***connected*** network of finite and possibly curved edges and their bounding vertices. For example, each edge may represent a street and each vertex may represent a street junction. Their union decomposes the plane into faces that may be topologically complex. For example, G may have ***multi-edges*** (more than one edge joining any given pair of vertices). Furthermore, a face may contain, in its boundary, ***cracks*** (edges that bound a single face), ***dead-ends*** (vertices that bound a single edge—a crack), and ***loops*** (edges that start and end at the same vertex). The unbounded face is called ***exterior***. An example is shown in Fig. 1. We use the term ***Planar Face Complex*** (PFC) for such an arrangement.

Many applications need to represent and to traverse a PFC. Examples include street networks in ***Geographic Information***

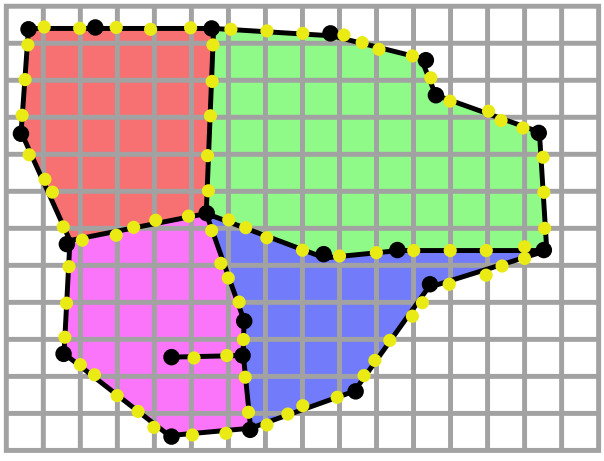

Figure 1. A Planar Face Complex (PFC), that has 21 vertices, 24 edges and 5 faces. (The portion of the external face shown here is white.) The chosen pixel resolution is valid: It ensures that no edge lies entirely in a single pixel. The domain, which includes all but the external face, comprises 192 pixels, of which 126 are empty. The 79 crossings (where an edge enters a pixel) are shown as small yellow disks.

*Corresponding author
e-mail: guillaume.damiand@liris.cnrs.fr (Guillaume Damiand), aldo.gonzalez-lorenzo@liris.cnrs.fr (Aldo Gonzalez-Lorenzo), jarek@cc.gatech.edu (Jarek Rossignac), florent.dupont@liris.cnrs.fr (Florent Dupont)

Author version of paper "Hierarchical Representation for Rasterized Planar Face Complexes; Damiand G., Gonzalez-Lorenzo A., Rossignac J., Dupont F.; Computers & Graphics (C&G), Volume 74, pages 161-170, August 2018". Thanks to Elsevier.

System (GIS) [1], geological models [2], overlapping SVG elements [3], and multi-material structures [4].

Different solutions have been proposed to represent PFCs. Some approaches describe the connectivity of the graph [5, 6, 7] explicitly. This may yield a high storage cost for complex graphs. Other approaches use an image format (regular grid of pixels) to describe a rasterized approximation of the PFC [8, 9, 10, 11], which assigns each pixel to a different face, without attempting to capture the topology inside the ***shared*** pixels that contains one or more edges.

The recently proposed ***rasterized Planar Face Complex*** (rPFC) [12] unifies these approaches by defining a compact representation of the topology of the PFC that decomposes it into per-pixel descriptors, each using a short string of bits to encode the topology of the intersection of the PFC with a pixel.

The rPFC model has many advantages: (1) It represents graph connectivity exactly and hence supports exact topological graph traversal; (2) It provides spatial indexing to both quantized geometry and exact topology; (3) It can represent non-trivial topology in a pixel (such as dangling edges, multiple vertices and multiple connected components); (4) It requires only a few bits per pixel.

However, the rPFC has a drawback: It cannot represent a graph that has an edge that fits entirely inside a pixel. Hence, to represent a graph with some relatively small edges, we either must use a high-resolution grid (see Fig. 1), which increases storage cost, or must simplify the graph by collapsing small edges in a preprocessing step, which implies the loss of the original topology. Furthermore, when the rPFC encoding stores a topology descriptor for each ***private*** pixel (a pixel that lies entirely in a face), the rPFC storage of large clusters of private pixels is wasteful.

1.1. Motivation

Our overarching motivation is to reduce the storage size of this graph, while preserving the benefits provided by the previously proposed rPFC representation, namely (1) random access and traversal (RAT) at constant amortized time (CAT) cost and (2) constant cost localization of the edges and vertices that intersect any given pixel. We also wish to provide efficient support for distributed processing, window-stream processing, and progressive refinements.

We believe that the above characteristics are important for navigation, query, and maintenance applications of huge databases of planar graphs, which may represent the geometry and connectivity of streets, rivers, or utility networks.

Our second main motivation is to use the 2D representation as the main tool to define 3D compact representation. This paper is the first step, necessary for the definition of a compact representation of 3D meshes.

1.2. Contribution

The high-level, novel contribution reported here is the combination of a hierarchical representation with the rPFC (per-pixel) encoding of geometry and connectivity. In this paper, we define the ***hierarchical rasterized Planar Face Complex*** representation (***hPFC***) which addresses the drawback of the previous rPFC.

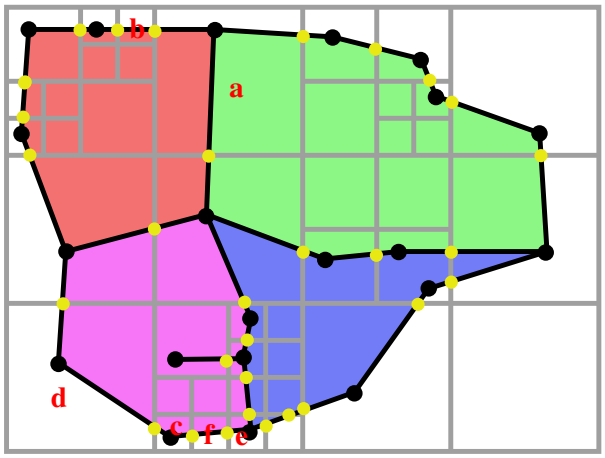

Figure 2. The hPFC of the PFC shown in Fig. 1 uses only 42 pixels (6 being labeled). Only 17 of these are empty. It has only 30 crossings.

The proposed hPFC is essentially a tree. Hence, our solution includes a quadtree as a special case. At the coarsest level, it is an rPFC, A. But some of the pixels of A, instead of containing the bit-string that encodes the local topology of the PFC, contain an index to a refined rPFC of the portion of the PFC inside that pixel. Such a more detailed rPFC, B, may, in turn contain pixels which, each, refer to even finer rPFC, C, and so on recursively.

This irregular representation allows us to remove the "no small edge" constraint imposed by the rPFC: When an edge fits entirely in a pixel, the pixel is subdivided.

Moreover, using an irregular (hierarchical) grid allows to reduce the storage cost of large clusters of private pixels.

For example, the rPFC shown in Fig. 1 uses a grid of 192 pixels and involves 79 ***crossings*** (points where an edge of G crosses a pixel border). A coarser grid would produce an invalid rasterization in which at least one edge is contained in a single pixel. As shown in Fig. 2, using the irregular grid of an hPFC solves the problem: The same PFC may be encoded as an hPFC that uses only a total of 42 pixels and involves only 30 crossings.

1.3. Organization

The paper is organized as follows. In Sect. 2, we review the rPFC model and discuss other relevant prior art. In Sect. 3, we present the hPFC model and the details of the operators needed to navigate through the PFC using its hPFC representation. In Sect. 4 we give a compact encoding of hPFC that provides a good time-complexity for traversal operators, while allowing to navigate through the graph without needing to decode the whole data-structure, but only the current pixel. In Sect. 5, we present experimental results, comparing our new solution with the previous regular version.

2. Prior Art

2.1. Data-Structures for Polygonal Meshes

A variety of edge-based data structures have been proposed in order to represent polygonal meshes, such as Combinatorial

Maps, Corner Table, Doubly Connected Edge List, Half-Edge, Surface Mesh... [5, 6, 13, 7, 14, 15, 16, 17, 18, 19]. They differ in their storage cost, in the type of operators that they support, and in the topological restrictions that they impose on the mesh. Many are reviewed in [20, 21].

These data structures provide ***Random Access and Traversal*** (***RAT***) of the meshes, often in constant time, or sometimes in ***Constant Amortized Time*** (***CAT***). Their main drawback is to use a large number of bits per element (edge, vertex), which limits their applicability and performance for complex meshes.

Some solutions used rasterized images, where each pixel stores the color of the region that contains its center. The image can be compressed, for example by using RLE (Run Length Encoding). But the digitization does not represent street networks, removes all cracks and dead ends, and can disconnect regions.

Rasterized images were used in [22] to accelerate the rendering of antialiased vector graphics. That approach uses a coarse lattice in which each cell contains a variable-length encoding of the graphics primitives that overlap it. The proposed hPFC extends this previous work by capturing the connectivity (incidence and ordering) of the graph in a constant-length per-cell format and hence providing support for RAT operators.

In [9, 23], a solution stores the crossing vertices between the mesh and an inter-pixel grid, and recompute (explicitly or implicitly) a simplified topology of the mesh. Such a representation can be used to accelerate Boolean operations [4]. But it only represents an approximated topological description of the mesh.

2.2. Compact Representations of Polygonal Meshes

Several compression schemes propose to encode local mesh connectivity by using a few bits per element for polygon graphs [24, 25] and for triangle meshes [26, 27, 28, 29, 30, 31, 7, 32].

Often, the connectivity information is broken into a chunk per face, per edge, or per vertex. For example, the 2D version of Tetstreamer [33] organizes triangle faces into topological rings and divides connectivity information into one bit per edge (for some edges) and one or several bits per vertex. But extensions of this approach to more general (PFC) graphs would be challenging and the representation more expensive. More importantly, such schemes assume that the bits of the mesh encoding is received in a specific order. This compressed format must be decompressed first and converted into a more expensive format that is suitable for RAT in CAT.

More recent representations offer a much low storage costs while still supporting RAT in CAT for the most common access and traversal operations. For example, the Zipper format is restricted to triangle meshes, but uses on average only 6 bits per triangle and can be constructed in linear space and time [34]. Such representations rely on a specific ordering of vertices. The streamable version, Grouper [32], of this approach stores about two vertex-references per triangle.

2.3. Hierarchical Representations of Polygonal Meshes

Many hierarchical solutions have been defined in order to reduce the memory space used in order to represent a mesh such as for example quadtrees [8, 35]. [36] proposes a progressive mesh representation, a new scheme which allows to store and to transmit arbitrary triangle meshes. Several other hierarchical and pyramidal models were defined and used for example to represent multiresolution terrain models [37]. In [38], a compressed encoding of 3D triangular meshes is defined, based on a hierarchy and an encoding of split operators, which allows to encode both manifold meshes but also "triangle soups". In [39], a compressed random-access tree is used in order to represent spatially coherent data. But these representations are either for grid of pixels, or for triangle meshes.

Quadtrees were also used to represent a set of points [40, 41] or of line-segments [42, 43, 44]. These representations do not capture connectivity. The MX quadtree [45] does capture the connectivity of simple polygons, but does not support junction vertices with more than two incident edges. Hence, these previously proposed hierarchical representations cannot be used to describe the connectivity of PFCs.

PM-quadtrees [46] allow us to represent PFCs. In that approach, the model is split recursively into chunks that are each *sufficiently simple*. The three variants proposed in [46] use different criteria to define *sufficiently simple*, but, similarly to [35], require to have no more than one vertex in any quadtree leaf. Our solution proposed here uses a slightly less constraining criterion: we allow more than one connected component per leaf, each having at most one vertex in the leaf. This added flexibility may reduce the need to subdivide a pixel, and hence the overall storage cost, in situations, such as thin regions or constrictions in a face, in which a vertex lies close to another vertex that is not connected to it by edges in the leaf. Another difference is the generality of our solution: we are not limited to quadtrees, but can instead use any image size as input and any image size for refined images. Finally, graph information in hPFC in not only stored in leaves, like in PM-quadtrees, but also in inner nodes. Indeed, each node of our hPFC is a rPFC that can mix word pixels describing local part of the PFC and refined pixels that link to refined images.

2.4. rPFC

In this section, we summarize the aspects of the rPFC definition that are important to understand our contribution. We encourage the reader to consult the original work [12] for further details.

Consider a regular grid of ***pixels*** (which we define to be open faces, which do not include their borders), separated by ***roads*** (relatively open edges, which do not contain their vertices), which meet at ***crossroads*** (vertices, each having four incident cross-roads and four incident pixels). Each pixel P is incident upon four roads: west, north, east and south (labeled 'W', 'N', 'E', and 'S'). A pixel is ***stabbed*** (by G) when its intersection with G is not empty. It is ***private*** (to a face) otherwise. A ***crossing*** is an intersection point between G and a road of the grid (Fig. 1). To simplify the rPFC construction, we bias the geometric tests to ensure that no vertex of G lies on a road or cross-road and that no edge of G contains a cross-road.

In our explanation, we make references to the crossings and think of them as vertices inserted into the graph at places where an edge crosses a road. We wish to stress that we do not actually insert these crossings. We just imagine them there so that

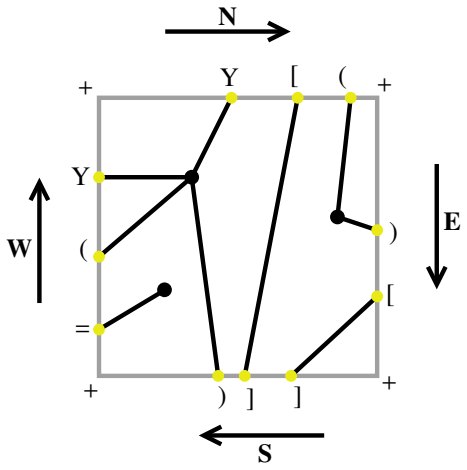

Figure 3. A pixel having 11 crossings (yellow disks), and their corresponding symbols. The word describing the local topology inside this pixel is "=(Y+Y[(+)[+]])+".

we can reference them more easily in our explanations. We use the term ***junction*** when referring to the original vertices of G, so as to distinguish them from the crossings. In the graph produced by these imaginary insertions of crossings, each connected component of the intersection of G with a given pixel comprises exactly zero or one junction. A component with no junction is a ***stabbing***: a segment of an edge of G between two consecutive crossings. A component with a junction comprises a set of edge segments that connect the junction to some of the crossings on the boundary of that pixel.

The rPFC model orders crossings and crossroads around the pixel, associates the symbol '+' with each crossroad, and associates a symbol with each crossing. It distinguishes the following crossing types and uses a different symbol for each type:

- crossing '=' leads to a dead end vertex;
- crossing '[' starts a component that has no junction;
- crossing ']' ends a component that has no junction;
- crossing '(' starts a component connected to a junction;
- crossing 'Y' adds a branch to a junction;
- crossing ')' ends a component connected to a junction.

The topology of a pixel, P, is encoded by the ***word*** formed by the concatenation of the clockwise sequence of the above symbols, starting from the lower left crossing (cf. example in Fig. 3).

The first three '+' symbols are used to separate the crossings into four sub-lists, one per crossroad. These will be referenced using labels: 'W', 'N', 'E', and 'S'. The occurrence of the 4$^{\text{th}}$ '+' symbol indicates the end of the word.

Assuming a valid resolution (no edge of G is contained in a single pixel), the rPFC of G is represented by the sequence of words listed in scanline order of the corresponding pixels. Private (i.e., empty) pixels are identified by the word "++++".

Random Access to the PFC uses the word associated with a chosen pixel P to verify whether P is private, and if not, to calculate the number of connected components and hence of the number of junction vertices contained in P. Different mechanisms may be used to provide direct or indirect access to the face ID of private pixels. Vertices in P are assigned local vertex IDs.

Traversal of the graph G of the PFC uses local identifiers (IDs) of ***darts*** (***sidewalks*** in the "edges are streets" metaphor that are oriented to have the street on their left). These IDs are assigned in clockwise order around P and grouped by road. Simple algorithms exist for mapping (i.e., for finding the local identifier of) a dart to the ***opposite*** dart (the sidewalk on the other side of the street) and for mapping (the local ID with respect to P of) a dart that exits a pixel P to the (the local ID with respect to Q of) the ***next*** dart that (continues the sidewalk past the crossing and) enters the adjacent pixel Q. One may traverse the bounding loop (which is always unique since faces of G have no hole) of sidewalks around a face by following the sequence of next maps.

Other simple algorithms have been provided to compute the next dart of a dart that enters P and the ***previous*** dart for both entering and exiting darts. Finally, given the local ID of a vertex in a pixel, one may easily compute the ID of one of the outgoing or incoming darts, and inversely, given an outgoing or incoming dart, one may compute the corresponding vertex ID.

The inserted crossings vertices are hidden from the application, which operates on the darts and vertices of the original PFC.

All these computations have cost that is proportional to the length (symbol count) of the word associated with the pixel.

When needed, mechanisms for identifying the global ID of the face that is on the right of a sidewalk may be provided, for example by using a hash table and may requiring the traversal of the bounding loop of a face.

3. hPFC

In this section, we propose a formal definition of the hPFC, an algorithm for constructing it, and the two basic tools allowing us to implement the RAT operators which are the navigation through pixels and the crossings identification.

3.1. Definition of hPFC and Terminology

We use the term ***image*** to refer to a regular grid of $n \times m$ pixels. Each pixel of an image stores either a word (in which case we say that it is a ***word pixel***) or the reference to another image (in which case we say that it is a ***refined pixel*** and that the referenced image is a ***refined image***). Note that we use a different terminology than the one proposed in [46, 47] because our approach is not restricted to quadtrees.

An hPFC is a tree of images, having a ***root image*** I_0 that has $n_0 \times m_0$ pixels.

For compactness, we assume that all refined images of an hPFC have the same $n \times m$ pixel size. Note that the tree becomes a quadtree [8] for $n_0 = m_0 = 1$ and for $n = m = 2$.

The hPFC of the PFC of Fig. 2 is shown in Fig. 4, where $n_0 \times m_0$ is set to 4×3 and $n \times m$ is set to 2×2.

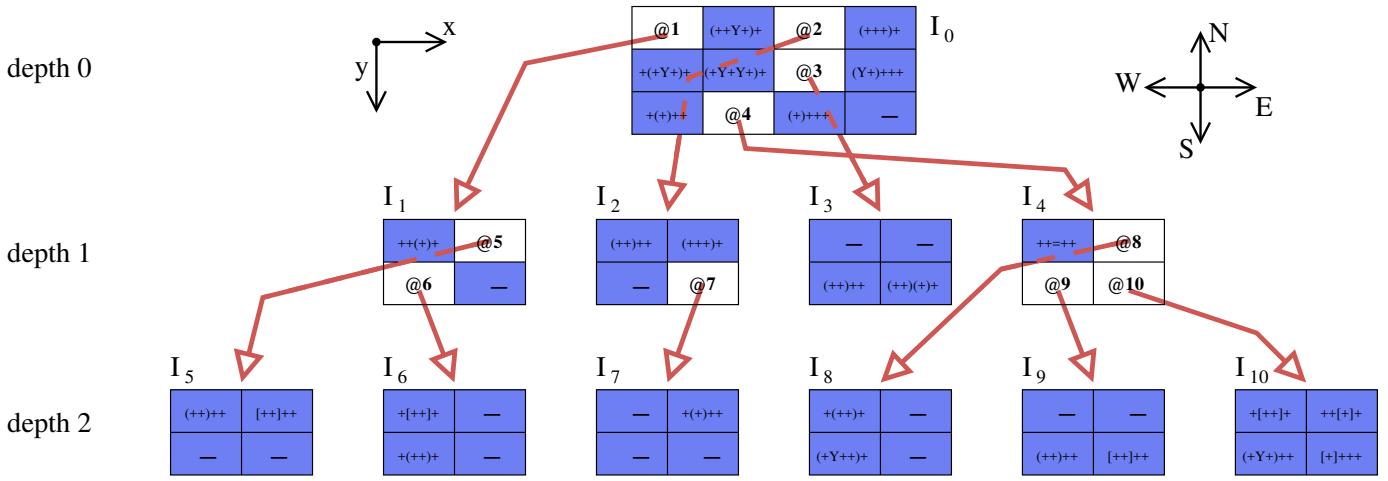

Figure 4. Tree representation of the hPFC of Fig. 2. The root image has 4 refined pixels. The tree has 3 depth layers and contains 42 word pixels (shown in blue) and 10 refined pixels. A "—" symbol is used to mark private pixels (which are contained in a finite face or in the external one).

3.2. hPFC Construction

We briefly explain how we build a hPFC. The pixels are computed recursively. We set an initial grid of $m_0 \times n_0$ pixels. For each pixel (identified by the coordinates of its diagonal), we check that there is no edge of the PFC contained in it. If so, we compute its word. Otherwise, we subdivide it into $m_r \times n_r$ equal pixels and repeat this operation. In practice, we filter the edges of the PFC for each pixel to avoid unnecessary computations of intersections.

Given a pixel P, we compute its word by first finding all its crossings. For this, we intersect the edges of the PFC with the edges of the pixel. For each crossing, we keep the intersected edge of the pixel, the coordinates of the intersection, and a reference to its endpoints contained inside the pixel. Then, we use the intersected edge and the coordinates for sorting the crossings, and we use the references to assign the symbols.

3.3. Navigation Through Pixels

To support constant cost moves of a random walk over the plane and to support the implementation the graph traversal operators (which are discussed in Sect. 2.4), we need to be able to identify the adjacent pixel(s) of any given pixel.

More precisely, given a pixel P, we want to find the first pixel Q adjacent to P in a given direction D (west, north, east or south). If P has several adjacent pixels in direction D (case of pixel a in Fig. 2 for west direction), Q will be the first one along the road between P and Q, considering the reverse orientation than D in pixel P (for pixel a, the first adjacent pixel considering south direction, i.e. pixel b).

A pixel P is denoted by a triplet (r, i, j), with r an image, and (i, j) the position of P in r.

Algorithm 1 allows to find Q, the first pixel adjacent to a given pixel P in the west direction (algorithms for other directions are all similar). This is the classical algorithm for quadtree navigation (see for example [48, 49]), slightly modified to deal with subdivision of any size $n \times m$.

Algorithm 1: Move a pixel to west.

Input: P: a pixel in an hPFC, that does not belong to the first column of I_0.
Output: Q: the first pixel to the west of P.

```
1  Q ← P;
2  while Q is adjacent to the west border do
3      Q←parent(Q);
4  Q.i ←Q.i − 1;
5  while Q is a refined pixel do
6      r ← address in pixel Q;
7      if Q.depth ≤ P.depth then
8          j2 ← y coordinate of pixel at depth Q.depth in the
           path going from I0 to P;
9      else j2 ← 0;
10     Q← (r2, n − 1, j2);
11 return Q;
```

This algorithm has two main parts. First, we do a ***up-the-tree ascension*** while the current pixel Q belongs to the west of its image (lines 2-3). Since P does not belong to the first column of I_0, we are sure that this loop finishes. Then we can move the pixel Q from one position in west direction (line 4).

The last main part (lines 5-10) is a classical ***down-the-tree descent***, while the current pixel Q is a refined pixel. During this descent, we use either the same y coordinate than the pixel at the same depth than Q in the path going from I_0 to P (line 8), or we go to the first pixel in the east road of Q (line 9) if such a pixel does not exist (case when $Q.depth > P.depth$).

In order to be able to find the parent of pixel Q in constant time, without storing explicitly the parent link in the tree, we represent each pixel by a ***stack of triplets***, describing the full path from I_0 to Q. Thanks to this stack, going up in the tree is done directly by a pop operation, and going down by a push of the new triplet.

Let us consider the hPFC given in Fig. 4 (that represents the

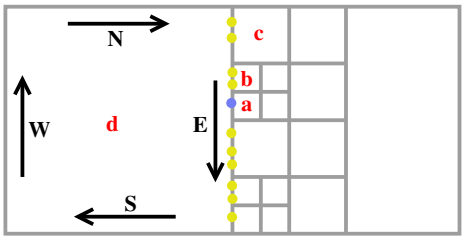

Figure 5. Example of adjacent pixels with different depths: a.*depth* = 2 > d.*depth* = 0.

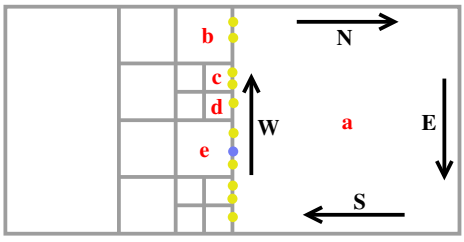

Figure 6. Example of adjacent pixels with different depths: a.*depth* = 0 < b.*depth* = 1.

irregular grid shown in Fig. 2) and show some examples of the move to west algorithm.

- starting from pixel $a = ((I_0, 1, 0))$, we first move to west to $((I_0, 0, 0))$, then we go down twice to reach $b = ((I_0, 0, 0),(I_1,1,0),(I_5,1,0))$ which is the first adjacent pixel to a;
- starting from pixel $c = ((I_0, 1, 2),(I_4,0,1),(I_9,0,1))$, we first go up twice to reach $((I_0, 1, 2))$, then we move to west to $d = ((I_0, 0, 2))$ which is the first (and unique) adjacent pixel to c;
- starting from pixel $e = ((I_0, 1, 2),(I_4,1,1),(I_{10},0,1))$, we first go up once to reach $((I_0, 1, 2),(I_4,1,1))$, then we move to west to $((I_0, 1, 2),(I_4,0,1))$, then we go down to $f = ((I_0, 1, 2),(I_4,1,1),(I_9,1,1))$ which is the first (and unique) adjacent pixel to e. Note in this case that we go directly to pixel (1,1) in image I_9 because we use the y coordinate of the pixel at depth 2 in the path going from I_0 to P (which is P itself).

3.4. Crossings Identification on an hPFC

The second tool required to support the graph traversal operators is the identification of a crossing of a pixel in its adjacent pixel. More precisely, this is the ***crossing problem***: Given the local ID of crossing X of a pixel P and the label of the road D that contains it, we want: (1) to identify the unique word pixel Q that also contains X and (2) to identify the local ID of X in Q.

This problem is discussed in [12], but only for cases where P and Q are word pixels of the same image.

There are three different cases:

1. P.*depth* = Q.*depth*: P and Q have the same number of crossings, and correspondence is straightforward (cf. [12], this is for example the case of pixels e and f in Fig. 2);
2. If P.*depth* > Q.*depth*: We count the number of crossings C in all the pixels adjacent to Q, after P (considering the order given by the orientation of road D in P), plus the number of crossings after X in P. The local ID of X in Q is the crossing number C in the road shared with P (counting the number of crossings is done by navigating through the pixel grids thanks to the operations introduced in the previous subsection).
   An example is given in Fig. 5. Let us suppose X is the unique crossing in the west road of pixel a (in blue in the figure). d is the pixel adjacent to the west of a. The number of crossings in the east road of d before a is 4 (crossings of pixels b and c). The local ID, in d, of X is 4 in the east road of d (thus the 5^{th} crossing of this road because IDs started from 0);
3. If P.*depth* < Q.*depth*: We compute N, the position of crossing X in its road, in reverse orientation. We move Q through the pixels adjacent to P, using the orientation of road D in Q. We stop this loop when the number of crossings, C, in road D is smaller or equal than N. In this case, we have found pixel Q which shares crossing X with P, and the local ID of X in Q is the crossing number N in the road shared with P. During the loop, when C>N, N is decreased by C.
   An example is given in Fig. 6. Here, X is the 5^{th} crossing in the west road of pixel a (in blue in the figure), and N = 6, the position of X in the west road starting from the north. b is the first pixel adjacent to the west of a. We iterate through pixels c, d, while decreasing N from 6 to 4, 2 then 1. Entering in pixel e, we have N = 1 < 3 the number of crossings in the east road of e, thus we stop the loop. The local ID, in e, of X is 1 in the east road of e (thus the 2^{nd} crossing of this road).

3.5. Computational Space and Time Complexity

3.5.1. Time Complexity

The time complexity of Algo. 1 that allows us to navigate between adjacent pixels in the west direction is linear in the depth of the tree (the same for algorithms for other roads). Indeed, this algorithm mainly traverses the tree, possibly a first time bottom-up and a second time top-down.

The complexity of the operator that computes the local crossing number of a given crossing in its adjacent pixel (given in the previous subsection), is linear in the depth of the tree plus the maximal number of crossings in the pixel (indeed, the time complexity of the navigation between crossings inside a same pixel is linear in the number or crossings in the pixel).

With these two operators, it is possible to define the previous, next and opposite operators by using the technique explained in [12] and summarized in Sect. 2.4. The idea is to follow the stabbings of an edge to a junction. The complexity of these operators is thus linear in the maximum number of stabbings in an edge times depth of the tree.

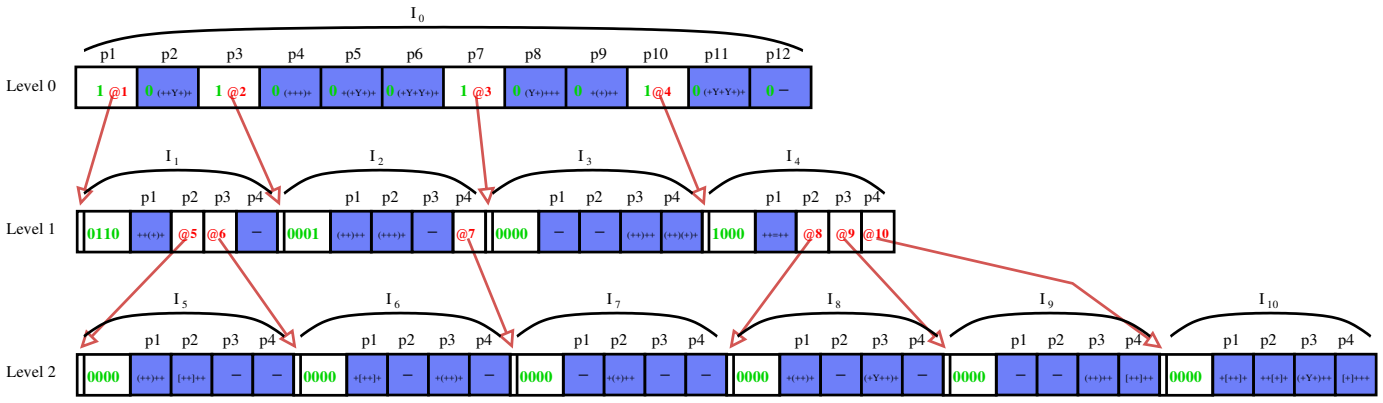

Figure 7. Compact encoding of the hPFC shown in Fig. 4. In level 0, the code of each pixel starts with a bit to differentiate refined and word pixels (in green), and each pixel is encoded by the same number of bits (both refined and word pixels). In other levels, these mask-bits are grouped at the beginning of each image. Different record sizes are used to encode word and refined pixels.

We assume that the depth of the tree, the maximal number of crossings per pixel and the number of stabbings per edge are small comparing to the number of edges, and thus can be considered as constants. This assumption is verified in our experiments and makes sense in general, even though pathological models could be designed to invalidate it. With these assumptions, the complexity of the three operators previous, next and opposite is constant.

3.5.2. *Computational Space Complexity*

The space complexity of hPFC is given by the number of pixels times the (constant) storage size of one pixel. The size of a refined pixel is the size used to store the address of the refined image, while the size of a word pixel is given by the encoding of its word. This size is set to be large enough to contain the largest encoding of a pixel. It may be large if the graph contains a vertex with a huge valence (count of incident edges).

4. Compact Encoding of hPFC

In this section, we propose a ***compact encoding*** of hPFC. The key point of our solution is to use a small amount of memory space, while allowing to navigate through an hPFC by decoding locally the traversed pixel (we do not need to decode the full model, contrary to several ***compressed encoding***).

4.1. *Main Principle and Notation*

rPFCs are encoded level by level of the tree. The first level, numbered 0, contains only I_0, the root of the tree, while other levels, between 1 and d (for depth) can contain several rPFCs. For example, the hPFC of Fig. 4 has 3 levels.

We use our own made memory manager that can allocate an arbitrary number of contiguous bits in memory, segment it into contiguous k-bit records, and support writing and reading k-bit integers in and out of these records using the integer index of a record.

All images at depth l are stored in a contiguous ***pool***.

Let s^l denote the size (measured in bits) of the memory pool allocated to store all of the images at depth l.

Let n_r^l denote the number of refined pixels at depth l, and let n_w^l denote the number of word pixels at depth l.

Let k_w^l denote the number of bits required to store words at level l, i.e. the number of bits to encode the longest word in this level, and let $k_@^l$ denote the number of bits required to store the rPFC addresses for refined pixels at level l.

4.2. *Data Structure for the Root Image I_0*

The information associated with each pixel of the root image is stored using a fixed representation that allocates the same number of bits to each pixel. For each pixel, P, the first bit indicates whether P is a word pixel or a refined pixel. The remaining $max(k_@^0, k_w^0)$ bits encode either the address of the rPFC in level 1 (for refined pixels) or the word (for word pixels) (cf. example in Fig. 7).

Hence, we have a direct access to any pixel (i, j) and can compute its memory address in the pool from $k_@^0$, k_w^0 and the dimensions n_0 and m_0 of I_0.

The main advantage of this encoding is the direct access to any pixel. Its main drawback is that some more memory space is wasted due to the use of the same number of bits for word and refined pixels. But this loss is negligible because the number of pixels in I_0 is usually very small comparing to the number of pixels in refined images (cf. Sect. 5).

4.3. *Data Structure for Refined Images*

The representation of the pool of a refined (non zero) level d stores the words of all its images in a contiguous (compacted) array.

The encoding of each refined image I starts by a bit mask (one bit per pixel of I) that identifies the type of each pixel (word or refined). Following this mask, each pixel P of I is encoded: the word of P for word pixel and the address of the refined image for refined pixels. Contrary to I_0, we use here an irregular encoding: word pixels are encoded by k_w^d bits while refined pixels use $k_@^d$ (cf. example in Fig. 7).

The key advantage of this encoding is to use less memory than the representation used for the root image (no memory space is lost to align the encoding of the two types of pixels).

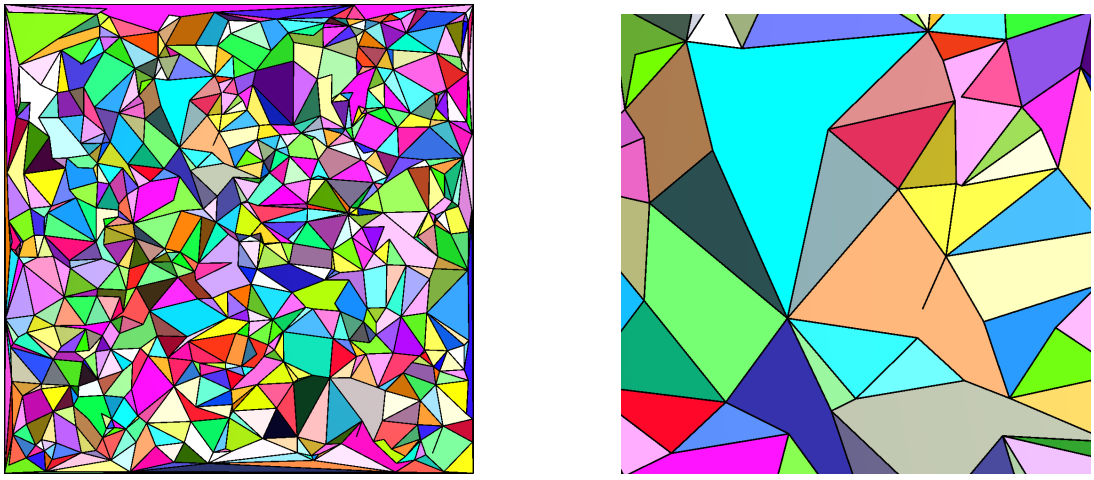

Figure 8. One of the Delaunay triangulations used with 500 vertices, and a detail of it.

Its main drawback is the cost of computing the location of one pixel: we need to decode first the bit mask for all pixels before P in order to compute the address of P in the current image. But the overhead is small because the size of refined images is usually very small (often 2×2 in our experiments, cf. Sect. 5).

4.4. Computing Sizes and In-Pool-Locations of Arrays

The numbers n_r^l, of refined pixels per level, n_w^l, of word pixels per level, and k_w^l, of bits to store the maximal word per level can all be computed for each level of the tree, during the building or the loading of an hPFC.

The number of bits needed to store rPFC addresses per level, $k_@^l$, and the size of each level, s^l, can be computed recursively (top-down) by using n_r^l, n_w^l and k_w^l.

First, there is no refined pixel at the last level of the tree, and thus $n_r^d = 0$ and $k_@^d = 0$. We can thus directly compute $s^d = n_w^d \times k_w^d$.

Then, each number of bits $k_@^l$ can be computed directly thanks to s^{l+1}. For each rPFC address at level l, we need to be able to address any position in the memory pool at level $l+1$. Thus $k_@^l = \lceil log_2(s^{l+1}) \rceil$. We can compute $s^l = n_w^l \times k_w^l + n_@^l \times k_@^l$, and thus compute $k_@^l$ and s^l for each level.

4.5. Illustration

Figure 7 shows an illustration of our compact encoding for the hPFC previously given in Fig. 4.

The first level has 12 pixels since the size of I_0 is 4×3; 4 of them being refined pixels having their encoding starting by 1.

The size of all refined images is 2×2, the encoding of each image starts by 4 bits that describe the type of all of its pixels.

The length of the maximal word in Level 2 is 7 and thus $k_w^2 = 16$ (1 bit per '+', 4 bits per other symbol). The number of bits used to store Level 2 is thus $s^2 = 6 \times (4 \times 16 + 4) = 408$ bits (there are 6 images, each one with 4 pixels, each pixel uses 16 bits plus the 4 bits mask).

The length of the maximal word in Level 1 is 8 and thus $k_w^1 = 20$. $k_@^1 = 9$ in order to address any position in Level 2. The number of bits used to store Level 1 is thus $s^1 = 4 \times 4 + 6 \times 9 + 10 \times 20 = 270$ bits (there are 4 images, 6 refined pixels and 20 word pixels).

Similarly for Level 0, we have $k_w^0 = 20$, $k_@^0 = 9$, $s^0 = 12 \times (20 + 1) = 252$ bits (there are 12 pixels, each one encoded with 20 bits whatever their type, plus one bit for the type of the pixel).

# vertices	0.5K	1K	2K	4K
# edges	1,205	2,405	4,805	9,605
# faces	703	1,403	2,804	5,605

Table 1. Average number of edges and faces in each batch of Delaunay triangulations.

5. Experiments

In [12], four different architectures were introduced to represent the words in a rPFC. The first version (V1) explicitly encodes the words using a fixed prefix code and puts them in a matrix with constant size entries. The other three versions concatenate the encoding of words in each row and store them in an array using different optimizations for the private pixels.

In this section we compare the hPFC against the first version of the rPFC using synthetic and real data sets. For the sake of simplicity, we set $m_0 = n_0$ and $m_r = n_r$.

5.1. Delaunay Triangulations

Our first data set consists of series of five random Delaunay triangulations with 500, 1,000, 2,000 and 4,000 vertices, from which we randomly removed 20% of the edges to increase irregularity. One of these triangulations is depicted in Fig. 8. More information about their number of cells is described in Table 1.

In our first experiment, we report results for different values for m_0 and m_r. We have considered only the five Delaunay triangulations with 4,000 vertices and we have computed hPFCs with m_0 in {1,2,4,8,16,32,64,125,256} and m_r in {2,4,8,16}. Figure 9 shows the results. Memory space grows with both m_r and

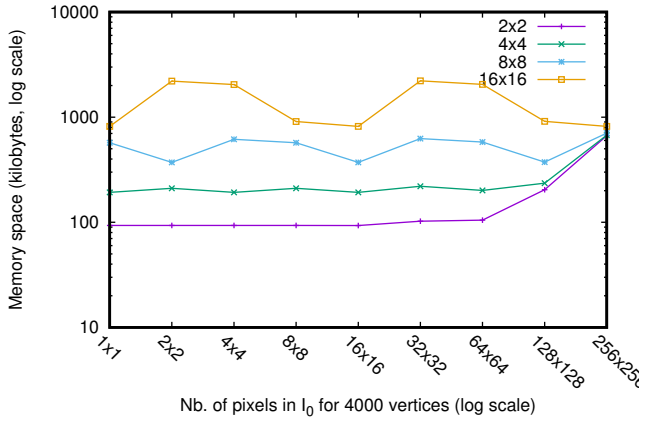

Figure 9. Memory comparison between hPFCs with different parameters.

m_0. Indeed, setting a large m_0 or m_r produces unnecessary small pixels, many of which are empty. For these inputs, we conclude that m_r must be 2 and m_0 must not be greater than 64.

Note that the oscillations in this graphic are produced by the combined effect of m_0 and m_r on the size of the largest pixels. To illustrate this, consider the curve for $m_r = 16$. The rPFCs with $m_0 = 1$ and $m_1 = 16$ are the same and thus occupy the same memory space. The memory space grows because when we refine the initial pixel grid, we multiply the number of pixels. On the other side, by refining further we end up avoiding later subdivision and thus the memory space decreases.

In our second experiment we process the twenty Delaunay triangulations and compare the memory space used by the rPFC and the hPFC. The grid size of the rPFCs is computed for each triangulation so that no edge fits inside a pixel. For the hPFCs, we fix (m_0,m_r)=(64,2). The results are given in Fig. 10. For the 20 Delaunay triangulations, on average, hPFC is **1352**, **1078**, **4910** and **3620** times more compact than the rPFC (for 0.5K, 1K, 2K and 4K vertices, resp.). The reason for this is that a single short edge in a mesh forces the rPFC to use a large number of pixels. The rPFCs of the triangulations with 4,000 vertices contain in average more than 39.9 million pixels, 99.4% of them being private, while the hPFCs have only 10,292 pixels, and only 2.7% of them are private. Thus, even with the overhead for storing the hierarchical structure of the hPFC, it is 3,000 times more compact than the rPFC. In this experiment, the depth of the tree is between 5 and 8, the maximum length of words is between 24 and 30, and the average number of stabbings per edge is between 1.2 and 1.3.

To complement the previous experiment, we now compare the time complexity of the hPFC and the rPFC. For each triangulation, we traverse all its half-edges with a depth-breadth search. Figure 11 shows the results. It is clear that the hPFC is much faster than the rPFC. For the triangulations with 4,000 vertices, the rPFC traverses in average 151.03 pixels per edge of the PFC, while the hPFC traverses only 8.33 (word and refined) pixels.

We conclude that the hPFC is more efficient both in memory space and time than the rPFC for these triangulations. Note,

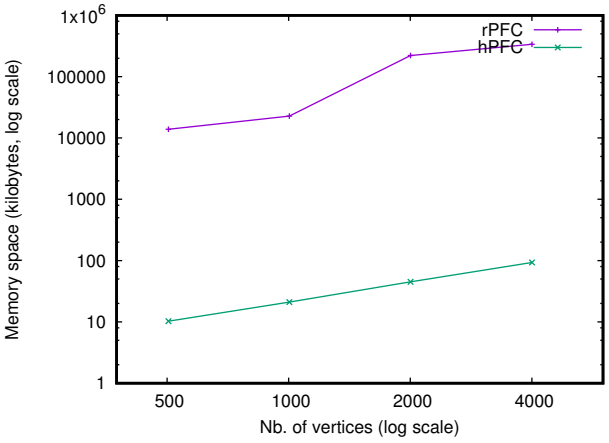

Figure 10. Memory comparison between the rPFC and the hPFC for the synthetic data set.

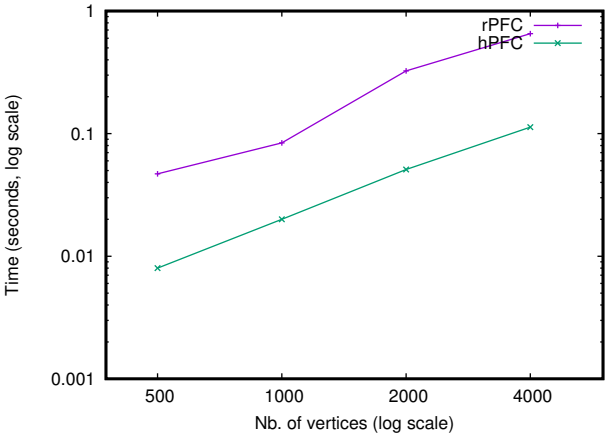

Figure 11. Time comparison between the rPFC and the hPFC for the synthetic data set.

however, that we were able to represent all these meshes with the rPFC, which is not the case for the following experiments.

5.2. GIS models

Our real data set consists of spatial databases of eight countries (Australia, Brazil, Canada, China, Germany, France, United Kingdom and Russia) available at `http://www.gadm.org/`. They are in shapefile format, a popular geospatial vector data format for geographic information system (GIS) softwares. The PFC of Australia is shown in Fig. 12.

Note that, unlike in the previous data set, these PFCs contain very short edges and thus the rPFC needs a huge number of pixels. Therefore, we have created five simplified versions of these meshes for comparing the hPFC with the rPFC. The simplified meshes are made so that they can fit in rPFCs with m_0 in {1,024, 2,048, 4,096, 8,192, 16,384}. Table 2 describes the average number of cells in each batch of simplified meshes. The original meshes, which contain on average 1,407,761 vertices, could only be represented with the hPFC. In this experiment, for the hPFCs that represent the original meshes, the depth of the tree is between 10 and 16, the maximum length of words is between 12 and 24, and the average number of stabbings per edge is between 0.3 and 0.5.

Figure 13 shows the average memory space and time for the simplified and the exact meshes. Regarding the memory space cost, the hPFC still outperforms the rPFC. The hPFC is **9**, **17**, **34**, **75** and **188** times more compact than the rPFC (for the simplified meshes). For the exact (non simplified) GIS meshes, the rPFC would require 1,079 millions of pixels in average, while the hPFC only has 1.5 million (41.4% of private).

Regarding the time complexity, the roles are inverted. For the least simplified meshes, the rPFC traverses only 3.79 pixels per edge and the hPFC, 5.34. The GIS models have much shorter edges than the Delaunay triangulations, and thus, the overhead for navigating through the levels of the hPFC ends up making it 3.5 times slower.

Note that we could not compute the rPFCs for the original meshes because of the excessive number of pixels necessary.

We conclude that the hPFC is more compact than the rPFC, and it is necessary for representing real data sets with exact topology. Also, we conjecture that its time complexity depends on the ratio between the minimum and the mean length of the edges of the PFC.

6. Conclusion and Future Works

The hPFC hierarchical representation of Planar Face Complexes that is proposed here uses a fixed-resolution pixel-grid, quantizes the geometry of the vertices and edges to pixel-resolution, and encodes graph connectivity by small number of bits per pixel. It improves the previously proposed rPFC, which requires that no edge be contained in a single pixel, by constructing a tree of rPFCs. This hierarchical, quadtree-like model, makes it possible to represent exactly any PFC topology and reduces storage cost significantly.

In future work, we plan to propose different compact representations for hPFC. We will explore different techniques that combine the principle of hPFC or of rPFC with more compact encodings or compression of the words, of the masks, and of the addresses. Indeed, there are numerous options for such improvements, and results of comparisons depend heavily on the nature of the PFC.

We would like to study if rPFC and hPFC can be used to represent non planar graphs, for example to describe a street network having bridges and tunnels. In some configurations, it maybe possible to represent a non-planar graph by a planar graph by inserting vertices, but support of traversal operators on such extensions is more challenging.

Lastly, we are working on the extension of rPFC and hPFC in 3D.

Acknowledgements

This project received funding from the European Unions Horizon 2020 Research and Innovation program under the Marie Sklodowska-Curie (grant 659526).

References

[1] Schmidt, A, Lafarge, F, Brenner, C, Rottensteiner, F, Heipke, C. Forest point processes for the automatic extraction of networks in raster data. ISPRS Journal of Photogrammetry and Remote Sensing 2017;126:38 – 55.

[2] Castanié, L, Lévy, B, Bosquet, F. VolumeExplorer: Roaming large volumes to couple visualization and data processing for oil and gas exploration. In: IEEE Visualization conference proceedings. 2005,.

[3] Dalstein, B, Ronfard, R, van de Panne, M. Vector graphics complexes. ACM Trans Graph 2014;33(4):133:1–133:12.

[4] Kwok, TH, Chen, Y, Wang, CC. Geometric analysis and computation using layered depth-normal images for three-dimensional microfabrication. In: Baldacchini, T, editor. Three-Dimensional Microfabrication Using Two-photon Polymerization. Micro and Nano Technologies; Oxford: William Andrew Publishing. ISBN 978-0-323-35321-2; 2016, p. 119–147.

[5] Baumgart, BG. Winged-edge polyhedron representation. Tech. Rep.; Stanford; 1972.

[6] Mäntylä, M. An Introduction to Solid Modeling. Computer Science Press; 1988.

[7] Alumbaugh, TJ, Jiao, X. Compact array-based mesh data structures. In: Proc. of 14th Int. Meshing Roundtable (IMR). 2005, p. 485–503.

[8] Samet, H. Connected component labeling using quadtrees. J ACM 1981;28(3):487–501.

[9] Benouamer, MO, Michelucci, D. Bridging the gap between CSG and Brep via a triple ray representation. In: Proceedings of the Fourth ACM Symposium on Solid Modeling and Applications. SMA '97; New York, NY, USA: ACM. ISBN 0-89791-946-7; 1997, p. 68–79.

[10] Szymczak, A, Rossignac, J, King, D. Piecewise regular meshes: Construction and compression. Graphical Models 2002;64(3):183–198.

[11] Ju, T, Losasso, F, Schaefer, S, Warren, J. Dual contouring of Hermite data. ACM Trans Graph 2002;21(3):339–346.

[12] Damiand, G, Rossignac, J. Rasterized planar face complex. Computer-Aided Design (CAD) 2017;90:146–156.

[13] Lienhardt, P. N-Dimensional generalized combinatorial maps and cellular quasi-manifolds. Inte J of Computational Geometry and Applications 1994;4(3):275–324.

[14] Dobkin, DP, Laszlo, MJ. Primitives for the manipulation of three-dimensional subdivisions. In: Proceedings of the Third Annual Symposium on Computational Geometry. SCG '87; New York, NY, USA: ACM. ISBN 0-89791-231-4; 1987, p. 86–99.

[15] Castelli Aleardi, L, Devillers, O, Schaeffer, G. Succinct representations of planar maps. Theor Comput Sci 2008;408(2-3):174–187.

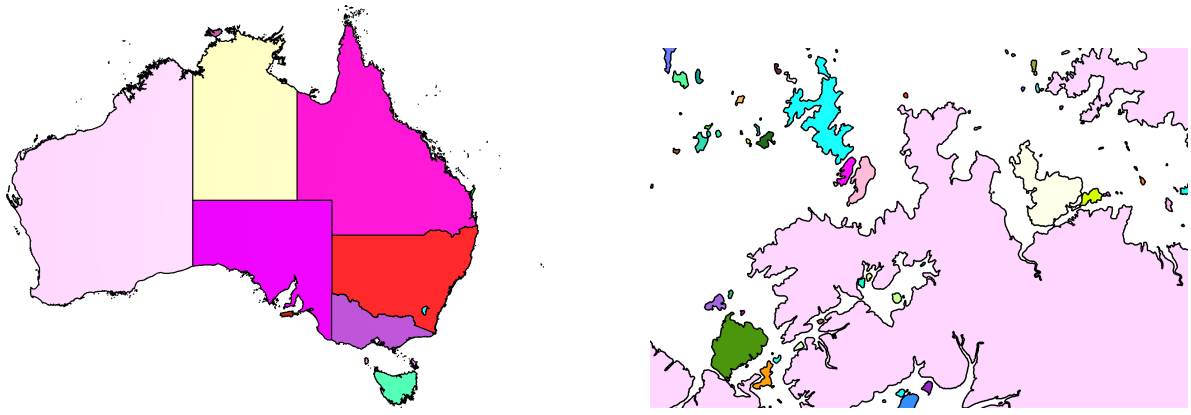

Figure 12. One of the GIS mesh used (Australia), and a detail of it.

m_0	1,024	2,048	4,096	8,192	16,384	
# vertices	84,083	150,143	248,144	380,719	562,380	1,407,761
# edges	90,707	160,008	262,055	397,685	580,635	1,429,188
# faces	7,743	11,952	17,522	22,806	26,858	31,421

Table 2. Average number of vertices, edges and faces in each batch of simplified GIS models and the exact GIS models.

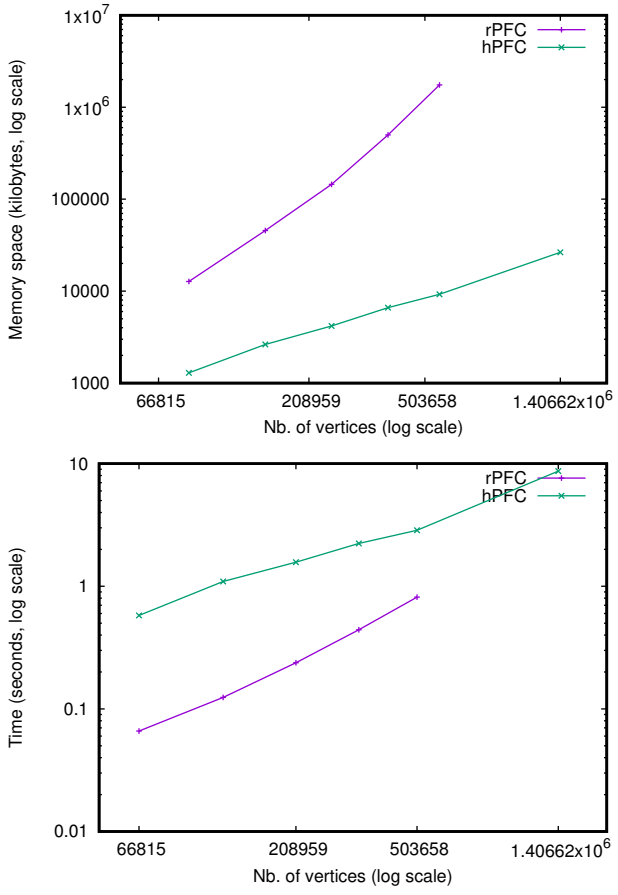

Figure 13. Memory and time comparison between the rPFC and the hPFC for the GIS data set.

[16] Kallmann, M, Thalmann, D. Star-vertices: a compact representation for planar meshes with adjacency information. Journal of Graphics Tools 2002;6:7–18.

[17] Kettner, L. Using generic programming for designing a data structure for polyhedral surfaces. Comp Geometry 1999;13:65–90.

[18] Snoeyink, J, Speckmann, B. Tripod: a minimalist data structure for embedded triangulations. In: Workshop on Comput. Graph Theory and Combinatorics. 1999,.

[19] Sieger, D, Botsch, M. Design, implementation, and evaluation of the surface_mesh data structure. In: Quadros, WR, editor. Proceedings of the 20th International Meshing Roundtable. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-24734-7; 2012, p. 533–550.

[20] Botsch, M, Kobbelt, L, Pauly, M, Alliez, P, Lévy, B. Polygon Mesh Processing. AK Peters; 2010.

[21] Damiand, G, Lienhardt, P. Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing. A K Peters/CRC Press; 2014.

[22] Nehab, D, Hoppe, H. Random-access rendering of general vector graphics. ACM Trans Graph 2008;27(5):135:1–135:10.

[23] Shade, J, Gortler, S, He, Lw, Szeliski, R. Layered depth images. In: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '98; New York, NY, USA: ACM. ISBN 0-89791-999-8; 1998, p. 231–242.

[24] Blelloch, GE, Farzan, A. Succinct representations of separable graphs. In: CPM. 2010, p. 138–150.

[25] Blandford, DK, Blelloch, GE, Cardoze, DE, Kadow, C. Compact representations of simplicial meshes in two and three dimensions. Int Journal on Comp Geometry and Applications 2005;15(1):3–24.

[26] Campagna, S, Kobbelt, L, Seidel, HP. Direct edges - a scalable representation for triangle meshes. Journal of Graphics tools 1999;3(4):1–12.

[27] Castelli Aleardi, L, Devillers, O, Mebarki, A. Catalog-based representation of 2D triangulations. Int J Comput Geometry Appl 2011;21(4):393–402.

[28] Castelli Aleardi, L, Devillers, O. Explicit array-based compact data structures for triangulations. In: Proc. 22th Ann. Internat. Sympos. Algorithms Comput.; vol. 7074 of *LNCS*. 2011, p. 312–322.

[29] Gurung, T, Laney, DE, Lindstrom, P, Rossignac, J. SQuad: Compact representation for triangle meshes. Comput Graph Forum 2011;30(2):355–364.

[30] Gurung, T, Luffel, M, Lindstrom, P, Rossignac, J. LR: Compact connectivity representation for triangle meshes. ACM Transactions on Graphics (TOG) 2011;30(4):67:1–67:8.

[31] Yamanaka, K, Nakano, SI. A compact encoding of plane triangulations with efficient query supports. Inf Process Lett 2010;110(18-19):803–809.
[32] Luffel, M, Gurung, T, Lindstrom, P, Rossignac, J. Grouper: A compact, streamable triangle mesh data structure. Visualization and Computer Graphics, IEEE Transactions on 2014;20(1):84–98.
[33] Bischoff, U, Rossignac, J. TetStreamer: compressed back-to-front transmission of Delaunay tetrahedra meshes. In: Data Compression Conference. 2005, p. 93–102.
[34] Gurung, T, Luffel, M, Lindstrom, P, Rossignac, J. Zipper: A compact connectivity data structure for triangle meshes. Computer-Aided Design 2013;45(2):262–269.
[35] Ayala, D, Brunet, P, Juan, R, Navazo, I. Object representation by means of nonminimal division quadtrees and octrees. ACM Trans Graph 1985;4(1):41–59.
[36] Hoppe, H. Progressive meshes. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '96; New York, NY, USA: ACM. ISBN 0-89791-746-4; 1996, p. 99–108.
[37] De Floriani, L, Marzano, P, Puppo, E. Multiresolution models for topographic surface description. The Visual Computer 1996;12(7):317–345.
[38] Gandoin, PM, Devillers, O. Progressive lossless compression of arbitrary simplicial complexes. ACM Trans Graph 2002;21(3):372–379.
[39] Lefebvre, S, Hoppe, H. Compressed random-access trees for spatially coherent data. In: Proceedings of the 18th Eurographics Conference on Rendering Techniques. EGSR'07; Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. ISBN 978-3-905673-52-4; 2007, p. 339–349.
[40] Finkel, RA, Bentley, JL. Quad trees a data structure for retrieval on composite keys. Acta Inf 1974;4(1):1–9.
[41] Orenstein, JA. Multidimensional tries used for associative searching. Information Processing Letters 1982;14(4):150–157.
[42] Shneier, M. Two hierarchical linear feature representations: Edge pyramids and edge quadtrees. Computer Graphics and Image Processing 1981;17(3):211–224.
[43] Tamminen, M, Sulonen, R. The excell method for efficient geometric access to data. In: Proceedings of the 19th Design Automation Conference. DAC '82; Piscataway, NJ, USA: IEEE Press. ISBN 0-89791-020-6; 1982, p. 345–351.
[44] Webber, RE, Samet, H. On encoding boundaries with quadtrees. IEEE Transactions on Pattern Analysis & Machine Intelligence 1984;6:365–369.
[45] Hunter, GM, Steiglitz, K. Operations on images using quad trees. IEEE Trans Pattern Anal Mach Intell 1979;1(2):145–153.
[46] Samet, H, Webber, RE. Storing a collection of polygons using quadtrees. ACM Trans Graph 1985;4(3):182–222.
[47] Samet, H. Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2005. ISBN 0123694469.
[48] Gargantini, I. An effective way to represent quadtrees. Commun ACM 1982;25(12):905–910.
[49] Samet, H. An overview of quadtrees, octrees, and related hierarchical data structures. In: Earnshaw, RA, editor. Theoretical Foundations of Computer Graphics and CAD. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-83539-1; 1988, p. 51–68.