Rasterized Planar Face Complex

Guillaume Damiand, Jarek Rossignac

▶ **To cite this version:**

Guillaume Damiand, Jarek Rossignac. Rasterized Planar Face Complex. Computer-Aided Design, 2017, 90, pp.146 - 156. 10.1016/j.cad.2017.05.010 . hal-01578921

HAL Id: hal-01578921
https://hal.science/hal-01578921v1

Submitted on 30 Aug 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rasterized Planar Face Complex

Guillaume Damiand*

Univ Lyon, CNRS, LIRIS, UMR5205, F-69622 France

Jarek Rossignac

School of Interactive Computing, Georgia Institute of Technology, Atlanta, USA

Abstract

We consider a Planar Face Complex (PFC). It is defined by the immersion of a planar and connected graph G, which comprises a set of vertices joined by curved edges. G decomposes the plane into faces that need not be manifold or open-regularized and may be bounded by a single loop edge. The PFC may, for example, be used to represent the complex street network of a city, the decomposition of a continent into countries, or the inhomogeneous structure made of a large set of regions of different materials possibly with internal cracks. The rasterized Planar Face Complex (rPFC) proposed here provides a compact representation of an approximation of a PFC, where the precise location of each vertex is quantized to the pixel that contains it and where the precise geometry of each curved edge is approximated by the ordered list (with possible repetitions) of the pixels traversed by (a chosen polygonal approximation of) the edge. We claim three key contributions: (1) The geometric error between a PFC and its rPFC is bounded by the pixel half-diagonal. (2) In spite of such a drastic discretization of the geometry, the rPFC captures the exact topology of the original PFC (provided that no street lies entirely inside a single pixel) and supports standard graph traversal operators that permit to walk the loop of sidewalks along the streets that bound a face, to cross a street to the opposite sidewalk, or to cross streets in order while walking around their common junction. (3) The local connectivity and order information needed to provide the above functionality is stored at each pixel using only about 4 bits per crossing. We discuss the details of this representation, our implementation of its exact construction, four possible embodiments that offer different space/time efficiency compromises, experimental results, relations between rPFC and prior solutions.

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

1. Introduction

We propose a rasterized representation for planar face graphs. It may be used to model planar multi-material structures [1], street networks [2], geological models [3], or to describe accurate rasterization of planar maps of overlapping SVG elements [4]. Our representation, called rasterized Planar Face Complex (rPFC), stores only a few symbols for each pixel intersected by the edges. It supports exact topological graph traversal, even when a pixel overlap with several regions and contains several vertices.

Many previously proposed representations of planar face graphs either store the connectivity using integer references (a solution that has a high storage cost) or re-sample the graph at intersections with grid lines and discard the topology information about what happens inside a pixel. Our solution unifies these approaches and provides a compact, per pixel representation. It provides a spatial decomposition of the connectivity of the graph into local (per pixel) descriptors that can be stored using a constant number of bits per pixel and streamed as needed.

Our solution stores the topology of the original complex (*assuming that no edge lies entirely inside a single pixel*) and is capable of representing sub-pixel connectivity. Hence, it should be distinguished from pure image-based approaches that only store a face color or scalar per pixel and from approaches that re-sample the graph at intersections with grid lines and that discard the original connectivity information. In fact, rPFC may be easily configured to extend these approaches by providing them with a space efficient representation of the exact topology in each pixel.

The paper is organized as follows. In Sect. 2, we define the Planar Face Complex and the operators that we advocate for traversing its graph in a consistent manner. In Sect. 3, we outline our approach, explaining our exact rasterization process, the computation of the local (per pixel) connectivity, and its compact encoding. In Sect. 4, we explain how we use it to support traversal of the original graph and ordered access to vertex and face attributes. In Sect. 5, we discuss four embodiment options, and some im-

*Corresponding author

Email addresses: guillaume.damiand@liris.cnrs.fr (Guillaume Damiand), jarek@cc.gatech.edu (Jarek Rossignac)

Author version of paper "Rasterized Planar Face Complex; Damiand G., Rossignac J.; Computer-Aided Design (CAD), Volume 90, pages 146-156, September 2017". Thanks to Elsevier.

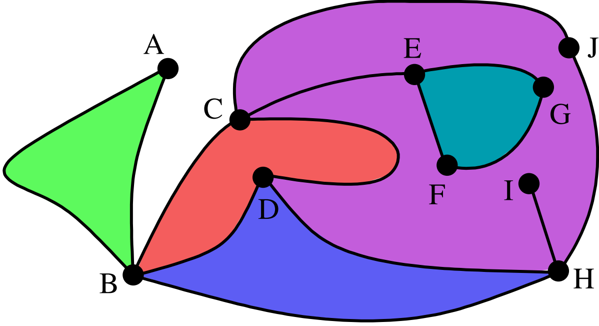


Figure 1: A Planar Face Complex (PFC) with 14 curved edges, 5 faces, a non-manifold vertex (B) on the boundary of the external (infinite) face, a crack (from C to E), and a dead-end vertex (I).

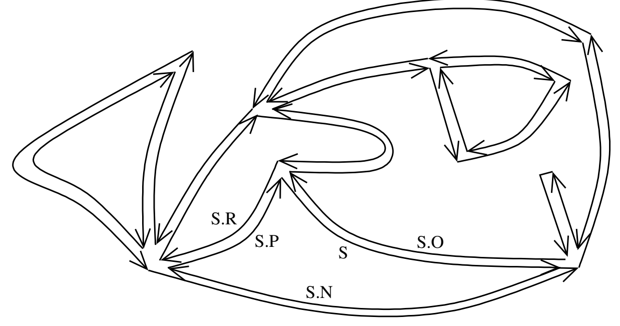


Figure 2: Primary sidewalks operators in a PFC map a sidewalk S to the opposite one, S.O on the other side of the curved street, to the next, S.N, along the clockwise sidewalk loop of face S.F, and to the previous one, S.P along that loop. The derived rotate operator, S.R, maps S to the next sidewalks that emanates from S.V and is obtained by rotating clockwise around S.V. Note that S.R may be derived as the composition S.P.O of the previous and opposite maps.

plementation details. In Sect. 6, we present experimental results. In Sect. 7, we compare our approach to relevant prior art. In Sect. 8, we summarize our contributions and outline future challenges.

2. Background and terminology

2.1. Planar Face Complex (PFC)

Consider a connected, curved, planar one-complex defined by a set of ***vertices*** in the plane and by a set of possibly curved and relatively open ***edges*** that connect them. We assume a proper embedding: (1) different vertices do not coincide, (2) each edge is manifold (free from self-intersection), and (3) the relative interior of each edge is disjoint from other edges and vertices. This set of edges and vertices is the embedding of a ***planar graph***, G.

Let U be the set theoretic union of the point sets of the vertices and edges of G. Consider the complement, !U, of U. Each maximally connected component of !U is called a ***face***. Note that a face is topologically open (i.e., it does not contain its boundary). Because we assume that G is connected, a face cannot have holes (i.e., its boundary must be connected). However, the boundary, B, of a face, F, may contain ***cracks*** (edges bounded by F on both sides) and ***pinch points*** (non-manifold vertices). G may have ***multi-edges*** (more than one edge joining any given pair of vertices) and ***loops*** (edges that start and end at the same vertex). All faces are bounded (i.e., fit in a disk of finite radius) except for one, which we call the ***external face***.

We distinguish three types of vertices: ***dead ends*** (which bound a single edge of G), ***turns*** (which bound two edges) and ***junctions*** (which bound 3 or more edges).

We use the term ***Planar Face Complex (PFC)*** when referring to the faces, edges, and vertices of a decomposition of the plane by such a connected and curved, planar one-complex. An example is shown in Fig. 1.

Consider a regular grid of ***pixels*** (open faces), separated by ***roads*** (their relatively open edges), and ***crossroads*** (their vertices). A pixel is ***stabbed*** when its intersection with G is not empty.

Our goal is to provide a compact representation of a rasterized (i.e. approximate) version of a PFC, where the vertex and edge geometry is approximated (known only up to pixel resolution), but where the graph connectivity is represented exactly and yet broken down into compactly encoded local components, stored at each stabbed pixel.

2.2. Sidewalks and operators

In this subsection, we describe a generic representation and associated traversal operators for PFC. We assume that the input PFC is formulated using such a representation (or an equivalent one) and we want to construct a new rasterized representation of the PFC that supports the original traversal operators exactly, so that the application that uses these operators does not need to be changed.

We use the term ***street*** for each edge of G. With each street, we associate two ***sidewalks***, one on each side. We orient each sidewalk, S, so that its street, ***S.E***, lies on the right of S (with respect to this orientation). We chose this non-standard terminology to distinguish the elements of G from those of the proposed representation. A sidewalk is ***stabbing*** a pixel, P, when its street, E, traverses P (splits it into two or more faces).

We use ***S.V*** to denote the vertex at which sidewalk S starts and ***S.F*** to denote the face abutting to S. Note that the circularly ordered list of the sidewalks of a face forms an oriented ***loop*** that goes clockwise around that face.

We use three ***primary sidewalk operators*** that map a sidewalk S to a neighboring sidewalk (see Fig. 2):

- ***S.N*** = ***next*** sidewalk along the loop of S.F
- ***S.P*** = ***previous*** sidewalk along the loop of S.F
- ***S.O*** = ***opposite*** sidewalk (on the other side of S.E)

The collection of these three operators defines a ***Combinatorial Map*** [5, 6] (the tuple of the set of sidewalks and of the set of these operators is an algebra). Note that S.P is the inverse of S.N (i.e., S.N.P=S) and that S.O is an involution (i.e., S.O.O=S).

Various data structures have been proposed to support these operations. We discuss some in Sect. 7. Note that in 2D, a combinatorial map is equivalent to several other data structures such as Half-Edge, Corner Table, or Doubly Connected Edge List (DCEL) (see [6]).

As done in several modern representations, we associate sidewalks with consecutive integer IDs. To simplify presentation, we will assume that S refers either to the abstract concept of a sidewalk or to the associated integer ID, depending on the context. Two arrays are often used to store the maps for two sidewalk operators: N[S] stores S.N and P[S] stores S.P. Trading processing time for storage, one may avoid storing P[S] explicitly and compute S.P, each time it is needed, by traversing the loop around S.F and returning the last sidewalk reached before returning to S. Assuming that the IDs of S and of S.O are consecutive makes it unnecessary to store the "opposite" map explicitly in an array O[] (S.O may be computed from S as S+1 if S is even and S-1 otherwise).

Our goal is to provide a more compact, yet localized (per pixel), representation of the O[], P[] and N[] maps of a rasterized version of a PFC, which we call the rasterized Planar Face Complex (and abbreviate rPFC).

3. Rasterized Planar Face Complex (rPFC)

In this section, we describe the proposed approach for computing the rPFC of a PCF. Our solution is based on a numerically robust rasterization process that performs the following steps: (1) identify the set of pixels that contains (the edges and vertices of) graph G, (2) split stabbing (entering and leaving the same pixel) street by inserting split vertices, (3) order the entering sidewalks around each pixel, and, (4) for each pixel, store the "next" map between entering and exiting sidewalks in a compact form, as a sequence of symbols (which we call a ***word***) taken from a small codebook of 7 or fewer symbols.

Compact data structures for storing these words while providing fast access to neighboring pixels are discussed in Sect. 5.

3.1. Rasterization and refinement of G

During the rasterization step, for each vertex and for each edge of G, we identify the pixels that contain it. Furthermore, for each edge E of G, and for each pixel P that E exits or enters, we identify the roads through which it does so.

To avoid dealing with singular configurations where a vertex of G lies exactly on a road or crossroad and configurations where a street of G passes exactly through a crossroad or overlaps with a road, we use a special version

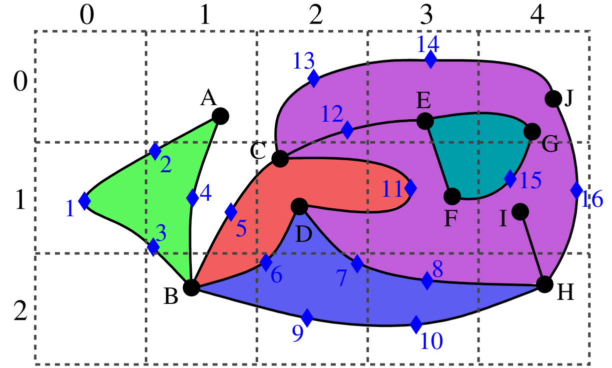


Figure 3: Refined graph R with original vertices of G shown as disks and inserted split vertices shown as diamonds. Note that pixel (1,1) is split twice by the same street, (A,B), of G, which in R is replaced by 4 streets: (A,2), (2,1), (1,3), and (3,B).

of Simulation of Simplicity [7] and bias our geometric tests so as to produce results consistent with a virtual perturbation of G that avoids such singularities (for example, when a street of G passes through a crossroad, we assume that it passes below it or on the right of it, depending on the street orientation. Similarly, when a vertex of G lies on a road or crossroad, we assume that it lies above, to the right, or both, depending on the configuration).

In our implementation, we assume that each street is represented by an approximating manifold polygonal curve. We chose in this paper to describe the input PFC in terms of curved, rather than straight, streets to emphasize the fact that the sampling vertices along these curved streets are not considered as vertices of the graph G and to not decompose the curved street into small straight or curved edge segments each having different sidewalks. The exact geometry of a street, its sample points, and its decomposition into straight line segments will be lost during rasterization. Only the connectivity between faces, streets, and junction or turn vertices will be preserved.

Rasterization distinguishes two types of pixels: ***shared*** pixels (which contain a junction, dead-end, or turn vertex of G or that are traversed by an street of G) and ***private*** pixels (which are each fully contained in a single face of the PFC).

To simplify explanations and terminology, in the remainder of the paper, we assume that we first construct a ***refined*** version, R, of graph G. R is obtained by initializing it to G and by identifying all street segments where a street, E, splits (i.e., enters and leaves) a pixel, P. In each such segment, we insert (into R) a new ***split*** vertex, V, anywhere along that segment. A street E that splits several pixels will be split several times, once per splitting segment. Note that a curved street may split the same pixel more than once as shown in Fig. 3 at pixel (1,1) for the left-most of the two streets between vertices A and B. These insertions do not alter the geometry of the graph, but increases the street count.

R is defined implicitly by G and by the pixel grid. Although it could be constructed easily during the above rasterization process, it can be processed from its implicit form directly using the representation of G without the need for storing the split vertices explicitly. But for simplicity of presentation, we assume below that R has been constructed and that we have access to its sidewalks and to primary operators on them.

For clarity and conciseness, to distinguish G from R, we use the term ***dart*** when referring to the sidewalks of R and we will use the term ***edge*** when referring to a street of R. We will use lowercase letters for darts (for example, d) and for their operators (d.o, d.p, d.n, d.v, d.e, d.f). Note that to each sidewalk of G corresponds a set of one or more darts of R.

Our solution is based on the fundamental validity assumption: we require that no street of G (and hence no edge of R) lies entirely in a single pixel. There are two important benefits of enforcing this assumption and of considering R rather than G: (1) No edge of R splits a pixel and (2) in each shared pixel, each connected component of R contains exactly one vertex of R.

A simple way to ensure that the pixel grid has sufficient resolution to preserve the original topology of G is to require that the pixel diagonal be smaller than the shortest edge of G.

3.2. Computing and ordering the crossings of a pixel

We process each shared pixel, P, one at a time. We identify each dart, d, of R that enters P (i.e., for which the starting vertex, d.v, lies outside of P and the ending vertex d.o.v lies in P). We compute the corresponding ***crossing*** (unique point where edge d.e crosses one of the roads in the boundary of P). We label the crossing using one of the 4 ***tags*** {'W','N','E','S'} depending on whether the crossing lies on the left, top, right or bottom road around P. Finally, we ***order*** the crossings of P clockwise around P, starting at the lower left crossroad of P.

To order the crossings along a road, we derive a rational expression for each crossing based on the standard slope-intercept formula. Assuming that the coordinates of the pixels are represented (or have been quantized) to 16 bits each, the exact ordering can be computed by performing only additions and multiplications on long integers.

3.2.1. Local dart indices

The integer ***symbol index***, s, is used to identify the crossings and crossroads of a pixel P in clockwise order around the border of P, starting from the lower-left crossroad. Observe that each crossing corresponds to a set of two opposite darts, d and d.o, of R. Each dart, d, intersects two pixels: it ***exits*** a pixel, Q, and ***enters*** a road-adjacent pixel, P. Dart d has a different dart index in each one of these two pixels.

Consider a dart, d, that enters or exits pixel P. Let ***i(d,P)*** denote the dart index of d with respect to P. Note that d is associated with a single crossing, x. Let x denote the symbol index of x with respect to P. Then i(d,P) is $2x$, if d enters P, and $2x + 1$, otherwise.

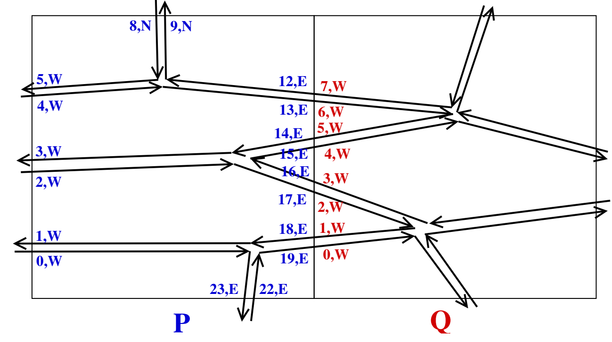


Figure 4: Local indices of darts in adjacent pixels. Notice that two indices are skipped for each crossroad. Global dart identifiers (15,'E',P) and (4,'W',Q) refer to the same dart, which exits P by the East side and enters Q from the West side.

By this convention, i(d,P) is even for exiting darts and odd for entering darts, and the dart indices of d and d.o in the same pixel are contiguous integers. Figure 4 illustrates this notation.

Note that four values of s are not associated with darts: they correspond to the crossroads of P.

A dart, d, that enters or exits pixel P is identified ***locally*** in P by (i,T), where i is the dart index of d in P and T is the tag of the crossing associated with d in P, and ***globally*** by the triplet (i,T,P).

3.2.2. Conversion between local indices of a dart

Given the local index, i, of a dart, d, that enters (resp. exits) pixel P with tag T, we want to compute the pixel Q that d exits (resp. enters) and the local index j and tag U of d in Q. Below, we describe informally the conversion procedure, (j,U,Q)=Convert(i,T,P), using an example that illustrates one of the possible configurations.

In Fig. 4, we know that dart (15,'E',P) exits pixel P, because it's dart index ($i = 15$) in P is odd. Since d exits P by the east side (T='E'), Q lies on the right of P and d enters Q by the west side (U='W'). We also know that j must be even. Below, we explain how we compute j.

Let B be the road between P and Q crossed by d. Let T (resp. U) be the tag associated with B in P (resp. Q). To compute j, we first compute:

- c = number of darts with tag T in P;
- x_0 = the local index of the first dart with tag T in P;
- e_0 = the local index of the first dart with tag U in Q.

and then set $j = e_0 + (x_0 + c - i - 1)$.

Note that c, x_0, and e_0 are always even. Thus, if i is even, then j is odd, and vice versa. In the example of Fig. 4, $j = 4$, because this is the 5th dart clockwise along the 'E' road of Q ($i = 15$, $c = 8$, $x_0 = 12$ and $e_0 = 0$).

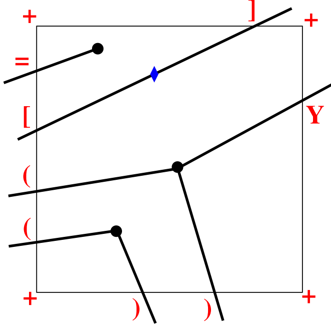


Figure 5: A pixel with 8 crossings. Its next map is encoded by the word "(([=+]+Y+))+".

3.3. Encoding of the combinatorial map in a pixel

In the previous subsection, we explained how to convert between the two local representations of a dart in adjacent pixels. In this section, we propose a compact encoding of the primary operators (previous, opposite, and next) between darts that enter pixel P and darts that exit it.

3.3.1. Computing the local index of the opposite dart

Given dart, d, with local index i in P, we do not need to store an explicit encoding of the local index of the opposite dart, d.o, because, as explained earlier, it is defined implicitly as $i + 1$ (if i is even) or as $i - 1$ (otherwise). Remember that darts d and d.o pass through the same crossing. Hence they have the same tag in P.

3.3.2. Compact encoding of the next dart map

Our encoding of the topology of R in a given pixel P builds a ***word*** (sequence of symbols) by considering, in clockwise order around the pixel, each crossing and each crossroad (pixel corner) and by assigning a symbol to each. The '+' symbols correspond to crossroads and is used to separate the crossings into the four sub-lists ('W', 'N', 'E', and 'S'). Furthermore, the occurrence of the 4th '+' symbol always indicates the end of the word. Figure 5 shows the word for a particular pixel.

Our choice of the symbol assigned to each crossing and its computation, described below, encodes the connected ***components*** of R in P (i.e., the subsets of crossings that are connected by the portion of R in P) and makes it easy to compute the "next" map for each incoming dart.

We use different symbols to distinguish crossings that open (i.e., start) a new connected component, that continue (i.e., add a branch to) the current component, and that close the current component. We use different symbols to distinguish between components connected to a split vertex added in R, components connected to an original junction or turn in G, and edges leading to a dead end.

For fully general PFC (with dead ends and turns), we define the following seven symbols:

- '=' : leads to a dead end
- '+' : denotes crossroad
- '[' : starts a component connected to a split
- ']' : ends a component connected to a split
- '(' : starts a component connected to a junction
- 'Y' : adds a branch to a junction
- ')' : ends a component connected to a junction

An example is shown in Fig. 5.

Starting from the lower left corner, the first crossing is labelled '(', because it denotes the first encounter with a new component connected to a turn vertex.

The next crossing is also labelled '(', because we encounter a new component, this time connected to a junction vertex (our encoding does not distinguish explicitly between junction and turn vertices, but this distinction may be easily established during traversal by using the "rotate" operators on the darts of the component to count the number of incident edges on the vertex to which the component is connected).

The next crossing is labelled '[', indicating that we have encountered a new component connected to a split vertex (that is not a proper vertex of G, but was inserted during the construction of R).

The next crossing is labelled '=', indicating that we have encountered a new component, but that the dart, d, that enters P through that crossing is connected to a dead-end vertex. Hence, the connected component is closed right away. At this point, we still have 3 active components.

The next crossing is labelled ']' to indicate that we close the component that was activated last (the component of the split vertex). At this point, we still have two active components left.

The next crossing is labelled 'Y', indicating that it corresponds to an edge that is connected to the junction vertex, V, of the current (last opened) component. 'Y' also implies that this crossing is neither the first nor the last (in the order around P) that is connected to V.

The next crossing is labelled ')', indicating that we have reached the last edge of the current component, which is the one connected to the valence 3 junction vertex.

The final crossing is labelled ')', indicating that we have reached the last of the edges connected to the turn vertex.

3.4. Algorithm for computing a pixel's word

To compute the word (i.e., the array W[] of symbols) for a given pixel, P, we start with the following two steps: (1) we identify all the edges of R that intersect P (remember that each edge of R has a single crossing with the boundary of P) and (2) we order their crossing around P.

Then, we visit them in order around P and, for each incoming dart, d, we use the dart operators in R to assess what kind of a component (dead end, split, junction or turn) d connects to and whether d opens, continues, or closes a component. Depending on these findings, we set

the symbol of the corresponding crossing and add it to the word of P (write it into the next available entry of the W array). The details are shown in Algo. 1.

Algorithm 1: Symbol of a dart

```
Input: d: a dart that enters into pixel P.
Output: symbol associated with the crossing of d.
d2← d;
repeat
  | d2← d2.n;
until d2.n.v is not in pixel P;
if d=d2.o then return '=';
else if isSplitVertex(d.n.v) then
  | if i(d.n,P)>i(d,P) then return '[';
  | else return ']';
else if i(d2,P)>i(d,P) then return '(';
else
  | d2← d.o;
  | repeat
  |   | d2← d2.p;
  | until d2.p.v is not in pixel P;
  | if i(d2,P)>i(d,P) then return 'Y';
  | else return ')';
```

As we progress around the border of P, we keep track of the which road we are on and append to W a '+' each time we reach a crossroad. For each pixel, we store a compact encoding of its word.

To avoid the computational cost of visiting all edges of R for each pixel, one may envision rasterizing R once and keeping track, at each pixel, of the edges that stab it. But such a solution would require a considerable amount of storage per pixel. Our implementation processes the pixels one row at a time, in top-down order, and maintains a list of active edges of R (those stabbing the current row).

The connectivity of R is fully represented by the list of shared pixels and by their words.

3.5. Geometric error bound

rPFC encodes the ordered sequence of pixels stabbed by each street, E, of the original graph G. Let U be the union of these pixels and C be their center-points. U is the Minkowski sum of C with a pixel centered at the origin. Since, by construction, E is contained in U, the Hausdorff distance between E and C does not exceed half of the pixel diagonal.

3.6. Adaptive version of rPFC

The presence of relatively short streets in G imposes an upper-bound on pixel size. When the exact topology must be preserved and simplification is not an option, such a constraint increases the pixel resolution and the number of stabbed pixels, hence also the storage and performance cost. To address this problem, we envision the option of a variable resolution rPFC. It stores a coarse resolution rPFC in which some of the stabbed pixels may be identified. These are each split into 4 quadrants, for which the word encoding may be accessed or transmitted through progressive refinements. This can be repeated recursively (as in a quad-tree).

4. Usage of rPFC

Here, we explain how we use these words and simple local procedures to support all primary dart and sidewalk operators and to retrieve associations of color or other attributes with the faces, edges, or vertices of the PFC.

4.1. Using pixel words for supporting dart operators in R

We show here how to use the word (the content of array W) of a pixel P to compute the dart indices of d.o, d.n and d.p from the dart index of an entering dart d.

4.1.1. Intra-pixel "opposite"

Let k be the dart index of a dart d that enters pixel P. The dart index of d.o is $k+1$ if k is even and $k-1$ otherwise. Moreover, d.o and d have the same tag.

4.1.2. Intra-pixel "next"

Given the local dart index, k, of a dart, d, that enters a pixel, P, we use the following procedure to compute the local index of d.n that exits P, and the tag of d.n. Note that the starting vertex of d.n is in P, because any dart of R that enters a pixel P reaches a vertex of R in P.

Computing d.n requires distinguishing different cases. A relatively simple algorithm is presented in Algo. 2 and explained below.

Let k be the dart index of entering dart d with respect to pixel P and let T be its tag. Let $c = k/2$ be the symbol index of the corresponding crossing. Let s be the symbol associated with that crossing.

When s is '=', d reaches a dead end and hence d.n is d.o. Therefore, the dart index of d.n is $k+1$.

When s is '[', d is the first dart around P that reaches a split vertex, v. We want to find the unique dart d.n that leaves v and is not d.o. To find the symbol index c of its crossing, we imagine walking around the boundary of P clockwise until we find the matching ']' symbol. Here, the term matching implies that we skip over, possibly nested, matching pairs ('[',']' or '(',')') and ignore all other symbols. In practice, we do not walk around P, but simply read the symbols in the order in which they appear in the word associated with P, starting from symbol index $c = d/2+1$. The dart index of d.n is $2c+1$. We also want to know the tag, T, of d.n. To compute it, we initialize it (T=U), and advance T to the next tag, each time we pass a '+'.

When s is '(', we follow the same process as for '['.

When s is ')', 'Y' or ']', we follow the same process, but walking backwards (i.e., reading the symbols in reverse order). Note that for ')' and 'Y', we need to stop either

Algorithm 2: Dart index of d.n for entering dart d

```
Input: (k,T): the local index and the tag of a dart d;
       W: the word of the pixel entered by d.
Output: (l,U): the local index and the tag of d.n.
c ← k/2 ;                    /* symbol index of dart d */
s ← W[c];
U ← T;                                 /* road of d */
if s='=' then
    return (k+1,T) ;                   /* dead end */
else
    if s='[' or s='(' then n ← 1; dir← 1;
    else n ← -1; dir ← -1;
    repeat
        c ← c+dir;  s ← W[c];
        if s='[' or s='(' then n++;
        else if s=']' or s=')' then n−−;
        else if s='+' then
            if dir=1 then
                U ← next road after U around P
            else
                U ← next road before U around P
    until n=0 or (n=-1 and s='Y');
    return (2×c+1,U)
```

when we found the matching '(' or when we found a 'Y' symbol that is associated with a crossing connected to the junction vertex reached by d.

To understand Algo. 2, observe that, when we walk clockwise around P and encounter a new component, we encounter first ***openings*** ('(' and '[') before the corresponding ***closings***. So, in Algo. 2, we keep the nesting count, n, of active components ('(' and '[' increment n and ')' and ']' decrements it) and return the index of the outgoing dart of the crossing for which $n = 0$.

When entering dart d correspond to a closing, we must move counterclockwise, skip over pairs of matching closings and openings, until we find an opening that matches the closing of d. For conciseness, in Algo. 2, we use the same loop to handle both clockwise and counterclockwise traversal. To differentiate them we use a variable dir. Furthermore, because openings increment the counter n and closings decrement it, when walking counterclockwise, n is negative. Also note that we use the test ($n = -1$ and s='Y') to identify the first 'Y' crossing encountered while walking counterclockwise from d.

4.1.3. Intra-pixel "previous"

Given the dart index of a dart, d, that exits pixel P, computing d.p is done by using exactly the same principle as in Algo. 2, but changing the direction of the traversals.

4.2. Inter-pixel operators

The previous subsection focused on computing intra-pixel operators: dart indices for darts d.p, d.o, and d.n that lie in the same pixel as dart d. Here, we discuss inter-pixel operators for next, previous and opposite.

Each dart, d, has two global descriptors in an rPFC: (x,T,P) in the pixel P that it exits and (e,U,Q) in the pixel Q that it enters. Remember that x is odd and e is even. By convention, we associate dart d with (x,T,P) because pixel P contains the vertex where d starts.

So, d.n is obtained by starting with the descriptor (x,T,P) of d, using the Convert() procedure, given in Sect. 3.2.2, to obtain descriptor (y,V,Q), and performing the intra-pixel next operator on (y,V,Q) in Q.

Similarly, d.p is obtained by starting with the descriptor (x,T,P) of d, performing the intra-pixel previous operator on (x,T,P) and then using Convert() on the result to obtain descriptor (y,V,Q) in Q.

Finally, d.o is obtained by starting with the descriptor (x,T,P) of d, using Convert() to obtain descriptor (y,V,Q), and performing the intra-pixel opposite operator on (y,V,Q) in Q.

4.3. Mapping between darts of R and sidewalk of G

Sidewalks of G and operators on them can be implemented using darts of R and their operators. Remember that each sidewalk of G is decomposed into a series of one or more darts of R, that are connected by split vertices. We associate a sidewalk, S, of G with the first dart, d, of R in the corresponding list. Hence, S inherit the vertex and face of d. Let ***dart***(S) denote the dart in R that is associated with sidewalk S of G. Similarly, let ***Sidewalk***(d) denote the sidewalk of G associated with dart d of R.

Let d.e denote the last dart of a series that starts at d and contains all the darts of the same sidewalk of G. d.e is computed by iterating "next" until we reach a dart with an end vertex that is not a split. Similarly, d.b is the first dart of a series that ends at dart d and contains all the darts of the same sidewalk (d.p is computed by iterating "previous" until we reach a dart with a beginning vertex that is not a split).

We can implement the primary operators on sidewalks as follows:

- S.N = Sidewalk(dart(S).e.n)
- S.O = Sidewalk(dart(S).e.o)
- S.P = Sidewalk(dart(S).p.b)

Hence, we are able to use the S.P, S.O, and S.N operators on G from the information contained in the words stored at the shared pixels of the proposed rPFC.

4.4. Indexing vertices, edges and faces

In some applications, it is desired to associate an attribute (color or property) with each face, edge, vertex. In modern data structures for face meshes, this is often done for face attributes by associating consecutive integer IDs with the faces and by storing the attribute values in an

array indexed by the face IDs. For example F[f] may be the value of face attribute (say color) of face with ID f.

Let us assume that such an array F[] exists and is controlled by the application.

The question that we address here is not the cost of storing the attribute values, but the space and time cost of using the rPFC to associate each dart, d, of R with the face ID, d.f, of the incident face.

We also must address the cost of associating each face, f, with at least one dart, d, on its boundary, such that d.f=f. We propose a partial solution inspired by previous work on mesh compression [8, 9, 10], where the faces IDs are assigned based on a specific traversal order of the graph. For example, we can use the depth-first traversal proposed in [11] and assign the next available integer f to each face when it is first encountered. A fast approach that only requires a relatively shallow stack requires storing a one-bit marker with each vertex to indicate whether the vertex has already been visited.

This approach allows to use the attribute F[f] when processing (for example coloring) the corresponding face. The solution can also be used to support the picking of a face by clicking on a private pixel (it suffices to use the above approach to render each face with a color that encodes its index and retrieve the color of the picked pixel). The advantage of this solution is that it does not require any additional storage (except of course for the array F[] of attributed). Similar solutions may be used to associate attributes with the darts, vertices, and edges of R and hence of G.

We see two important drawbacks of such a partial approach: (1) Accessing (measuring or displaying) a particular vertex, edge, or face requires performing the traversal of the entire graph. (2) When an algorithm uses the dart operators to perform a different or a partial graph traversal and arrives at a dart d, we do not have direct access to its ID, nor to the IDs of its vertex d.v or face d.f. Of course, these could be retrieved at no additional storage cost by performing the depth first traversal and stopping when it reaches d, but this cost would raise the computational cost of the algorithm and would lower its performance.

5. Data structures for rPFC

In this section, we first discuss strategies for compact representations of individual symbols and words and then outline four different architectures for rPFC representations: (1) A matrix with constant size entries, one per pixels, that has many advantages, but requires a large amount of storage in configurations with a relatively large proportion of private pixels, (2) A pixel adjacency graph with variable size representation of words that requires storing with most shared pixels the addresses of their neighbors, (3) A version of the above that does not store private pixels and is attractive or modeling street maps or river graphs, and (4) A compact version of the above where addresses

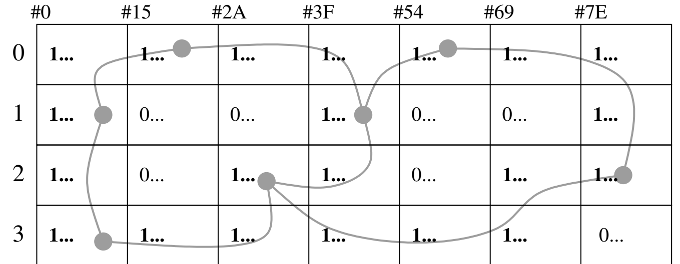


Figure 6: Example of rPFC (grey) encoded with the **V1** architecture. The maximal number of crossings, m, is 4. Thus, each word, W[i,j], is stored using 25 bits (addresses are multiples of #15 in hexadecimal). Words starting with '1' identify stabbed pixels.

of neighbors are not stored, but need to be retrieved by a partial traversal of the corresponding row of pixels.

5.1. Compact encoding of symbols and words

The set of symbols used in an rPFC depends on the topology of G and on the application's needs. But in any case, we can replace ']' by ')' because, without ambiguity, each closes the component opened last either by '[' or by '('. When using the remaining 6 symbols "()Y[=+", the symbol entropy ranges from 1.3 to 1.9 depending on the model and pixel size. It averages 1.5 bits per symbol for targeted applications.

Rather than using Huffman or entropy codes, we use a simple and unique encoding of each symbol and form the word by concatenating them. If the model has no dandling edges ('=' is not needed) and if we do not need to differentiate splits from turns ('[' is not needed), we have 4 symbols "()Y+" and use 2 bits for each. Otherwise, we represent '+' by a single bit 0 (since 66% of symbols are '+') and use a fixed or variable length encoding of the other symbols (for example 100 for '(', 101 for 'Y', 110 for ')', and 111 for '[' if '=' is not needed; storage costs for this encoding are reported in Sect. 6).

5.2. V1: Matrix of fixed size record

In the **V1** architecture, the rPFC is encoded by a ***matrix***, ***W***[], of words, one for each pixel (shared or private) (see Fig. 6). Each word, W[i,j], is stored using $4m + 5$ bits, where m is the maximum of the numbers of crossings per pixel (we use 1 bit to differentiate private from shared pixels, and for each shared pixel, we use 1 bit for each one of the four '+' symbols and 4 bits for each one of the other symbols). In our experiments, about 98% of pixels have no more than 6 crossings (average for all our models and all our different number of pixels, see Sect. 6) and hence could be encoded in 32 bits, about 99% of pixels have no more than 14 crossings and hence could be encoded in 64 bits, but, in some models, a minute fraction of pixels have more crossings (up to 18 in the worst case in our experiments) and should be encoded using an exception mechanism. For private pixels, this constant storage

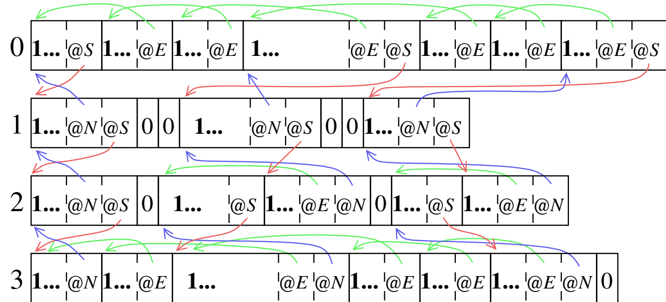


Figure 7: rPFC of Fig. 6 encoded with the **V2** architecture. Red (resp. blue, green) arrows represent @S (resp. @N, @E) addresses. No information is stored for private pixels, except for a 1 bit mask.

(minus the one bit) may be used to store the face ID and possibly the distance to the boundary or some other scalar field.

The main properties of V1 are: (1) identification of private pixels and face membership classification at constant cost, and (2) constant cost access to the encoding $W[i,j]$ of the word of each shared pixel during graph traversal;

5.3. V2: Pixel adjacency graph with variable size records

In the **V2** architecture, the rPFC is encoded by an ***array***, ***R***[], of bitsets, one for each row (see example in Fig. 7). Each bitset, R[j], is the concatenation of the encoding of all words of pixels in row j. Each word is now encoded using variable size records. To support constant time cost access to neighboring shared pixels during graph traversal, we store, at each shared pixel (i,j), in addition to its private/shared bit and to its word, up to three references: (1) the address (beginning of the descriptor), @E, of the previous pixel $(i-1,j)$ in row j, (2) the address, @S, of the south neighbor $(i,j+1)$ in row $j+1$, and (3) the address, @N, of the north neighbor $(i,j-1)$ in row $j-1$. @E is encoded as a relative displacement (the bit-length of the descriptor of pixel $(i-1,j)$). @N and @S are absolute bit-indices in the corresponding rows. Each address is encoded using a minimal, although known, number of bits (typically 6 for @E and 12 for @S and @N) that depends on the length of each row and on the maximum record size. Each reference is only encoded when the corresponding neighbor pixel is shared. Note that we do not need to store the address of pixel $(i+1,j)$ to access it from pixel (i,j). Even though the length of the word of (i,j) is not fixed, and hence, not know a priori, it can be easily established be identifying the last '+' in that word.

We use a fixed format for private pixels. It includes a 0 to distinguish them from shared ones and also a fixed (possibly null) number of bits that may be used to identify the face or material type and optionally to store a scalar field, which may for example capture the distance from the boundary of the face or some physical property, such as heat.

5.4. V3: Version of V2 without private pixels

The **V3** architecture, is almost identical to V2, but does not store anything for private pixels. It is particularly useful for applications which do not support face membership or material queries and are instead focused on the traversal and analysis of graph G. With additional storage per face and with additional cost per query, V3 could be extended to support such queries. The layout for the **V3** architecture is the one shown in Fig. 7, but without the '0' pixels.

5.5. V4: Version of V3 without addresses

The **V4** architecture is a variant of V3 where we trade performance for space. Instead of storing for each shared pixel up to three neighbors' addresses (as explained for V2), we recompute these addresses when needed. This solution is based on the key observation upon that the @E, @N, and @S addresses can be computed (each time they are needed) by starting from the beginning of the corresponding row ($j+1$, j, or $j-1$) and by walking East by the desired number of records.

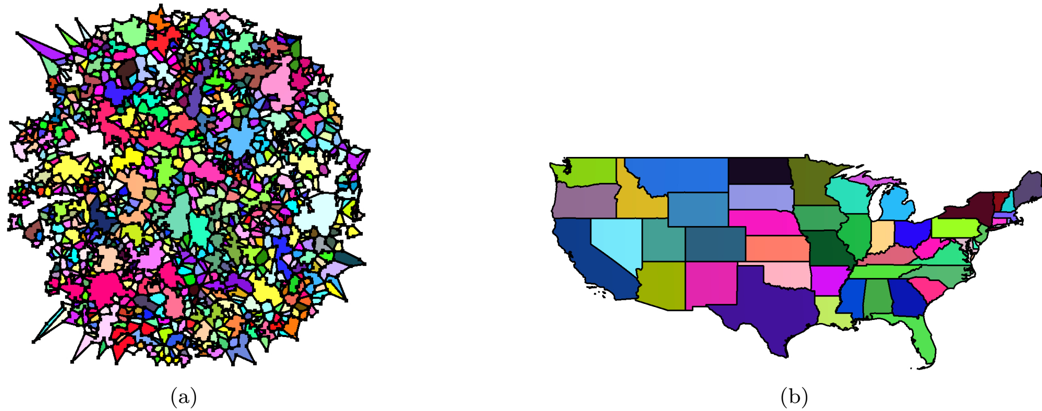
We could store the shared/private bit for each pixel to facilitate this process, but we do not have to. Indeed, for row j, we simply walk until we encounter a pixel that has (i,j) as East neighbor. To identify the south neighbor, we walk along row $j+1$ and count the number of visited shared pixels that have a north neighbor (that have at least one crossing on their north road). We stop when this count matches to the number of pixels along row j, from pixel $(0,j)$ to pixel (i,j), that have a south neighbor. The layout for the **V4** architecture is the one shown in Fig. 7, but without the '0' pixels and without the @S, @N, and @E addresses.

5.6. Indexing vertices and faces

In some applications, it is desired to provide direct access from a face to one of its darts and inversely, from a dart to its face. To do so, for each face, f, we walk along the loop of its darts and identify the ***root*** dart (the dart d with the lexicographically smallest index composed of the fixed-length strings representing coordinates of the pixel containing d.v and of the dart index in that pixel). We use array Root to encode the root dart of face f as Root[f]. This provides constant cost access from f to d. Now, to compute d.f, we use the next operator to walk from d around the loop of f. During that walk, we identify the root dart r of f. Then, we search f such that Root[f]=r. To reduce the cost of the search from $O(F)$ to $O(logF)$, where F is the face count, we sort the face indices so that the array Root is sorted.

6. Experiments

We have computed rPFC representations for two different types of 2D face meshes. 2D Voronoi diagram of random points and GIS data of countries.



(a) (b)

Figure 8: (a) One face mesh obtained from a Voronoi 2D diagram of 2,000 random points, where 20% of edges are randomly removed. (b) The GIS map of USA states.

To ensure that the input data satisfies our constraint that no edge lies entirely in a pixel, in a preprocessing step we simplify the input face mesh (one could use a variant of [12] or a feature preserving simplification [13]).

In these experiments, we use the encoding given in Sect. 5.1, with 1 bit for '+', 3 bits for other symbols "(Y)[", and without '=' for dead-ends.

6.1. Data sets used in experiments

Firstly, we use face meshes computed as 2D Voronoi diagrams of random points generated in a disk of radius 500. Five face meshes were generated from 1,000, 2,000, 4,000, 8,000 and 16,000 points. In a post-processing, 20% of edges are randomly removed from the 2D Voronoi diagram, in order to decrease the face mesh regularity. The face mesh obtained for 2,000 points is shown in Fig. 8(a).

The number of vertices of these face meshes are respectively 1,550, 3,174, 6,479, 13,094 and 26,244; the number of edges 1,945, 3,976, 8,113, 16,439, and 33,032; and the number of faces 399, 806, 1,644, 3,359, and 6,810.

Secondly, we used geographic information system (GIS) data obtained from `http://www.gadm.org/` web site, which provides a spatial database of the location of the world's administrative areas. We used Brasil, China, France, England, Russia and USA countries (see Fig. 8(b) for the USA data set). The files are provided in the Shapefile format, a popular geospatial vector data format for GIS softwares.

The number of vertices of the original meshes are respectively 560,823, 815,556, 246,880, 424,717, 1,662,902 and 57,406; the number of edges are 560,882, 815,622, 247,606, 424,993, 1,663,102, 57,506; and the number of faces are 849, 2,021, 763, 1,113, 5,899, 345.

In both cases, we computed our four versions of rPFC representations, for numbers of pixels in the horizontal direction equal to 128, 256, 512, 1024, 2048, 4096, 8192, 16384 (while computing automatically the number of pixels in the vertical direction).

The entropy per symbol depends on the model and the pixel sizes. In average for the five synthetic face meshes, it starts from $e = 1.9$ for 128 pixels in the horizontal direction and decreases to $e = 1.3$ for 16,384 pixels. The entropy per words, in average for the five models, starts from $e = 6.6$ and decreases to $e = 2.6$. Entropy per words decreases when the number of pixels increases because the number of different words does.

In average for the six GIS maps, the entropy per symbol starts from $e = 1.9$ for 128 pixels in the horizontal direction and decreases to $e = 1.4$ for 16384 pixels. The entropy per words, in average for the six models, starts from $e = 5$ and decreases to $e = 3$. The maximal size of pixel words is between 6 and 16, and is equal to 6.04 in average.

6.2. Storage analysis

In Table 1, we compared the storage used in the proposed data structures. Column CM reports storage for a combinatorial map representation of the simplified version of G. It stores, three references (d.n, d.v, and d.f) for each dart, d, two integer pixel coordinates for each vertex, and one dart reference for each face. Column CMS reports storage for a rasterized version of CM, augmented with a bit mask that identifies shared pixels, and, for each shared pixel, with a list of references to the vertices it contains. Although the largest of all four, storage for V1 is smaller than uncompressed (32 bits per pixel) image format, except at low resolution. V2 uses between 35% and 80% less storage than CMS, while providing more functionality. V3 is between 80% and 95% more compact than CMS. It is also more compact than CM for GIS models. V4 is the most compact. It requires between 85% and 99% less storage than CMS. For the GIS maps, it uses less storage than CM. For our synthetic meshes, it becomes more storage expensive than CM at high pixel resolution, when average-length edges stab several pixels.

To further reduce storage, we identify the 2^k most ***frequent words*** and use $k + 1$ bits to encode each of them

	#pixels	#simplif-v	#split-v	CM		CMS		V1		V2		V3		V4	
				kb	bps	kb	bps	kb	bps	kb	bps	kb	bps	kb	bps
Synth. face meshes	128×128	2,057	9,532	50	24	195	97	76	38	34	16	34	17	16	8.2
	256×257	4,710	24,787	115	14	546	68	272	34	92	11	88	11	39	4.8
	512×514	7,200	54,532	163	5.2	1,518	47	949	29	200	6.2	170	5.3	60	1.9
	1,024×1,028	8,650	112,336	189	1.5	3,806	29	3,359	26	446	3.5	318	2.5	90	0.70
	2,048×2,055	9,373	227,575	201	0.39	7,905	15	13,375	26	1,142	2.2	615	1.2	148	0.29
	4,096×4,111	9,741	457,976	208	0.10	16,871	8.2	53,461	26	3,302	1.6	1,240	0.60	265	0.13
	8,192×8,221	9,919	918,663	211	0.03	37,883	4.6	189,101	23	10,731	1.3	2,443	0.30	498	0.06
	16,384×16,443	10,015	1,840,125	212	0.0065	92,244	2.8	756,396	23	38,107	1.1	5,004	0.15	964	0.029
GIS maps	128×69	577	1,403	8	7.3	24	22	32	29	5	4.5	4	3.7	2	1.7
	256×138	1,509	3,256	22	5.1	63	14	126	29	13	3.0	9	2.0	3	0.80
	512×275	3,501	7,429	49	2.9	149	8.7	499	29	36	2.1	19	1.1	7	0.39
	1,024×549	7,402	16,666	102	1.5	363	5.3	1,991	29	109	1.6	41	0.60	14	0.20
	2,048×1,097	14,450	36,642	215	0.79	1,188	4.3	8,778	32	367	1.3	91	0.33	28	0.10
	4,096×2,194	26,204	78,544	377	0.34	3,486	3.2	35,107	32	1,301	1.2	194	0.18	58	0.05
	8,192×4,387	46,093	165,303	640	0.15	9,425	2.1	127,229	29	4,819	1.1	412	0.09	117	0.03
	16,384×8,774	77,963	341,157	1,591	0.090	27,996	1.6	508,901	29	18,449	1.0	853	0.049	231	0.013

Table 1: Memory space for the synthetic face meshes (first 8 rows) and for the GIS maps (last 8 rows). Each value is the mean for the all tested model (five generated face meshes or six GIS maps). #pixel is the number of pixels; #simplif-v the number of vertices of the simplified mesh; #split-v the number of split vertices added by the rasterization. CM (resp. CMS) is the computed memory size of an equivalent representation of a combinatorial map of the simplified version of G (resp. R). CMS includes, for each pixel, the list of vertices of R that it contains. V1, . . . , V4 are our different representations, using frequent words. V2 versions do not store any information for private pixels, which are thus encoded by only 1 bit. Each memory size is given in kilo-bytes (*kb* columns) and in bit-per-pixel (*bps* columns).

Synthetic face meshes					GIS maps				
#pixels	V3		V4		#pixels	V3		V4	
128×128	7,215	44%	201,576	1,232%	128×69	862	9.8%	11,315	129%
256×257	17,914	27%	759,020	1,156%	256×138	1,895	5.4%	29,100	82%
512×514	37,208	14%	1,940,132	738%	512×275	4,177	2.9%	74,386	53%
1,024×1,028	74,027	7.0%	4,122,003	392%	1,024×549	9,182	1.6%	186,591	33%
2,048×2,055	147,380	3.5%	8,428,277	200%	2,048×1,097	19,876	0.89%	448,478	20%
4,096×4,111	294,040	1.7%	17,030,686	101%	4,096×2,194	42,038	0.47%	1,022,653	11%
8,192×8,221	587,287	0.87%	34,247,793	51%	8,192×4,387	87,320	0.24%	2,247,795	6.3%
16,384×16,443	1,173,840	0.44%	68,733,223	25%	16,384×8,774	178,425	0.12%	4,793,714	3.3%

Table 2: Comparison of performance between V3 and V4, showing the total number of pixels accessed during a walk around each face of G.

(using the leading bit to distinguish them form others). We pick k to minimize total storage (typically $1 \leq k \leq 6$). This trivial compression reduces storage by about 5% for V2, 20% for V3, and 58% for V4.

6.3. Comparison of traversed elements

We did a second experiment in order to compare our different representations in term of operations. For that, we iterated through all the faces of each rPFC, and for each one, we traversed through its contour by using the next operation. During this traversal, we computed the number of accessed pixels. Results are given in Table 2 for the synthetic face meshes and the GIS maps.

We can see on these results that the number of traversed pixels is much more important when iterating through the V4 version than for the V3 representation. Indeed, each time we need to move to the pixel above (resp. below) of the current pixel, pixels in the previous (resp. next) row are iterated starting from the beginning of the row. In average, V4 traverses between 10 and 58 times the number of pixels traversed by V3. This implies of course a big overhead for operations, but this is the price to pay in order to have a very compact representation.

In order to limit this overhead, it is possible to store traversed elements during an operation and use this cache when possible instead of recomputing the next, previous or opposite relations.

7. Relation to prior art

We divide relevant prior art into four categories: (1) rasterized image with different colors per face, (2) original edge graph connectivity, (3) implicit connectivity inferred from crossings roads, and (4) inferred or encoded connectivity from a unique vertex per shared pixel.

7.1. Rasterized image

The partition of a portion of the plane into regions (faces), each assigned a different color, may be represented as a rasterized image: coloring each pixel based on the region that contains its center. The image may be represented compactly using a fixed number of bytes per pixel.

More compact encodings include RLE (Run Length Encoding) and quadtree [14]. Such a digitization of a PFC not only suffers from the same quantization error as the rPFC, but also eliminates all cracks and dead ends and hence cannot be used to represent street networks. Furthermore it may disconnect regions at constrictions thinner than a pixel and offers no bound on the Hausdorff error between the original boundary of a face and the boundary of its rasterized version. The proposed rPFC does not suffer from these shortcomings and offers from the same localization capabilities (constant cost computation of the face ID for a query point in a private pixel) and streaming capabilities.

7.2. Graph connectivity

Various data structures for polygon meshes have been proposed over the last few decades: [15, 16, 17, 18, 19, 20, 21, 22, 23]. Some are discussed in [24]. Many of them store ***Geometry*** (the coordinates of each vertex) and, ***Connectivity*** (references to d.v, d.f, d.n, d.p, and d.o for each dart). More compact representations have been proposed for polygon graphs [25, 26] and for triangle meshes [27, 28, 29, 30, 10, 31, 17, 32]. Their connectivity storage cost per triangle is only a few times larger than that of compressed representations [8, 33, 34]. Even though some variants have been proposed recently that support streaming [30, 35] or graph editing [36], most these compact representations do not support efficient connectivity editing or streaming, because they assign vertex and triangle IDs in a specific order, and because they use global ID. Furthermore, they do not support spatial indexing.

The rPFC solution proposed here strives to unify and combine the best features of the above two categories of representations, but without resorting to the systematic resampling of the PFC edges that is preformed by approaches in the following two categories and that may result in drastic and unbounded connectivity changes.

7.3. Crossings with grid lines

Several approaches insert crossings where edges of G intersect inter-pixel grid lines and recompute or define implicitly a simplified (vertex free) topology of G inside each pixel (or voxel) [37, 38] or obtain a simplified surface through edge collapses that eliminate all original vertices [39].

In particular, the Layer Depth Image (LDI) and their "Hermite" extensions, the Layer Depth-Normal Image (LDNI) representations accelerate Boolean operations and other analysis and manufacturing planning tasks [1].

rPFC can be trivially augmented by associating with each crossing its geometric position along the corresponding road (for example encoded using 6 bits). In that case, rPFC could be viewed as an extension of the LDI model, where, in addition to the crossings, we encode the explicit connectivity inside each pixel, and hence do not need to infer it from the crossing or from the associated tangents.

7.4. A vertex per shared pixel

Arguably the dual of the above category is the set of approaches that generate a single vertex per shared pixel (or voxel) either via resampling, via edge contraction simplification, or by computing the dual of results obtained by solutions in the previous category [40, 41, 42].

rPFC could be viewed as an extension of such dual models: for a modest cost, it makes it possible to encode explicitly a variety of topologies with possibly multiple components and vertices per pixel.

A hierarchical structure (octree, kd-tree) is sometimes used to reduce storage cost for the private pixels (voxels) [43] or to support progressive refinements, of the connectivity [44] or of the topology [45], or to support adaptive resolution [46].

8. Summary and conclusion

The proposed rPFC approximates vertex and curved edge geometry of a PFC by the list of pixels they intersect, but represents graph connectivity exactly and hence supports exact topological graph traversal. It can accommodate non trivial topology in a pixel, including multiple vertices and multiple connected components, but assumes that no edge fits entirely inside a single pixel. If this happens, the invalid model must be simplified by a set of edge contractions or the pixel size must be reduced.

The rPFC decomposes the graph connectivity description into a set of local maps, one per pixel, and stores each map using a very compact representation. As such, it offers a new paradigm that makes it possible to provide spatial indexing to both quantized geometry and exact topology of a valid PFC that may be complex both combinatorially and topologically.

Future challenges include extensions to irregular grids, multi-resolution grids, and, of course, 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] T.-H. Kwok, Y. Chen, C. C. Wang, Geometric analysis and computation using layered depth-normal images for three-dimensional microfabrication, in: T. Baldacchini (Ed.), Three-Dimensional Microfabrication Using Two-photon Polymerization, Micro and Nano Technologies, William Andrew Publishing, Oxford, 2016, pp. 119–147.

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

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

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

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

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

[7] H. Edelsbrunner, E. P. Mücke, Simulation of simplicity, a technique to cope with degenerate cases in geometric computations, ACM Trans. Graphics 9 (1990) 66–104.

[8] J. Rossignac, Edgebreaker: Connectivity compression for triangle meshes, Trans. Vis. and Comput. Graph. 5 (1) (1999) 47–61.

[9] T. Gurung, M. Luffel, P. Lindstrom, J. Rossignac, Zipper: A compact connectivity data structure for triangle meshes, Computer-Aided Design 45 (2) (2013) 262–269.

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

[11] D. King, J. Rossignac, Guaranteed 3.67V bit encoding of planar triangle graphs, in: CCCG, 1999, pp. 146–149.

[12] J. Rossignac, P. Borrel, Multi-resolution 3d approximations for rendering complex scenes, in: Modeling in computer graphics, Springer Berlin Heidelberg, 1993, pp. 455–465.

[13] D. Salinas, F. Lafarge, P. Alliez, Structure-aware mesh decimation, Comput. Graph. Forum 34 (6) (2015) 211–227.

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

[15] B. G. Baumgart, Winged-edge polyhedron representation, Tech. rep., Stanford (1972).

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

[17] T. J. Alumbaugh, X. Jiao, Compact array-based mesh data structures, in: Proc. of 14th Int. Meshing Roundtable (IMR), 2005, pp. 485–503.

[18] D. P. Dobkin, M. J. Laszlo, Primitives for the manipulation of three-dimensional subdivisions, in: Proceedings of the Third Annual Symposium on Computational Geometry, SCG '87, ACM, New York, NY, USA, 1987, pp. 86–99.

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

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

[21] L. Kettner, Using generic programming for designing a data structure for polyhedral surfaces, Comp. Geometry 13 (1999) 65–90.

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

[23] D. Sieger, M. Botsch, Design, implementation, and evaluation of the `Surface_mesh` data structure, in: Proceedings of the 20th International Meshing Roundtable, 2011, pp. 533–550.

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

[25] G. E. Blelloch, A. Farzan, Succinct representations of separable graphs, in: CPM, 2010, pp. 138–150.

[26] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, C. Kadow, Compact representations of simplicial meshes in two and three dimensions, Int. Journal on Comp. Geometry and Applications 15 (1) (2005) 3–24.

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

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

[29] L. Castelli Aleardi, O. Devillers, Explicit array-based compact data structures for triangulations, in: Proc. 22th Ann. Internat. Sympos. Algorithms Comput., Vol. 7074 of LNCS, 2011, pp. 312–322.

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

[31] K. Yamanaka, S.-I. Nakano, A compact encoding of plane triangulations with efficient query supports, Inf. Process. Lett. 110 (18-19) (2010) 803–809.

[32] M. Luffel, T. Gurung, P. Lindstrom, J. Rossignac, Grouper: A compact, streamable triangle mesh data structure, Visualization and Computer Graphics, IEEE Transactions on 20 (1) (2014) 84–98.

[33] J. Rossignac, 3D mesh compression, in: Visualization Handbook, 2005, pp. 359–379.

[34] A. Maglo, G. Lavoué, F. Dupont, C. Hudelot, 3D mesh compression: Survey, comparisons, and emerging trends, ACM Comput. Surv. 47 (3) (2015) 44:1–44:41.

[35] M. Sati, P. Lindstrom, J. Rossignac, eBits, Comput. Aided Des. 78 (C) (2016) 168–178.

[36] L. C. Aleardi, O. Devillers, J. Rossignac, ESQ: Editable squad representation for triangle meshes, in: Graphics, Patterns and Images (SIBGRAPI), 2012 25th SIBGRAPI Conference on, IEEE, 2012, pp. 110–117.

[37] M. O. Benouamer, D. Michelucci, 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, ACM, New York, NY, USA, 1997, pp. 68–79.

[38] J. Shade, S. Gortler, L.-w. He, R. Szeliski, Layered depth images, in: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98, ACM, New York, NY, USA, 1998, pp. 231–242.

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

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

[41] N. Zhang, W. Hong, A. Kaufman, Dual contouring with topology-preserving simplification using enhanced cell representation, in: Proceedings of the Conference on Visualization '04, VIS '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 505–512.

[42] M. Campen, D. Bommes, L. Kobbelt, Dual loops meshing: Quality quad layouts on manifolds, ACM Trans. Graph. 31 (4) (2012) 110:1–110:11.

[43] B. Kim, Multi-phase fluid simulations using regional level sets, ACM Trans. Graph. 29 (6) (2010) 175:1–175:8.

[44] P. Kraemer, D. Cazier, D. Bechmann, Extension of half-edges for the representation of multiresolution subdivision surfaces, The Visual Computer 25 (2) (2009) 149–163.

[45] S. Valette, R. Chaine, R. Prost, Progressive lossless mesh compression via incremental parametric refinement, Computer Graphics Forum 28 (5) (2009) 1301–1310.

[46] P.-M. Gandoin, O. Devillers, Progressive lossless compression of arbitrary simplicial complexes, Transactions on Graph. 21 (3) (2002) 372–379.