Transparent management of adjacencies in the cubic grid

POST-PRINT

Article published on Lecture Notes in Computer Science, volume 13256
https://link.springer.com/chapter/10.1007/978-3-031-04881-4_23

Paola Magillo
University of Genova,
Department of Computer Science, Bioengineering, Robotics,
and Systems Engineering, Genova, Italy
magillo@dibris.inige.it

Lidija Čomić
University of Novi Sad, Faculty of Technical Sciences,
Novi Sad, Serbia
comic@uns.ac.rs

Abstract

We propose an integrated data structure which represents, at the same time, an image in the cubic grid and three well-composed images, homotopy equivalent to it with face-, edge- and vertex-adjacency. After providing an algorithm to build the structure, we present examples showing how, thanks to such data structure, image processing algorithms can be written in a transparent way w.r.t. the adjacency type. Applications include rapid prototyping and teaching.

keywords: Well-composed images, cubic grid, BCC grid, FCC grid, data structure

1 Introduction

Given a polyhedral grid (e.g., a cubic grid), we denote as a 3D image a set of black voxels, while other voxels (background) are white. A 3D image is well-composed if its boundary surface is a 2-manifold. Intuitively, this means that, at each point, the surface is locally topologically equivalent to a disc.

In the cubic grid, not all 3D images are well composed, as two black cubes may share just an edge or just a vertex, and no face. Therefore, it is necessary to take into account three possible types of adjacency relations (vertex-, edge- or face- adjacency) for the definition of homotopy-related properties, such as connectedness, cavities or tunnels, and for the design of algorithms.

The process of transforming a 3D image in another one which is well-composed is called repairing. Recently [3, 4], approaches have been proposed to repair a cubic image by transferring it into another polyhedral grid, ensuring that the resulting image is homotopy equivalent to the original one, with respect to the chosen type of adjacency relation, among the three possible ones. The resulting images are defined in polyhedral grids derived from the cubic one, namely the Body Centered Cubic (BCC) grid and the Face Centered Cubic (FCC) grid.

These grids have been effectively used as viable alternatives to the cubic grid in computer graphics, rendering and illumination [2, 9, 21], ray tracing and ray casting [12, 13, 14], discrete geometry [5, 6], voxelization [10], reconstruction [7, 18], simulation [20] distance transform [23], fast Fourier transform [24], and a software system for processing and viewing 3D data [8].

We propose a unified data structure which represents simultaneously the given 3D image in the cubic grid and its repaired versions in the BCC and FCC grids. Algorithms operating on any of the repaired images can be written in a simple and transparent way.

This paper is organized as follows. In Section 2, we introduce 3D grids and well-composed images. In Section 3, we describe the proposed unified data structure and in Section 4 a construction algorithm unifying the ones in [3, 4]. In Section 5 we show how our unified data structure allows for writing image processing algorithms that may work with any of the three adjacency types in a transparent manner. Finally, in Section 6, we draw concluding remarks.

2 3D Grids and Well-composed Images

In general, we can define a 3D grid as any partition of the 3D space into convex polyhedral cells, called voxels. The voxels of a grid naturally define their faces (polygons where two voxels meet), edges (segments where three or more faces meet), and vertices (points where three or more edges meet). Voxels, faces, edges and vertices constitute a cell complex associated with the grid. Adjacency and incidence relations in the grid are then defined.

Within a grid, an image I is a finite set of voxels. Conventionally, we call black the voxels belonging to I and white the voxels belonging to its complement I^c (background). An image I is (continuously) well-composed [17] if the boundary surface of I, i.e., the surface made up of faces that are incident with exactly one black voxel, is a 2-manifold. Recall that a 2-manifold is a topological space in which each point has a neighborhood homeomorphic to the open unit disc.

The cubic grid, made up of unit cubes, can be obtained by placing a seed at each 3D point with integer coordinates, and considering the Voronoi diagram of such seeds. The resulting Voronoi cells are cubes. Two non-disjoint cubes may share exactly one face (and all its edges and vertices), exactly one edge (and its two vertices) and no face, or exactly one vertex and no edge or face. This leads to three types of adjacency relations: face-adjacency, edge-adjacency, and vertex-adjacency, respectively. As a cube has six face-adjacent cubes, eighteen edge-adjacent cubes, and 26 vertex-adjacent cubes, the three adjacency types are also known as 6-adjacency, 18-adjacency, and 26-adjacency, respectively.

An image I in the cubic grid is (digitally) well-composed [17] if and only if there is no occurrence of either critical edges or critical vertices (see Figure 1). A critical vertex v is a vertex having two (six) incident black cubes and six (two) incident white cubes, where the two black (white) cubes share just the vertex v. A critical edge e is an edge having two incident black cubes and two incident white cubes, where the two black (white) cubes share just the edge e. The two characterizations of well-composed images (continuous and digital) in the cubic grid are equivalent [1].

At critical edges and vertices, the image has different topology if we consider it with 6-, 18-, or 26–adjacency. Therefore, algorithms on general cubic images need to take all the possible critical configurations into account,

Figure 1: Two black cubes sharing a critical vertex and a critical edge. The configurations with exchanged cube colors are also critical.

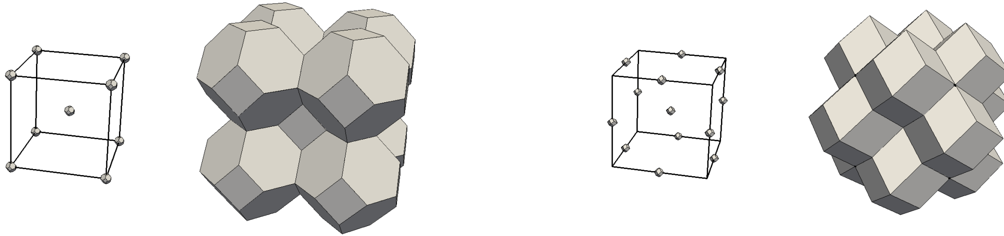

Figure 2: Placement of seeds and growing process of seeds to create the BCC grid (left) and the FCC grid (right).

with a different treatment depending on the considered adjacency type. Moreover, for consistency reasons, an image I and its complement I^c need to be considered with different types of adjacency relations, for example 6-adjacency for I and 26-adjacency for I^c.

The Body Centered Cubic (BCC) grid is obtained from the cubic grid by placing a seed at the center of each cubic voxel, and a seed at each cube vertex. In the Voronoi diagram of such seeds, the Voronoi cells are truncated octahedra (see Figure 2), and these are the voxels of the BCC grid. The BCC grid has just one type of adjacency, i.e., face-adjacency, and any 3D image in such grid is well-composed. In fact, if two truncated octahedra share a vertex, they must share a face as well.

The Face Centered Cubic (FCC) grid is obtained from the cubic grid by placing a seed at the center of each cubic voxel, and a seed at the midpoint of each edge. In the Voronoi diagram of such seeds, the Voronoi cells are rhombic dodecahedra (see Figure 2), and these are the voxels of the FCC grid. Not all images in the FCC grid are well-composed, as pairs of rhombic dodecahedra, sharing just a vertex and no face or edge, exist. On the other hand, two rhombic dodecahedra cannot share an edge, without sharing a face as well.

Note that a subset of voxels in both the BCC and the FCC grid corresponds to the voxels of the cubic grid. Other voxels of the BCC grid correspond to the vertices of the cubic voxels, while other voxels of the FCC grid correspond to the edges of the cubic voxels.

In [3] (respectively, [4]), an algorithm has been proposed to transform an image I in the cubic grid into a well-composed image in the BCC (FCC) grid, which is homotopy equivalent to I according to 6- or 26-adjacency (18-adjacency). The BCC voxels (FCC voxels) corresponding to the cubic voxels maintain the same color they had in the cubic grid. All other black BCC (FCC) voxels correspond to some of the vertices (edges) of black cubes. The new data structure proposed here will store the original cubic image and all its repaired versions in the BCC and FCC grids, in a compact way.

3 Data structure

We propose a unified data structure which can mark as black or white the cubic voxels (equivalently, the voxels of the BCC and FCC grid corresponding to them), the FCC voxels corresponding to cube edges, and the BCC voxels corresponding to cube vertices (with two marks, as they may get a different color in the repaired BCC image, equivalent to the cubic one with 26- or 6-adjacency). In this way, we can represent the original 3D image in the cubic grid and, simultaneously, its well-composed versions in the BCC and FCC grids.

Assuming that the cubic grid is made up of unit cubes centered at points with integer Cartesian coordinates, we multiply all coordinates by two, so that we can identify all cubes, faces, edges, vertices, with integer coordinates [16]: cubes have three even coordinates; vertices have three odd coordinates; edges have one even and two odd coordinates (the even coordinate corresponds to the axis the edge is parallel to), and faces have two even and one odd coordinate (the odd coordinate corresponds to the axis the face is orthogonal to).

Each cube has eight vertices, each shared by eight cubes. So, we can associate each cube with one of its vertices, by convention the one with maximum Cartesian coordinates. If (x, y, z) denotes a cube, the associated vertex is $(x + 1, y + 1, z + 1)$. Each cube has twelve edges, each shared by four cubes. So, we can associate each cube with three of its edges, by convention the ones incident with the vertex with maximum Cartesian coordinates. If (x, y, z) denotes a cube, the associated edges are $(x, y+1, z+1)$, $(x+1, y, z+1)$, and $(x+1, y+1, z)$.

In a black and white (binary) image, the color of a cell (cube, edge or vertex) can be stored in just one bit: 1 for black and 0 for white. For representing the color of a cube, the color of the three associated edges, and the two colors of the associated vertex, we need six bits, which fit in one byte. Given a triplet (i, j, k), with $i, j, k \in \mathbb{Z}^3$, the bits of the associated byte contain:

- bit 0 is the color of the cube $b = (2i, 2j, 2k)$, i.e., the color of the BCC and of the FCC voxels corresponding to b;
- bits 1 and 2 are not used;
- bit 3 is the color of the edge $e_x = (2i, 2j + 1, 2k + 1)$ of b, i.e., the color of the FCC voxel corresponding to e_x, in the equivalent well-composed FCC image with 18-adjacency;
- bit 4 is the color of the edge $e_y = (2i + 1, 2j, 2k + 1)$ of b, i.e., the color of the FCC voxel corresponding to e_y, in the equivalent well-composed FCC image with 18-adjacency;
- bit 5 is the color of the edge $e_z = (2i + 1, 2j + 1, 2k)$ of b, i.e., the color of the FCC voxel corresponding to e_z, in the equivalent well-composed FCC image with 18-adjacency;
- bit 6 is the color of the vertex $v = (2i+1, 2j+1, 2k+1)$ of b, i.e., the color of the BCC voxel corresponding to v, in the equivalent well-composed BCC image with 26-adjacency;
- bit 7 is the color of the same vertex (i.e., the BCC voxel) v in the equivalent well-composed BCC image with 6-adjacency.

We have developed our implementation in Python. Python is a high-level language providing the dictionary as a built-in type. The implementation uses a dictionary, where the keys are tuples of three integers, and the values are bytes. In another programming language, we would use a 3D matrix. For such purpose, we would select a subset of $\mathbb{Z}^3$, for example we can consider the bounding box of black voxels in the cubic grid, plus an extra layer of white voxels in all six directions. In addition, we would shift coordinates to ensure that they are non-negative, and can thus be used as matrix indexes.

Given a cell $c = (x, y, z)$, which may be a cube, an edge, or a vertex (or, equivalently, given a voxel in the cubic, FCC, or BCC grids), the dictionary key is obtained by simply dividing the coordinates of c by two with integer division. Figure 3 shows the Python code of the function returning the bit index, inside the byte, storing the color of a cell. The two Python functions for reading and setting the color of a cell (x, y, z) are shown in Figure 4. If not present in the dictionary, a triplet (i, j, k) is considered as having byte 00000000 as value. Therefore elements of the dictionary having zero value do not need to be stored. That is why, in Figure 4, a new element is added only if the value to be set is 1.

From the point of view of the spatial complexity, just one byte is stored for each cube belonging to the bounding box of the object (plus a layer of white cubes around it), and the single bits inside it are used to code the color of the other cells (BCC and FCC voxels corresponding to cube vertices and edges). As one byte is the smallest storage unit, it is not possible to use less memory than this, even for representing just the color of the cubes. Therefore, the storage cost of our unified data structure can be considered as optimal.

4 Integrated image repairing algorithm

The data structure, described in Section 3, can be filled by running the algorithms proposed in [3] and [4], which specify the vertices and edges, respectively, to be set as black in order to have a well-composed image, equivalent to the given one with 6-, 26- and 18–adjacency, depending on the case. In the following, we describe a new integrated algorithm.

We consider a vertex v and two collinear incident edges e_1 and e_2 extending in negative and in positive axis direction from v, respectively (see Figure 5). Referring to such a configuration, we rewrite the rules used in [3] and in [4] to decide whether (the BCC voxel corresponding to) v and (the FCC voxel corresponding to) e_1 must become black. In [3], v becomes black if:

1. edge e_1 has four incident black cubes,
2. edge e_1 is critical;
3. edge e_2 is critical, and edge e_1 does not have four incident white cubes;
4. v is a critical vertex;

Rule 1 also applies for producing an equivalent BCC image to the given one with 6-adjacency. These rules must be applied for all three axes. Rule 4 is applied only when edges e_1 and e_2 are parallel to the x-axis, otherwise we would check it three times. The algorithm in [4] performs in two stages. In the first stage:

- if e_1 is incident with three or four black cubes, then e_1 becomes black;
- if e_1 is critical, then e_1 becomes black;
- in configurations where e_1 and e_2 are both critical, with two pairs of 6-adjacent black cubes, then two more edges e_3 and e_4, incident with v, become black. Edge e_3 is chosen conventionally, depending on the supporting axis of e_1, e_2, and e_4 is chosen in such a way that e_3 and e_4 belong to the same two white cubes (see [4] for details).

As before, these rules must be repeated for all three axes. The second stage examines all faces f shared by two black cubes. If no edge of f is black, or exactly two opposite edges of f are black, then one (more) edge of f becomes black. Such edge is chosen conventionally, depending on the normal axis of f, and on the supporting axis of its two black edges (see [4] for details).

```
def getBitIndex(x,y,z, face_adj=False):
    rx, ry, rz = x%2, y%2, z%2
    if face_adj: return rx + 2*ry + 3*rz + (rx*ry*rz)
    else: return rx + 2*ry + 3*rz
```

Figure 3: Python code of the function returning the bit index for a cell (x, y, z), which may be a cube, an edge or a vertex. The returned value is an integer from 0 to 7. The value of `face_adj` is only relevant if (x, y, z) denotes a vertex, i.e., if all three coordinates are odd. True means 6-adjacency, False means 26-adjacency. The operator % denotes the remainder of integer division.

```
def getCellColor(x,y,z, adj=0):
    ind1 = getMainIndex(x,y,z)
    if not ind1 in colorMatrix.keys(): return 0 # default is white
    ind2 = getBitIndex(x,y,z, adj==6)
    return getBitFrom(ind2, colorMatrix[ind1])


def setCellColor(value, x,y,z, adj=0):
    ind1 = getMainIndex(x,y,z)
    if not ind1 in colorMatrix.keys():
        if value==0: return 0
        else: colorMatrix[ind1] = 0x0
    ind2 = getBitIndex(x,y,z, adj==6)
    colorMatrix[ind1] = setBitInto(value, ind2, colorMatrix[ind1])
    return value
```

Figure 4: Python functions `getCellColor` to return and `setCellColor` to set the color of a cell (x, y, z), which can be a cube, an edge, or a vertex. The color is 1 for black and 0 for white. Parameter `adj` is relevant only if (x, y, z) is a vertex, as a vertex may have a different color with 6-adjacency (`adj==6`) or 26-adjacency (any other value). Variable `colorMatrix` is a dictionary associating triplets with bytes. The function call `getMainIndex(x,y,z)` returns the dictionary key for cell (x, y, z). The function call `getBitFrom(triplet,byte)` returns the bit of the given triplet within the given byte. The function call `setBitInto(value,triplet,byte)` sets to value the bit of given triplet within the given byte, and returns the modified byte.

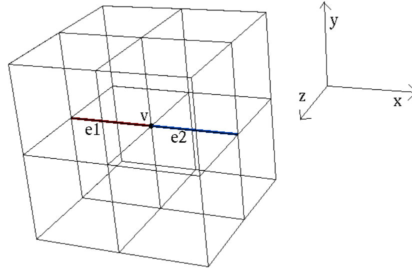

Figure 5: A vertex v and its incident edges e_1, e_2 in negative and positive axis directions, respectively (here for the x-axis).

Our integrated algorithm factorizes the configurations checked by both algorithms. The first stage considers a vertex v and, in turn, the three Cartesian directions. We describe the procedure for the x-parallel direction. Let e_1 and e_2 be the edges incident with v in negative and positive direction with respect to v (as in Figure 5). Based on the configuration at e_1, e_2, we change the color of v, e_1 to black according to the following rules:

1. If e_1 is incident with four white cubes, we skip.
2. If e_1 is incident with four black cubes, then e_1 becomes black and v becomes black (with both 26- and 6-adjacency).
3. If e_1 is incident with three black cubes, then e_1 becomes black.
4. If e_2 is critical (for Rule 1, here e_1 cannot have four incident white cubes), then v becomes black with 26-adjacency (e_2 will become black when processing its second endpoint).
5. If e_1 is critical, then e_1 becomes black, and v becomes black with 26-adjacency.
6. If e_1 and e_2 are both critical with the same configuration, then two more edges e_3 and e_4 become black, as specified in [4].
7. If v is critical, then v becomes black with 26-adjacency. This last condition is only checked for e_1, e_2 parallel to the x-axis (otherwise we would check it three times).

The main algorithm iterates the above described procedure three times, once for each axis, for all vertices (x, y, z) within or on the boundary of the bounding box of the image. Then, it iterates once on the cubes $b = (x, y, z)$, and, only if the cube b is black, executes the second stage of [4] for the three faces of b in the positive axis directions, i.e., the faces $(x + 1, y, z), (x, y + 1, z), (x, y, z + 1)$.

5 Applications

Thanks to the unified data structure, many image processing tasks, which need to consider a 3D image with one of 26-, 18- or 6-adjacency, can be performed in a simple and transparent manner. It is enough to specify the desired adjacency type as a parameter. In this section, we present some examples.

The first example is the extraction of the connected components. In a general cubic image, this operation will give a different result, depending on the considered adjacency type. After the three repaired images have

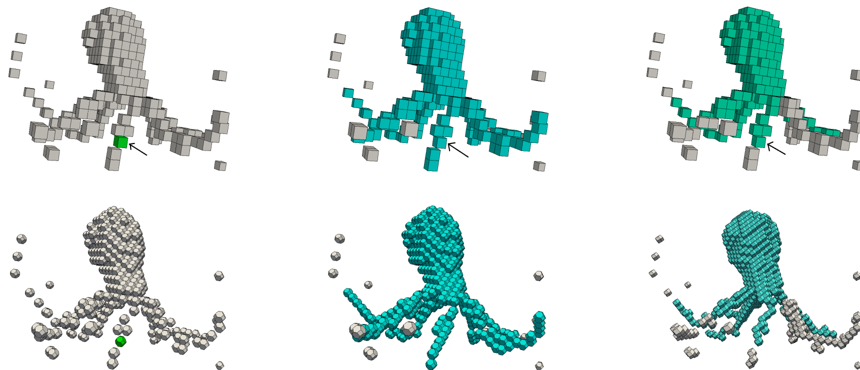

Figure 6: A raw cubic discretization of an octopus, with many critical edges and vertices, and the connected component containing the cube pointed at by the arrow (which has a critical edge and a critical vertex) on the cubic image (upper row) and on its repaired version (lower row), with 6-, 26- and 18-adjacency, from left to right.

been computed (see Section 4) and stored in our unified data structure (see Section 3), it is sufficient to choose which one we want to deal with. Figure 7 shows the Python function for this task. The different treatment of the three cases is hidden inside functions `adjacents` and `getCellColor`. Given a BBC voxel (a FCC voxel) e, `adjacents`(e) provides a list of BCC voxels (FCC voxels) if the adjacency type is 6 or 26 (18). Remember that a tuple representing a cube also represents a FCC and a BCC voxel. Function `getCellColor` returns the black or white color (1 or 0) of a cell, also depending on the adjacency type if the cell is a vertex.

A second example is extracting the boundary surface of a 3D image. In a well composed image, this is a 2-manifold surface consisting of one or more closed surfaces. The Python function shown in Figure 8 collects all polygonal faces composing the boundary. Again, this is done in a transparent way with respect to the adjacency type.

Because the image is well-composed, the boundary surface can also be used to compute the Euler characteristic of the repaired image in the BCC or FCC grid (i.e., that of the original image in the cubic grid, with the corresponding adjacency type). In fact, given a 3-manifold object O, its boundary surface ∂O is a 2-manifold, and the Euler characteristics χ of the two sets are related by property $2\chi(O) = \chi(\partial O)$. The Euler characteristic of a polygonal surface can be computed as $\chi(\partial O) = n_0 - n_1 + n_2$, where n_0, n_1, n_2 denote the number of vertices, edges, and polygonal faces, respectively. A common representation of a polygonal surface is the so-called indexed representation, storing an array of vertices (without duplicates), and each face as a list of indexes within the vertex array. Numbers n_0 and n_2 are stored, and n_1 is easily retrieved by exploiting the fact that, as the surface is a 2-manifold without boundary, an edge is shared by exactly two faces. Therefore $n_1 = \frac{1}{2}\sum_f \mathrm{len}(f)$, where $\mathrm{len}(f)$ denotes the number of edges of a face f.

Another example is the computation of digital distances in an image. Digital distances are used, for example, for computing the medial axis. Given a black cube b, the Euclidean distance of b from the white background is digitally approximated by taking the minimum length of a path of adjacent black cubes connecting b to a white cube. Of course, the digital distance depends on the considered adjacency type. Again, this can be done with our unified data structure in a transparent way. Many other image processing operations [11, 15, 16, 19, 22] might benefit from our unified data structure.

```
def connectedComponents(cube_list, adj_type):
    components = dict() # empty dictionary
    comp_list = [] # empty list
    ind = -1 # index of last found connected component
    for b in cube_list:
      if not b in components.keys():
         # Start a new connected component from b:
         ind += 1
         comp_mark[b] = ind
         comp_list.append([b]) #init comp_list[ind] as [b]
         # Depth first search to find all cells in the connected component:
         stack = [b]
         while len(stack)>0:
            e = stack.pop()
            for d in adjacents(e, adj_type):
              if getCellColor(d, adj_type)==1 and not d in comp_mark.keys():
                 comp_mark[d] = ind
                 stack.append(d)
                 if isCube(d): comp_list[ind].append(d)
    return (ind+1, comp_list, comp_mark)
```

Figure 7: Python code of the function finding the connected components. `cube_list` is a list of black cubes, `adj_type` is 6, 18 or 26. Dictionary `comp_marks` will mark each black cell with the index of the connected component containing it. `comp_list` is a list of lists of cubes, each list `comp_list[i]` will contain the cubes belonging to the i-th connected component. In addition, the function returns the number of connected components. The process applies a standard depth first search using a stack, here a Python list with operations `append` and `pop`.

```
def findBoundary(cell_list, adj_type):
    boundary = Surface()
    for c in cell_list:
      for d in adjacents(c, adj_type):
           # If the adjacent voxel is white, the common face is on the boundary
           if getCellColor(d, adj_type)==0:
              boundary.addFace(commonPolygon(c,d))
    return boundary
```

Figure 8: Python code of the function finding the boundary surface of the repaired image according to a given adjacency type. `cell_list` is a list of black cells, `adj_type` is 6, 18 or 26.

6 Concluding remarks

We have proposed a new compact data structure which allows storing, at the same time, a cubic image and its repaired versions according to vertex-, face- and edge- adjacency. The specific contribution of this work is in the possibility of writing image processing algorithms in a uniform way, without the need for a different treatment of critical configurations in the three adjacency types. With almost no extra space, this simplifies the implementation of algorithms, and can be especially useful at a prototype level, when execution time is not yet an issue. Among possible application fields, we mention the design of new mathematical definitions and tools, and teaching. In the first case, it will be possible to get preliminary results and select the best adjacency type for a more efficient implementation. In teaching, students will be able to write code comparing the impact of the adjacency type on the known image processing operations.

Acknowledgment

This research has been partially supported by the Ministry of Education, Science and Technological Development through project no. 451-03-68/2022-14/ 200156 ”Innovative scientific and artistic research from the FTS (activity) domain”.

References

[1] N. Boutry, T. Géraud, and L. Najman. A Tutorial on Well-Composedness. *Journal of Mathematical Imaging and Vision*, 60(3):443–478, 2018.

[2] V. E Brimkov and R. P. Barneva. Analytical honeycomb geometry for raster and volume graphics. *The Computer Journal*, 48:180–199, 2005.

[3] L. Čomić and P. Magillo. Repairing 3D binary images using the BCC grid with a 4-valued combinatorial coordinate system. *Information Sciences*, 499:47–61, 2019.

[4] L. Čomić and P. Magillo. Repairing 3D binary images using the FCC grid. *Journal of Mathematical Imaging and Vision*, 61:1301–1321, 2019.

[5] L. Čomić and P. Magillo. On Hamiltonian cycles in the FCC grid. *Comput. Graph.*, 89:88–93, 2020.

[6] L. Čomić, R. Zrour, G. Largeteau-Skapin, R. Biswas, and E. Andres. Body Centered Cubic Grid - Coordinate System and Discrete Analytical Plane Definition. In *Discrete Geometry and Mathematical Morphology DGMM*, volume 12708 of *LNCS*, pages 152–163, 2021.

[7] B. Csébfalvi. An evaluation of prefiltered b-spline reconstruction for quasi- interpolation on the body-centered cubic lattice. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):499–512, 2010.

[8] E. S. Linnér and M. Morén and K.-O. Smed and J. Nysjö and R. Strand. LatticeLibrary and BccFccRaycaster: Software for processing and viewing 3D data on optimal sampling lattices. *SoftwareX*, 5:16–24, 2016.

[9] B. Finkbeiner, A. Entezari, D. Van De Ville, and T. Möller. Efficient volume rendering on the body centered cubic lattice using box splines. *Computers & Graphics*, 34(4):409–423, 2010.

[10] L. He, Y. Liu, D. Wang, and J. Yun. A Voxelization Algorithm for 3D Body-Centered Cubic Line Based on Adjunct Parallelepiped Space. In *Information Computing and Applications - Third International Conference, ICICA, Part I*, pages 352–359, 2012.

[11] G. T. Herman. *Geometry of Digital Spaces.* Birkhauser, Boston, 1998.

[12] L. Ibáñez, C. Hamitouche, and C. Roux. Ray-tracing and 3D Objects Representation in the BCC and FCC Grids. In *Discrete Geometry for Computer Imagery, 7th International Workshop, (DGCI)*, pages 235–242, 1997.

[13] L. Ibáñez, C. Hamitouche, and C. Roux. Ray casting in the BCC grid applied to 3D medical image visualization. In *Proceedings of the 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society. Vol.20 Biomedical Engineering Towards the Year 2000 and Beyond (Cat. No.98CH36286)*, volume 2, pages 548–551 vol.2, 1998.

[14] M. Kim. GPU isosurface raycasting of FCC datasets. *Graphical Models*, 75(2):90–101, 2013.

[15] R. Klette and A. Rosenfeld. *Digital geometry. Geometric methods for digital picture analysis, Chapter 2.* Morgan Kaufmann Publishers, San Francisco, Amsterdam, 2004.

[16] V. A. Kovalevsky. *Geometry of Locally Finite Spaces (Computer Agreeable Topology and Algorithms for Computer Imagery), Chapter 3.* Editing House Dr. Bärbel Kovalevski, Berlin, 2008.

[17] L. J. Latecki. 3D Well-Composed Pictures. *CVGIP: Graphical Model and Image Processing*, 59(3):164–172, 1997.

[18] T. Meng, B. Smith, A. Entezari, A. E. Kirkpatrick, D. Weiskopf, L. Kalantari, and T. Möller. On visual quality of optimal 3d sampling and reconstruction. In *Graphics Interface 2007, ACM, New York, NY, USA*, pages 265–272, 2007.

[19] T. Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press, 1982.

[20] K. Petkov, F. Qiu, Z. Fan, A. E. Kaufman, and K. Mueller. Efficient LBM Visual Simulation on Face-Centered Cubic Lattices. *IEEE Trans. Vis. Comput. Graph.*, 15(5):802–814, 2009.

[21] F. Qiu, F. Xu, Z. Fan, N. Neophytou, A. E. Kaufman, and K. Mueller. Lattice-Based Volumetric Global Illumination. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1576–1583, 2007.

[22] A. Rosenfeld and A. C. Kak. *Digital Picture Processing.* Academic Press, London, 1982.

[23] R. Strand. The Euclidean Distance Transform Applied to the FCC and BCC Grids. In *Pattern Recognition and Image Analysis, Second Iberian Conference, IbPRIA*, pages 243–250, 2005.

[24] X. Zheng and F. Gu. Fast Fourier Transform on FCC and BCC Lattices with Outputs on FCC and BCC Lattices Respectively. *Journal of Mathematical Imaging and Vision*, 49(3):530–550, 2014.