Technical Report

Pattern Recognition and Image Processing Group
Institute of Visual Computing and Human-Centered Technology
TU Wien
Favoritenstrasse 9-11/193-03
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801 - 18661
Fax: +43 (1) 58801 - 18697
E-mail: jiri@prip.tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-154 April 13, 2022

Implicit unbounded n-Gmaps for images: A membrane-centric encoding using bit-flips

Jiří Hladůvka, Florian Bogner, and Walter G. Kropatsch

Abstract

The practical use of generalized combinatorial maps (n-Gmaps) to represent nD-images is limited by large memory requirements. Implicit representations, if well designed, can come to the rescue. Unbounded n-Gmaps are surrounded by an infinite background region that renders pixel-oriented implicit representation schemes inconvenient. The contribution of this paper is twofold. First, we propose an implicit, membrane-centric (rather than pixel-centric) arrangement of darts in unbounded n-Gmaps. Second, we introduce involutions based on bit flips, which allow efficient iterations in membranes and have the potential to speed up computations. We have validated our approach on a variety of 2D and 3D images, including those whose memory requirements would far exceed the main memory.

Implicit unbounded n-Gmaps for images: A membrane-centric encoding using bit-flips

Jiří Hladůvka, Florian Bogner, and Walter G. Kropatsch

Pattern Recognition and Image Processing Group
Vienna University of Technology
https://www.prip.tuwien.ac.at

Abstract. The practical use of generalized combinatorial maps (n-Gmaps) to represent nD-images is limited by large memory requirements. Implicit representations, if well designed, can come to the rescue. Unbounded n-Gmaps are surrounded by an infinite background region that renders pixel-oriented implicit representation schemes inconvenient. The contribution of this paper is twofold. First, we propose an implicit, membrane-centric (rather than pixel-centric) arrangement of darts in unbounded n-Gmaps. Second, we introduce involutions based on bit flips, which allow efficient iterations in membranes and have the potential to speed up computations. We have validated our approach on a variety of 2D and 3D images, including those whose memory requirements would far exceed the main memory.

Keywords: combinatorial maps · involutions · bitwise operations

1 Introduction

Image representation by region adjacency graph (RAG), dual graph, or a combinatorial map is the necessary step for a range of graph-based image processing applications and methods based on irregular hierarchies. Combinatorial maps and especially their generalized versions, i.e., the n-Gmaps [3], are currently the most versatile data structure to represent the content of higher dimensional images.

The main limitation of image representation by n-Gmaps, however, are both huge storage and high computational demands. For example, a 3-Gmap representing a moderately-sized 512^3 (1024^3) volume requires 192 GB (1.6 TB) of information. This is first infeasible to store and second time-demanding to process.

Thanks to their regular nature it is of a little surprise that nD images can be described by n-Gmaps implicitly (darts and the involutions computed on the fly) rather than explicitly (stored in the memory). Nevertheless it still remains important that such an implicit representation is (1) computationally efficient and (2) well-suited for a subsequent simplification.

This paper contributes in two ways. First, we develop concept for efficient involutions that are based on bit operations and may potentially benefit from

instruction sets of modern processors or graphics cards. Second, we design an implicit encoding scheme for *unbounded* n-Gmaps which is membrane-centric (rather than pixel-centric) and which utilizes the new involutions for efficient iterations within membranes.

2 Definitions

Definition 1 (Involution). *Involution (or involutary permutation) on a finite set $\mathcal{D}$ is a permutation $\alpha : \mathcal{D} \to \mathcal{D}$ such that $\forall d \in \mathcal{D} : \alpha(\alpha(d)) = d$.*

In this work we aim at highly efficient involutions. Lemma 1 states that, for special sets of integers, they can be achieved by bit flipping.

Lemma 1 (Bit-flip involution). *Let $b \in \mathbb{N}$, $\mathbb{Z}_{2^b} = \{0, \dots 2^b - 1\}$ be a set of first 2^b non-negative integers representable by b bits, $d \in \mathbb{Z}_{2^b}$, and let $0 \leq i, j < b$. Let ϕ_i flip the i-th bit of d: $\phi_i(d) = d \oplus 2^i$, where $\oplus$ denotes the bitwise* XOR. *Then:*

- *ϕ_i is an involution on $\mathbb{Z}_{2^b}$, and*
- *$\phi_i \circ \phi_j$ is an involution on $\mathbb{Z}_{2^b}$.*

Proof. It is easy to see that flipping a bit (two different bits) within $\mathbb{Z}_{2^b}$ is a bijection, hence a permutation. To prove the involutionary property it is helpful to realize that flipping a bit (two different bits) twice does not change the number. Formally, we resort to the associativity of the XOR ($\oplus$):
$\phi_i(\phi_i(d)) = (d \oplus 2^i) \oplus 2^i = d \oplus (2^i \oplus 2^i) = d \oplus 0 = d$, and similarly
$\phi_i(\phi_j(\phi_i(\phi_j(d)))) = (((d \oplus 2^j) \oplus 2^i) \oplus 2^j) \oplus 2^i = d \oplus ((2^j \oplus 2^i) \oplus (2^j \oplus 2^i)) = d \oplus 0 = d$. □

This work is centered around representations of images by means of n-Gmaps [3]:

Definition 2 (n-Gmap). *An n-dimensional* generalized map *(n-Gmap) with $n \in \mathbb{N}_0$, is an $(n + 2)$-tuple $G = (\mathcal{D}, \alpha_0, \dots, \alpha_n)$ where:*

- *$\mathcal{D}$ is a finite set (of* darts *d);*
- *$\forall i \in \{0, \dots, n\} : \alpha_i$ is an involution on $\mathcal{D}$;*
- *$\forall i \in \{0, \dots, n-2\}, \forall j \in \{i+2, \dots, n\} : \alpha_i \circ \alpha_j$ is an involution.*

Recognition algorithms centered around images often incorporate the surrounding background region, i.e., an infinite n-cell. In this paper we refer to the corresponding n-Gmaps as *unbounded* and focus on their representation.

Figure 1 shows an example of an unbounded 2-Gmap representing a 2D image. Here, the apparent visual difference to *bounded* maps (w/o the background) is the extra layer of darts wrapped around the image which bounds the background 2-cell. The extra layer of darts violates the otherwise pixel-centric regularity of *bounded* maps and complicates the design of implicit representation of *unbounded* maps.

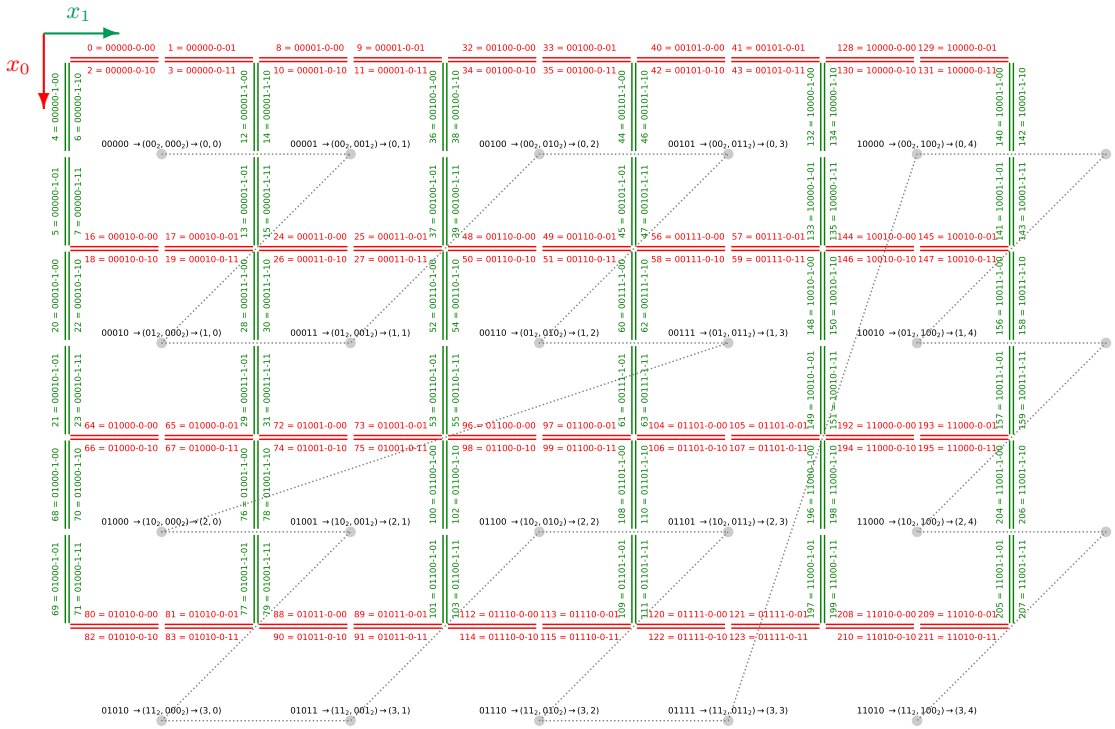

Fig. 1. Example of dart numbering for an unbounded 2-Gmap representing a 3×5 image, where pixels follow the z-curve order (dotted line connecting points). Binary numbers along and across edges demonstrate the flip-involutions ϕ_0 and ϕ_2 Binary numbers next to anchor points show conversion between the z-code and coordinates. Best viewed magnified and in color.

In a search for implicit representations we propose a membrane-centric (as opposed to pixel-centric) approach. Such a membrane-centric approach may additionally become beneficial in applications based on connected component labeling [1] or irregular n-D image pyramids [9] where membranes are iteratively removed and simplified.

2.1 Membranes

In many applications $(n-1)$-dimensional manifolds are considered as the interfaces of "exchange" between two adjacent n-dimensional cells [5]. These manifolds will be referred to as *membranes* for short. In the context of n-Gmaps, membranes are $(n-1)$-*cells* and are defined with the help of orbits [3]:

Definition 3 (Orbit). *Let $G = (\mathcal{D}, \alpha_0, \ldots, \alpha_n)$ be an n-Gmap. Let $S \subseteq \{\alpha_0, \ldots, \alpha_n\}$. The orbit $\langle S \rangle(d)$ of a dart $d \in \mathcal{D}$ is the set of all darts which can be reached, starting from d, by applying any composition of (inverses of) permutations of S.*

Definition 4 (Membrane). *Let* $1 < n \in \mathbb{N}$, $G = (\mathcal{D}, \alpha_0, \ldots, \alpha_n)$ *be an n-Gmap and* $d \in \mathcal{D}$. *A membrane associated with dart* d *is an orbit* $M(d) = \langle \alpha_0, \ldots, \alpha_{n-2}, \alpha_n \rangle(d)$.

In the context of images represented by unbounded n-Gmaps, all membranes are $(n-1)$-hypercubes and it can be easily shown they consists of $|M| = 2^n(n-1)!$ darts. Membranes for 2D (resp. 3D) images are displayed in Fig. 1 (resp. Fig. 2) as differently coloured groups of 4 (resp. 16) darts.

In this paper we consider the *membranes* (rather than the n-dimensional pixels) to be the basic building blocks of the *unbounded* n-Gmap. Involution α_{n-1} will be used to sew the membranes together.

3 Membrane-centric dart numbering

While n-Gmap's set of darts $\mathcal{D}$ can be any finite set of abstract objects we aim at non-negative integer representation of darts. The main motivation for such a representation is the utilizations of the introduced bit-flip involutions, as detailed in section 4.

We first design each dart as an $(n+2)$ tuple $(x_0, \ldots, x_{n-1}, a, i)$ comprised of n voxel coordinates $\vec{x}$, axis index a, and in-membrane index i. Later we show how to binary-encode such tuples to a subset of non-negative integers $\mathcal{D} \subset \mathbb{N}_0$.

3.1 Darts as tuples

Each membrane is orthogonal to one of the n coordinate axes $a \in \{0 \ldots n-1\}$ and, in an infinitely large lattice, would be incident to two n-cells. From these two, we associate the membrane with the *dominating* n-cell, i.e, cell that has one of its coordinates larger by one. We will refer to the coordinates $\vec{x} = (x_0, \ldots, x_{n-1})$ of this cell as the *anchor* coordinates of the membrane.

This way n membranes will be anchored to every pixel center whereas only single membrane will be anchored to out-of the image pixels bounding the scene from the "dominant" side. This is illustrated in Figure 1 for $n = 2$. Here, two membranes are anchored to each of the 3×5 pixels whereas one membrane is anchored to each of $3 + 5$ out-of-scene pixels found to the right and bottom of the image.

The tuple $(x_0, \ldots, x_{n-1}, a)$ can be thought of as the unique membrane identifier. To identify darts withing membranes, this tuple can be naturally extended by an in-membrane index $i \in \{0, \ldots, |M| - 1\}$.

In summary, the $(n+2)$ tuples $\vec{d} = (x_0, \ldots, x_{n-1}, a, i)$ representing darts are comprised of the following components:

1. n anchor coordinates $\vec{x} = (x_0, \ldots, x_{n-1})$ of the membrane,
2. axis number $a \in \{0, \ldots, n-1\}$ the membrane is orthogonal to, and
3. in-membrane index $i \in \{0, \ldots, |M| - 1\} = \{0, \ldots, 2^n(n-1)! - 1\}$.

3.2 Darts as binary codes

There are numerous ways to bijectively[1] map integer tuples $\vec{d} \in \mathbb{N}_0^{n+2}$ to non-negative integers $d \in \mathcal{D} \subset \mathbb{N}_0$. To do so using binary codes, it is sufficient to reserve a fixed, sufficiently wide block of bits for each component of the tuple. Darts of n-Gmaps representing an $(m_0 \times m_1 \times \ldots \times m_{n-1})$ image require n blocks to encode the n anchors, demanding respectively $\lceil \log_2(m_i + 1) \rceil$ bits[2], followed by one $\lceil \log_2 n \rceil$-bit block to encode the axis index, and one block of $\lceil \log_2(2^n(n-1)!) \rceil = n + \lceil \log_2(n-1)! \rceil$ bits to encode the $2^n(n-1)!$ possible in-membrane indices i. Table 1 shows examples of how many trailing bits are needed for a and i for dimensions 2 to 6.

As an example consider a 2-Gmap representing a 3×5 image. The four blocks of $\vec{d}$ need 2, 3, 1, and 2 bits, respectively. 3^{rd} dart of 1^{st} membrane of pixel (0,2) is represented by a 4-tuple $\vec{d} = (0, 2, 1, 3) = (00_2, 010_2, 1_2, 11_2)$ which after concatenation yields dart code $d = 00\,010\,1\,11_2 = 23$. While the

Table 1. How many trailing bits are needed?

n	dimensions	2	3	4	5	6
$\lceil \log_2 n \rceil$	bits for axis index a	1	2	2	3	3
$2^n(n-1)!$	$\lvert M \rvert$ darts in each membrane	4	16	96	768	7680
$\lceil \log_2(2^n(n-1)!) \rceil$	bits for in-membrane index i	2	4	7	10	13

concept introduced above can be efficiently implemented by bit concatenation it inevitably leads to a linear (e.g., column-by-column, row-by-row) ordering of membranes (and thus of its darts). This may pose challenges in distributed processing of huge images [2] as the different membranes of one voxel may get largely distinct numbers. To overcome this, and to gain a better locality we resort to space-filling curves. From a catalogue of choices (Hilbert, Pean, Morton [8]) we choose the latests one, a.k.a the *z-order* curve [7], see Figure 1. The reason for this curve ist that its indices can be computed implicitly (rather than recursively): it is sufficient to replace bit-concatenation of the anchors by bit-interleaving:

2D Example Encoding the 3^{rd} dart of 1^{st}-axis membrane of pixel (0,2) = (00_2, 010_2) in a 2-Gmap following the z-ordering of pixels (cf. Fig. 1) yields:

$$\vec{d} = (0, 2, 1, 3) \qquad d = \big(\overbrace{\underbrace{00}_{x_0}\underbrace{10}_{x_1}\underbrace{0}_{a}}^{\text{membrane id}}\underbrace{11}_{i} \big)_2 = 39 \tag{1}$$

[1] To express the two representations of a dart are equivalent we put $d \equiv \vec{d}$.
[2] $(m_i + 1)$ is needed because of the extra dominant-side, out-of-image anchors.

3D example Encoding the 14^{th} dart of 2^{nd}-axis membrane of voxel (7,5,3) = (111_2,101_2,011_2) in a 3-Gmap following the z-ordering of voxels yields:

$$\vec{d} = (7,5,3,2,14) \qquad d = \big(\overbrace{\underbrace{110101111}_{(x_0\,x_1\,x_2)}\,\underbrace{10}_{a}}^{\text{membrane id}}\,\underbrace{1101}_{i}\big)_2 = 27629 \tag{2}$$

Without loss of generality we assume two routines, `encode_anchor()` and `decode_anchor()`, that bring the anchor coordinates to the corresponding bit positions and back. Both the linear and the z-code alternatives can be implemented using bit shifts and masking or, on modern processors, by leveraging the `pdep` and `pext` instructions [8].

4 Involutions

In this section our aim is to show (1) how the membrane involutions can be efficiently performed by bit operations and (2) how the membrane-sewing involution can be efficiently performed by means of lookup tables.

4.1 Membrane-internal involutions $\alpha_0, \ldots, \alpha_{n-2}, \alpha_n$

All membrane involutions $\alpha_0, \ldots, \alpha_{n-2}, \alpha_n$ are restricted to membrane darts of a fixed anchor $\vec{x}$, fixed axis a, and index $i \in \{0, \ldots, |M|-1\}$.

Since membranes in 2D (3D) images consist of $|M| = 4 = 2^2$ ($|M| = 16 = 2^4$) darts (cf. Tab. 1), $i \in \mathbb{Z}_4$ ($i \in \mathbb{Z}_{16}$), and we may exploit the bit-flip involutions ϕ of Lemma 1 in the design of membranes.

2D images : Referring to Fig. 1 one possibility is to put:

$$\alpha_0(d) \equiv \alpha_0(\vec{d}) = \alpha_0((\vec{x},a,i)) = (\vec{x},a,\phi_0(i)) = (\vec{x},a,i \oplus 01_2) \equiv d \oplus 01_2 \tag{3}$$

$$\alpha_2(d) \equiv \alpha_2(\vec{d}) = \alpha_2((\vec{x},a,i)) = (\vec{x},a,\phi_1(i)) = (\vec{x},a,i \oplus 10_2) \equiv d \oplus 10_2 \tag{4}$$

The third requirement of Definition 2 for 2-Gmaps, namely that $\alpha_0 \circ \alpha_2$ is also an involution follows from Lemma 1.

3D images : Similar to 2D, we choose ϕ_0 for α_0 and the leftmost flip ϕ_3 for α_n. Letting ϕ_1 for α_1 would not, however, result in 2-hypercubes. Instead we need to find a different (optimally bitwise[3]) involution α_1 to sew the 1-cells (Fig.2):

$$\alpha_0(d) = \phi_0(d) = d \oplus 0001_2 \tag{5}$$

$$\alpha_1(d) = d \oplus 0111_2 \oplus (d \ll 1 \,\&\, 0100_2) \oplus (d \ll 2 \,\&\, 0100_2) \tag{6}$$

$$\alpha_3(d) = \phi_3(d) = d \oplus 1000_2 \tag{7}$$

Again, the requirements of Definition 2 for 3-Gmaps must be verified. $\alpha_0 \circ \alpha_3 = \phi_0 \circ \phi_3$ is an involution due to Lemma 1. Verification that both α_1 and $\alpha_1 \circ \alpha_3$ are involutions on $\mathbb{Z}_{16}$ can be done by enumeration $\forall i \in \mathbb{Z}_{16} = \{0 \ldots 15\}$ and by applying mechanism analogous to Equations (3) and (4).

[3] "&" stands for "bitwise AND"; "$\ll$" stands for "left bit-shift".

4.2 Membrane-sewing involution α_{n-1}

In terms of mutual orientation there are many possible ways to embed the membranes on the n-dimensional lattice while they are sewn. A natural restriction is to demand axis-wise consistency, meaning parallel membranes are oriented in the same way. What remains is a specification of $(n-1)$ α_{n-1} sews to determine how the n orthogonal membranes anchored at the origin $(0,0,...,0)$ are mutually α_{n-1}-sewn. This choice, which is a free parameter, then naturally propagates across the whole image and determines all remaining membrane sews α_{n-1}. In this paper our choice for 2D images is determined by $\alpha_1(0,0,0,0) = (0,0,1,0)$ and for the 3D images (cf. Fig. 2, left) by $\alpha_2(0,0,0,0,0) = (0,0,0,1,7)$, $\alpha_2(0,0,0,1,0) = (0,0,0,2,7)$.

In contrast to membranes where the orbit-involutions only change the in-membrane index i (and thus only the rightmost bits), the membrane-sewing involutions α_{n-1} become more involved: they additionally may change the axis index a (when sewing two orthogonal membranes) and may decrement/increment anchor coordinates x (when sewing two membranes of different anchors).

Furthermore it turns out that three sewing scenarios $s \in \{0,1,2\}$ are needed in unbounded n-Gmaps to distinguish whether two α_{n-1}-sewn darts belong to one of the many bounded n-cells ($s = 2$, cf. Fig.2, right) or to the only infinite (background) n-cell. Within the background cell it is furthermore necessary to distinguish whether the two sewn membranes are parallel ($s = 1$, Fig.2, middle) or orthogonal ($s = 0$, Fig.2, left).

Sewing scenarios : For dart $\vec{d} = (x_0, \ldots, x_{n-1}, a, i)$ of a fixed axis a and index i two nested anchor tests are needed to determine if it belongs to the background cell and, if so, whether the sew is parallel or orthogonal. The first test compares the a-th anchor coordinate x_a to the bound 0 (resp. m_a). The second test compares an additional c-th coordinate ($c \neq a$) x_c to 0 (resp. $m_c - 1$). Both c and its corresponding bound m_c are uniquely determined once the size of the image is known, and both the membrane-involutions, and the mutual membrane orientations are fixed.

An efficient branchless (w/o if-then-else statements) pattern-matching to determine s can be implemented using one coordinate-wise XOR followed by two AND masking operations. Such masks can be precomputed and stored in lookup table `LUT_XOR` and two lookup tables `LUT_AND` all indexed by $[a, i]$.

As an example, consider the bottom-right vertical darts $(*, *, 1_2, 11_2)$ in Figure 1 representing an $m_0 \times m_1 = 3 \times 5$ image. Scenario ($s = 0$) must match $(2, 5, 1_2, 11_2) = (m_0 - 1, m_1, 1_2, 11_2)$. If this fails the edge scenario ($s = 1$) is tested by matching $(*, 5, 1_2, 11_2) = (*, m_1, 1_2, 11_2)$. If this fails, too, the dart is α_1-sewn in the interior scenario $s = 2$.

α_{n-1} lookup tables Once the mutual orientations are fixed we pre-compute, for efficiency reasons, lookup tables indexed by $[a, i, s]$ that store, for each axis a, in-membrane index i, and scenario s, $(n+2)$ entries to determine how α_{n-1}

updates dart's anchor by n increments Δx_a and specifies both the new axis a^* and the new in membrane index i^*. Such lookup tables are essentially tensors of size $n \times |M| \times 3 \times (n+2)$. To show them in this paper (cf. Tables 2 and 3) they are reshaped to 2D tables of sizes $n|M| \times 3(n+2)$.

In the following we detail α_1-LUT for 2-Gmaps and α_2-LUT for 3-Gmaps.

α_1-LUT for 2D images is precomputed in Table 2. For dart d in its tuple notation $\vec{d} = (x_0, x_1, a, i)$, the scenario s is first identified (as explained later) and update entries $(\Delta x_0, \Delta x_1, a^*, i^*)$ are found in α_1-LUT $[a, i, s]$. Then, the membrane sewing α_1 is computed as follows (cf. Algorithm 1, `alpha_1()`):

$$\alpha_1(\vec{d}) = \alpha_1((x_0, x_1, a, i)) = (x_0 + \Delta x_0,\ x_1 + \Delta x_1,\ a^*,\ i^*) \tag{8}$$

To be an involution, $\alpha_1(\alpha_1(\vec{d})) = \vec{d}$ must hold. This can be verified by enumeration for every $a \in \{0, 1\}$, $i \in \{0, 1, 2, 3\}$, and $s \in \{0, 1, 2\}$. Let $\alpha_1(\vec{d}) = \vec{e}$ and the corresponding update entries of $\vec{e}$ are $(\Delta x_0^*, \Delta x_1^*, a^{**}, i^{**})$. Then, it is sufficient to verify that the coordinate updates Δ_{x_a} cancel out $(\Delta x_0 + \Delta x_0^* = 0 = \Delta x_1 + \Delta x_1^*)$ and that $(a^{**}, i^{**}) = (a, i)$.

As an example consider the axis $a = 0$ index $i = 3 = 11_2$ sewn in scenario $s = 0$: $(x_0, x_1, 0, 11_2) \xrightarrow{\alpha_1} (x_0 - 1, x_1 + 1, 1, 11_2) \xrightarrow{\alpha_1} (x_0 - 1 + 1, x_1 + 1 - 1, 0, 11_2) = (x_0, x_1, 0, 11_2)$.

α_2 LUT for 3D images is precomputed in Table 3. For dart d in its tuple notation $\vec{d} = (x_0, x_1, x_2, a, i)$, the scenario s is first identified and update entries $(\Delta x_0, \Delta x_1, \Delta x_2, a^*, i^*)$ are found in α_2-LUT $[a, i, s]$. Then, the membrane sewing α_2 is computed as follows (cf. Algorithm 2, `alpha_2()`):

$$\alpha_2(\vec{d}) = \alpha_2((x_0, x_1, x_2, a, i)) = (x_0 + \Delta x_0,\ x_1 + \Delta x_1,\ x_2 + \Delta x_2,\ a^*,\ i^*) \tag{9}$$

To satisfy the definition of 3-Gmap, both α_2 and $\alpha_0 \circ \alpha_2$ must be involutions.

Referring to Table 3, proof by enumeration for α_2 is an analogy to α_1 in 2D case: for fixed scenario $s \in \{0, 1, 2\}$ we need to verify that coordinate updates Δ_{x_a} cancel out and that $(a^{**}, i^{**}) = (a, i)$.

$\alpha_0 \circ \alpha_2$: Neither scenario index s nor any of the Δx_i is affected by α_0. It is thus sufficient to verify by enumeration $(\forall a \in \{0, 1, 2\},\ \forall i \in \{0, \ldots, 15\})$ that Δx_a under $\alpha_0 \circ \alpha_2$ cancel out, and that $\alpha_0 \circ \alpha_2$ restricted to the rightmost 6 bits is an involution.

As an example consider axis $a = 0 = 00_2$ indices $i = 7 = 0111_2$ sewn in interiors $s = 2$ (cf. Fig. 2, right), with causes increase/decrease of anchor's x_0: $(x_0, x_1, x_2, 00_2, 7 = 0111_2) \xrightarrow{\alpha_2} (x_0 - 1, x_1, x_2, 10_2, 13 = 1101_2) \xrightarrow{\alpha_0} (x_0 - 1, x_1, x_2, 10_2, 12 = 1100_2) \xrightarrow{\alpha_2} (x_0 - 1 + 1, x_1, x_2, 00_2, 6 = 0110_2) \xrightarrow{\alpha_0} (x_0, x_1, x_2, 00_2, 7 = 0111_2)$.

5 Summary and validation

As soon as the sizes (m_0, m_1) (resp. (m_0, m_1, m_2)) of a 2D (resp. 3D) image are known the scenario-mask tables (LUT_XOR, LUT_AND) are precomputed and LUT_A1 (resp. LUT_A2) is loaded from Tab. 2 (resp Tab. 3).

Algorithm 1 (resp. Algorithm 2) summarizes[4] the implicit, membrane-centric, and branch-less involutions for 2D (resp. 3D) images. For dart d, lines 1–3, correspond to Equations (3), (4) (resp. (5), (6), (7)). Line 6 computes a combined index $[a, i]$ to retrieve entries from lookup Tab. 2 (resp. Tab. 3). Lines 9,10 compute the scenario index s. Line 12 first converts d to its anchor $\vec{x}$ and updates it by the corresponding Δ entries. Line 13 converts the anchor back and appends the trailing part, i.e, the new a^* and i^*, yielding $\alpha_{n-1}(d)$, which is returned.

As a proof of concept we have successfully performed two tests for a multitude of 2D and 3D images of random sizes $(m_0, \ldots, m_{n-1})$ restricted to $1 \leq m_a \leq 64$. First test checked if functions `alpha_i` and their necessary compositions are involutions consistent with Definition 2. Second test compared the numbers of i-cells $\forall i \in \{0, ..., n\}$ to the expected ground truths computed from image sizes.[5]

To prove the usefulness of implicit encoding we have successfully run both tests for an additional 512^3 3D image (for which generic approaches such as [4] would require a storage of at least 192 GB) on a computer with only 32 GB of RAM.

Table 2. α_1 lookup table for 2-Gmaps

last 3 bits	$s = 0 : 270°$-sew, BG cell			$s = 1 : 180°$-sew, BG cell			$s = 2 : 90°$-sew, interior cell		
$a\ i$	Δx_0	Δx_1	$a^*\ i^*$	Δx_0	Δx_1	$a^*\ i^*$	Δx_0	Δx_1	$a^*\ i^*$
$0 = 0\text{-}00_2$			$4 = 1\text{-}00_2$	$-$		$1 = 0\text{-}01_2$	$-$		$7 = 1\text{-}11_2$
$1 = 0\text{-}01_2$		$+$	$6 = 1\text{-}10_2$	$+$		$0 = 0\text{-}00_2$	$-$	$+$	$5 = 1\text{-}01_2$
$2 = 0\text{-}10_2$	$-$		$5 = 1\text{-}01_2$	$-$		$3 = 0\text{-}11_2$			$6 = 1\text{-}10_2$
$3 = 0\text{-}11_2$	$-$	$+$	$7 = 1\text{-}11_2$	$+$		$2 = 0\text{-}10_2$		$+$	$4 = 1\text{-}00_2$
$4 = 1\text{-}00_2$			$0 = 0\text{-}00_2$	$-$		$5 = 1\text{-}01_2$		$-$	$3 = 0\text{-}11_2$
$5 = 1\text{-}01_2$	$+$		$2 = 0\text{-}10_2$	$+$		$4 = 1\text{-}00_2$	$+$	$-$	$1 = 0\text{-}01_2$
$6 = 1\text{-}10_2$		$-$	$1 = 0\text{-}01_2$	$-$		$7 = 1\text{-}11_2$			$2 = 0\text{-}10_2$
$7 = 1\text{-}11_2$	$+$	$-$	$3 = 0\text{-}11_2$	$+$		$6 = 1\text{-}10_2$	$+$		$0 = 0\text{-}00_2$

6 Conclusions and future work

We proposed a membrane-centric approach to implicit representation of images by means of *unbounded* n-Gmaps. For 2D and 3D images, we have proposed sets of efficient involutions that are based on bit-flips and additional bit-tricks (for membranes) and on precomputed lookup tables (for membrane sews). This decimates the huge storage requirements needed in generic approaches such as [4, 9].

[4] Python is preferred over pseudocode to demonstrate the usage of bitwise operations. Symbols `&`, `|`, `^`, and `<<` denote binary AND, OR, XOR, and left bit-shifts.

[5] The number of n-cells including the background cell equals to $1 + \prod m_a$.

Table 3. α_2 lookup table for 3-Gmaps

last 6 bits	$s=0:270°$-sew, BG cell				$s=1:180°$-sew, BG cell				$s=2:90°$-sew, interior cell			
a i	Δx_0	Δx_1	Δx_2	a^* i^*	Δx_0	Δx_1	Δx_2	a^* i^*	Δx_0	Δx_1	Δx_2	a^* i^*
$0 = 00\text{-}0000_2$				$23 = 01\text{-}0111_2$		$-$		$5 = 00\text{-}0101_2$	$-$			$26 = 01\text{-}1010_2$
$1 = 00\text{-}0001_2$				$22 = 01\text{-}0110_2$		$-$		$4 = 00\text{-}0100_2$	$-$			$27 = 01\text{-}1011_2$
$2 = 00\text{-}0010_2$			$+$	$40 = 10\text{-}1000_2$			$+$	$7 = 00\text{-}0111_2$	$-$		$+$	$37 = 10\text{-}0101_2$
$3 = 00\text{-}0011_2$			$+$	$41 = 10\text{-}1001_2$			$+$	$6 = 00\text{-}0110_2$	$-$		$+$	$36 = 10\text{-}0100_2$
$4 = 00\text{-}0100_2$		$+$		$30 = 01\text{-}1110_2$		$+$		$1 = 00\text{-}0001_2$	$-$	$+$		$19 = 01\text{-}0011_2$
$5 = 00\text{-}0101_2$		$+$		$31 = 01\text{-}1111_2$		$+$		$0 = 00\text{-}0000_2$	$-$	$+$		$18 = 01\text{-}0010_2$
$6 = 00\text{-}0110_2$				$33 = 10\text{-}0001_2$			$-$	$3 = 00\text{-}0011_2$	$-$			$44 = 10\text{-}1100_2$
$7 = 00\text{-}0111_2$				$32 = 10\text{-}0000_2$			$-$	$2 = 00\text{-}0010_2$	$-$			$45 = 10\text{-}1101_2$
$8 = 00\text{-}1000_2$	$-$			$18 = 01\text{-}0010_2$		$-$		$13 = 00\text{-}1101_2$				$31 = 01\text{-}1111_2$
$9 = 00\text{-}1001_2$	$-$			$19 = 01\text{-}0011_2$		$-$		$12 = 00\text{-}1100_2$				$30 = 01\text{-}1110_2$
$10 = 00\text{-}1010_2$	$-$		$+$	$45 = 10\text{-}1101_2$			$+$	$15 = 00\text{-}1111_2$			$+$	$32 = 10\text{-}0000_2$
$11 = 00\text{-}1011_2$	$-$		$+$	$44 = 10\text{-}1100_2$			$+$	$14 = 00\text{-}1110_2$			$+$	$33 = 10\text{-}0001_2$
$12 = 00\text{-}1100_2$	$-$	$+$		$27 = 01\text{-}1011_2$		$+$		$9 = 00\text{-}1001_2$		$+$		$22 = 01\text{-}0110_2$
$13 = 00\text{-}1101_2$	$-$	$+$		$26 = 01\text{-}1010_2$		$+$		$8 = 00\text{-}1000_2$		$+$		$23 = 01\text{-}0111_2$
$14 = 00\text{-}1110_2$	$-$			$36 = 10\text{-}0100_2$			$-$	$11 = 00\text{-}1011_2$				$41 = 10\text{-}1001_2$
$15 = 00\text{-}1111_2$	$-$			$37 = 10\text{-}0101_2$			$-$	$10 = 00\text{-}1010_2$				$40 = 10\text{-}1000_2$
$16 = 01\text{-}0000_2$				$39 = 10\text{-}0111_2$			$-$	$21 = 01\text{-}0101_2$		$-$		$42 = 10\text{-}1010_2$
$17 = 01\text{-}0001_2$				$38 = 10\text{-}0110_2$			$-$	$20 = 01\text{-}0100_2$		$-$		$43 = 10\text{-}1011_2$
$18 = 01\text{-}0010_2$	$+$			$8 = 00\text{-}1000_2$	$+$			$23 = 01\text{-}0111_2$	$+$	$-$		$5 = 00\text{-}0101_2$
$19 = 01\text{-}0011_2$	$+$			$9 = 00\text{-}1001_2$	$+$			$22 = 01\text{-}0110_2$	$+$	$-$		$4 = 00\text{-}0100_2$
$20 = 01\text{-}0100_2$			$+$	$46 = 10\text{-}1110_2$			$+$	$17 = 01\text{-}0001_2$		$-$	$+$	$35 = 10\text{-}0011_2$
$21 = 01\text{-}0101_2$			$+$	$47 = 10\text{-}1111_2$			$+$	$16 = 01\text{-}0000_2$		$-$	$+$	$34 = 10\text{-}0010_2$
$22 = 01\text{-}0110_2$				$1 = 00\text{-}0001_2$	$-$			$19 = 01\text{-}0011_2$		$-$		$12 = 00\text{-}1100_2$
$23 = 01\text{-}0111_2$				$0 = 00\text{-}0000_2$	$-$			$18 = 01\text{-}0010_2$		$-$		$13 = 00\text{-}1101_2$
$24 = 01\text{-}1000_2$		$-$		$34 = 10\text{-}0010_2$			$-$	$29 = 01\text{-}1101_2$				$47 = 10\text{-}1111_2$
$25 = 01\text{-}1001_2$		$-$		$35 = 10\text{-}0011_2$			$-$	$28 = 01\text{-}1100_2$				$46 = 10\text{-}1110_2$
$26 = 01\text{-}1010_2$	$+$	$-$		$13 = 00\text{-}1101_2$	$+$			$31 = 01\text{-}1111_2$	$+$			$0 = 00\text{-}0000_2$
$27 = 01\text{-}1011_2$	$+$	$-$		$12 = 00\text{-}1100_2$	$+$			$30 = 01\text{-}1110_2$	$+$			$1 = 00\text{-}0001_2$
$28 = 01\text{-}1100_2$		$-$	$+$	$43 = 10\text{-}1011_2$			$+$	$25 = 01\text{-}1001_2$			$+$	$38 = 10\text{-}0110_2$
$29 = 01\text{-}1101_2$		$-$	$+$	$42 = 10\text{-}1010_2$			$+$	$24 = 01\text{-}1000_2$			$+$	$39 = 10\text{-}0111_2$
$30 = 01\text{-}1110_2$		$-$		$4 = 00\text{-}0100_2$	$-$			$27 = 01\text{-}1011_2$				$9 = 00\text{-}1001_2$
$31 = 01\text{-}1111_2$		$-$		$5 = 00\text{-}0101_2$	$-$			$26 = 01\text{-}1010_2$				$8 = 00\text{-}1000_2$
$32 = 10\text{-}0000_2$				$7 = 00\text{-}0111_2$	$-$			$37 = 10\text{-}0101_2$			$-$	$10 = 00\text{-}1010_2$
$33 = 10\text{-}0001_2$				$6 = 00\text{-}0110_2$	$-$			$36 = 10\text{-}0100_2$			$-$	$11 = 00\text{-}1011_2$
$34 = 10\text{-}0010_2$		$+$		$24 = 01\text{-}1000_2$		$+$		$39 = 10\text{-}0111_2$		$+$	$-$	$21 = 01\text{-}0101_2$
$35 = 10\text{-}0011_2$		$+$		$25 = 01\text{-}1001_2$		$+$		$38 = 10\text{-}0110_2$		$+$	$-$	$20 = 01\text{-}0100_2$
$36 = 10\text{-}0100_2$	$+$			$14 = 00\text{-}1110_2$	$+$			$33 = 10\text{-}0001_2$	$+$		$-$	$3 = 00\text{-}0011_2$
$37 = 10\text{-}0101_2$	$+$			$15 = 00\text{-}1111_2$	$+$			$32 = 10\text{-}0000_2$	$+$		$-$	$2 = 00\text{-}0010_2$
$38 = 10\text{-}0110_2$				$17 = 01\text{-}0001_2$		$-$		$35 = 10\text{-}0011_2$			$-$	$28 = 01\text{-}1100_2$
$39 = 10\text{-}0111_2$				$16 = 01\text{-}0000_2$		$-$		$34 = 10\text{-}0010_2$			$-$	$29 = 01\text{-}1101_2$
$40 = 10\text{-}1000_2$			$-$	$2 = 00\text{-}0010_2$	$-$			$45 = 10\text{-}1101_2$				$15 = 00\text{-}1111_2$
$41 = 10\text{-}1001_2$			$-$	$3 = 00\text{-}0011_2$	$-$			$44 = 10\text{-}1100_2$				$14 = 00\text{-}1110_2$
$42 = 10\text{-}1010_2$		$+$	$-$	$29 = 01\text{-}1101_2$		$+$		$47 = 10\text{-}1111_2$		$+$		$16 = 01\text{-}0000_2$
$43 = 10\text{-}1011_2$		$+$	$-$	$28 = 01\text{-}1100_2$		$+$		$46 = 10\text{-}1110_2$		$+$		$17 = 01\text{-}0001_2$
$44 = 10\text{-}1100_2$	$+$		$-$	$11 = 00\text{-}1011_2$	$+$			$41 = 10\text{-}1001_2$	$+$			$6 = 00\text{-}0110_2$
$45 = 10\text{-}1101_2$	$+$		$-$	$10 = 00\text{-}1010_2$	$+$			$40 = 10\text{-}1000_2$	$+$			$7 = 00\text{-}0111_2$
$46 = 10\text{-}1110_2$			$-$	$20 = 01\text{-}0100_2$		$-$		$43 = 10\text{-}1011_2$				$25 = 01\text{-}1001_2$
$47 = 10\text{-}1111_2$			$-$	$21 = 01\text{-}0101_2$		$-$		$42 = 10\text{-}1010_2$				$24 = 01\text{-}1000_2$

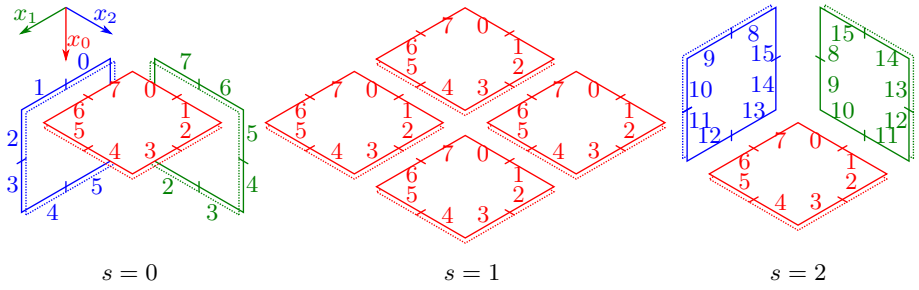

Fig. 2. Sewing scenarios for 3-Gmaps: α_2 maps darts $(x_0, x_1, x_2, 0, 7)$ of axis $a = 0$ and membrane index $i = 7$ differently in three sewing scenarios s to $(x_0, x_1, x_2, 2, 0)$, $(x_0, x_1, x_2{-}1, 0, 2)$, and $(x_0{-}1, x_1, x_2, 2, 13)$, respectively. Compare to line 7 of Tab 3.

Algorithm 1 1D Membrane (α_0, α_2), and membrane-sewing (α_1) involutions

```
1  def alpha_0 (d): return d ^ 0b01
2  def alpha_2 (d): return d ^ 0b10
3
4  def alpha_1 (d):
5      # mask out the axis a and the in-membrane index i. ai = 0...7
6      ai =  d & 0b111
7
8      # scenario index
9      s  = (d ^ LUT_XOR_2D [ai] & LUT_AND_2D [ai,0]) > 0
10     s += (d ^ LUT_XOR_2D [ai] & LUT_AND_2D [ai,1]) > 0
11
12     x0,x1 = decode_anchor_2D (d)     + LUT_A1 [ai,s,:2] # coords update
13     return  encode_anchor_2D (x0,x1) | LUT_A1 [ai,s, 2] # encode + new a,i
```

Algorithm 2 2D Membrane $(\alpha_0, \alpha_1, \alpha_3)$, and membrane-sewing (α_2) involutions

```
1  def alpha_0 (d): return d ^ 0b0001
2  def alpha_1 (d): return d ^ 0b0111 ^ (d << 1 & 0b0100) ^ (d << 2 & 0b0100)
3  def alpha_3 (d): return d ^ 0b1000
4  def alpha_2 (d):
5      # mask out the axis a and the in-membrane index i. ai = 0...47
6      ai =  d & 0b111111
7
8      # scenario index
9      s  = (d ^ LUT_XOR_3D [ai] & LUT_AND_3D [ai,0]) > 0
10     s += (d ^ LUT_XOR_3D [ai] & LUT_AND_3D [ai,1]) > 0
11
12     x0,x1,x2 = decode_anchor_3D (d)        + LUT_A2 [ai,s,:3] # coords update
13     return     encode_anchor_3D (x0,x1,x2) | LUT_A2 [ai,s, 3] # encode + new a,i
```

The membrane centric concept can naturally be extended to dimensions beyond 3D. For an implementation, however, we will be confronted with several currently unresolved issues. First, both in-membrane numbering and membrane-sewing in this work is based on an ad-hoc design. Such an approach quickly finds its limits in higher dimensions and we'll need to resort to a more principled organization of darts in the $(n-1)$-d membranes as well as to more principled membrane sews. To this end we intend to find inspiration in Cartesian products of n-Gmaps [6]. Also, for $n > 3$, the number of membrane darts $|M|$ is not a power of two (cf. Tab. 1) and Lemma 1 cannot be applied out of the box: flipping the left-most membrane bit (as done in this paper for $n = 2$ and $n = 3$) would break the permutation. We'll need to carefully generalize the Lemma to identify which bits in higher dimensional membranes can be securely flipped. Finally it is currently unclear if bit tricks like the one used to define α_1 in 3D can be easily designed in higher dimensions or whether their replacement by additional lookup tables would be a better alternative.

One result of this work is a set of algorithms which are currently in their proof-of-concept state. For a fair time-performance comparisons with generic implementation [4] we plan a C++ port.

It will be of interest to see how concepts introduced in this paper can be applied when removing the membranes during connected component labeling or construction of image pyramids. To maintain the dart set for instance, one bit of information is required to mark absence/presence of all darts of one membrane. We will also need to study which data structures are appropriate to maintain membrane sews α_{n-1} wherever membranes are removed.

References

1. M. Banaeyan, D. Batavia, and W. G. Kropatsch. Removing redundancies in binary images. In *International Conference on Intelligent Systems and Patterns Recognition*, 2022 (to appear).
2. G. Damiand, A. Gonzalez-Lorenzo, F. Zara, and F. Dupont. Distributed combinatorial maps for parallel mesh processing. *Algorithms*, 11(7), August 2018.
3. G. Damiand and P. Lienhardt. *Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing.* A K Peters/CRC Press, 2014.
4. G. Damiand and M. Teillaud. A generic implementation of dD combinatorial maps in CGAL. In *International Meshing Roundtable*, volume 82, pages 46–58, 2014.
5. B. Lane. *Cell Complexes: The Structure of Space and the Mathematics of Modularity.* PhD thesis, University of Calgary, 2015.
6. P. Lienhardt, X. Skapin, and A. Bergey. Cartesian product of simplicial and cellular structures. *International Journal of Computational Geometry & Applications*, 14(03):115–159, 2004.
7. G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing.* International Business Machines Company New York, 1966.
8. M. Perdacher, C. Plant, and C. Böhm. Improved data locality using Morton-order curve on the example of LU decomposition. In *IEEE International Conference on Big Data*, pages 351–360, 2020.
9. F. Torres and W. G. Kropatsch. Canonical encoding of the combinatorial pyramid. In *Proceedings of the Computer Vision Winter Workshop*, pages 118–125, 2014.