Rasterization and voxelization of two- and three-dimensional space partitionings

Gorte, Ben; Zlatanova, Sisi

DOI
10.5194/isprsarchives-XLI-B4-283-2016

Publication date
2016

Document Version
Final published version

Published in
ISPRS Archives

RASTERIZATION AND VOXELIZATION OF TWO- AND THREE-DIMENSIONAL SPACE PARTITIONINGS

Ben Gorte[1], Sisi Zlatanova[2]
[1]Dept. of Geoscience and Remote Sensing, [2]Dept. of Urbanism,
Delft University of Technology, the Netherlands
b.g.h.gorte@tudelft.nl, s.zlatanova@tudelft.nl

Commission IV, WG IV/7

KEY WORDS: Vector-to-raster conversion, voxelization, 3D grid, indoor model

ABSTRACT:

The paper presents a very straightforward and effective algorithm to convert a space partitioning, made up of polyhedral objects, into a 3D block of voxels, which is fully occupied, i.e. in which every voxel has a value. In addition to walls, floors, etc. there are 'air' voxels, which in turn may be distinguished as indoor and outdoor air. The method is a 3D extension of a 2D polygon-to-raster conversion algorithm. The input of the algorithm is a set of non-overlapping, closed polyhedra, which can be nested or touching. The air volume is not necessarily represented explicitly as a polyhedron (it can be treated as 'background', leading to the 'default' voxel value). The approach consists of two stages, the first being object (boundary) based, the second scan-line based. In addition to planar faces, other primitives, such as ellipsoids, can be accommodated in the first stage without affecting the second

1. INTRODUCTION

Three-dimensional grids, where data are represented as values at regularly-spaced grid points (voxels), are drawing attention increasingly, in addition to the more common 3D vector representations. For certain categories of spatial analysis, such as those involving 3D scalar or vector fields (air pollution, noise, wind) that vary continuously over space, gridded data representations offer clear advantages. Such representations has been largely used for modelling of geological structures. It seems natural to have objects like buildings and vegetation, as well as the terrain, represented by voxels too, when they play a role in applications that use field representations. Another application area where grid representations are currently studied is (indoor) navigation, where routes are computed along which persons, robots, or drones are moving through collections of 'free space' or 'air' voxels.

As many 3D GIS (e.g. building) models are in existence and are represented by vector data sets, there is a need for 3D vector-to-raster conversion algorithms that translate vector models into grid representations. To avoid data duplication, but also to serve applications at different spatial resolutions, it is advantageous to perform those conversions 'on the fly'. This calls for efficient conversion routines.

Two general approaches for 3D rasterization can be distinguished: object rasterization and scan-conversion rasterization (Xie, et al 2009). Object rasterization considers only the object of interest and follows two steps: boundary rasterization and interior filling, when needed (e.g. polygon and polyhedron). Some of the first studies of object rasterization are made by (Kaufman & Shimony 1986, Kaufman 1987, Wang and Kaufman 1993). They presented a set of algorithms for voxelizing 3D lines, polygons, polyhedral, cubic parametric curves, bi-cubic surfaces, circles, quadratic objects and tri-cubic solids. The algorithms are developed for a specific connectivity (i.e. 26-connected only), which influences the performance in case of large data sets. This issue has been lately revisited by the same research group (Cohen-or & Kaufman 1997). Huang et al. 1998 presents algorithms for voxelizing planes and surfaces. However, this approach involves many spatial operations such as computation of distances, intersections and constructions of planes, which leads to high complexity and difficulties for implementation. A topological method presented by (Laine 2013) has been seen as one of the best, since it offers a mechanism of voxelizing, which can be applied for all objects and provides options to specify the desired connectivity. An implementation of this approach is reported in Nourian et al 2016.

The above mentioned research is all object-type specific, which works well for non-manifold (watertight) objects such as surfaces and lines. However, a number of different algorithms must be used for the different kinds of input objects.

As opposed to that, a generic solution can be achieved simply by creating a voxel if it is overlaid by a portion of an input object, i.e. intersection-based approach in a given volume. This approach considers continuous rasterization of a given 3D raster box and investigates which voxels get what kind of value. Most of the implementations follow the scan-conversion approach, which is well studied and commonly used in the field of 2D computer graphics. Examples of such libraries are binvox (Min 2016), Voxelization toolkit (Milossramek 2013). Many of the existing approaches make use of bitwise arithmetic to speed up the performance and facilitate the raster conversion of big models (Eisemann, 2008).

On one hand, one could say that 3D vector-to-raster conversion is a complicated, however solved, problem. On the other hand, the “best” algorithms doesn't seem to have crystallized out yet. Moreover, the different algorithms show various degrees of strictness (vs. tolerance) w.r.t. their input formats, for instance

1 Corresponding author

regarding topological constraints. Lastly, the problem is demanding in terms of computational load. GPU-based solutions are present (for example in binvox), but this complicates the methods even further and poses additional hardware constraints.

Long before these developments became actual, vector-to-raster conversion was already an issue in 2D GIS, where, for example, the ILWIS (Gorte et al 1988) system included an approach that was already making use of functionality provided by dedicated PC graphics hardware. Nowadays, standard libraries like GDAL (www.gdal.org) provide excellent software for this purpose.

This paper presents a novel approach to raster-to-vector conversion in 2D (converting a set of closed polygons to a grid of pixels) and in 3D (converting a set of closed polyhedra to a 3D grid of voxels). It is assumed that to each input polygon (or polyhedron) a numerical value is associated, such as an id-number, a class label, a semantic value or any other attribute, which will become the pixel (or voxel) value in the grid to be produced. Polygons/polyhedra should be non-overlapping. In case they do not fill the entire space to-be-gridded, a default (background) value is assumed for the remainder.

The approach is quite tolerant w.r.t. its input. Preferably this should include a complete representation of the topology of vertices, edges, faces and polygons/polyhedral. In practice it will be sufficient that each boundary "knows" the two polygons/polyhedra on either side, and then it is not even important to know which one is at which side. Furthermore it will be acceptable when each polygon/polyhedron in the input dataset is a separate object, having no explicitly stored relationship with the others, and every boundary separates the object from "background" (therefore, boundaries *between* entities appear twice in the dataset).

Moreover, the approach presented below can be easily extended, in order to include other objects than polygons or polyhedra, such as ellipses (2D) and ellipsoids (3D), but also splines, nurbs, etc. as soon as they are watertight.

2. METHODOLOGY

The proposed vector-to-raster conversion method is clearly divided into two stages (Fig. 1). The first stage converts vector polygons (2D) or polyhedra (3D) into an intermediate raster format, which we call a 'sparse boundary grid'. In the second stage this is then be converted into the final 'full pixel/voxel grid', where each pixel/voxel belonging to a certain object has been assigned the value of that object. The second stage can be easily inverted, deriving a 'sparse boundary grid' from a full pixel or full voxel grid.

The explanation of the two stages follows below in the reverse order: Stage one, the conversion between vector polygons and polyhedra to sparse boundary grids, will be introduced in section 2.3. There it will also become clear that such grids can be relatively easily created from other vector input elements, such as sphere, ellipsoids, and 2.5d surfaces.

First, we explain stage two: the conversion between a sparse boundary grid and a full grid, and *vice versa*. These are done by operations called bitwise **xor prefix** resp. **xor infix**, as introduced below.

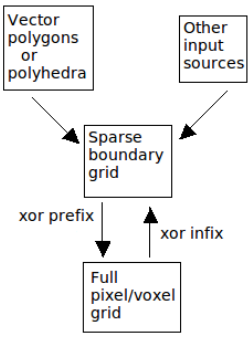


Fig. 1 Flowchart of the method

1.1 Bitwise Exclusive Or (xor)

The vector-to-raster conversion method heavily relies on the bitwise exclusive **or** operator. It is applied to the numerical values of polygons/polyhedra in the input, which appear as pixel/voxel values in the output. The operator, denoted *a* **xor** *b* when applied to values *a* and *b*, is illustrated in Fig. 2. Here the range of values is assumed to be 0-255 (8 bits), but the principle is the same for larger ranges (16, 32 or 64 bits). Bitwise Exclusive Or is a primitive operation (instruction) in the CPU of any computer, and is therefore executed at the highest speed possible.

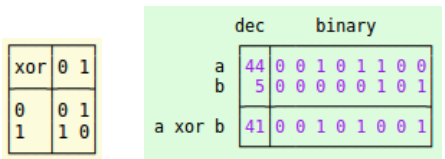


Fig. 2: Bitwise Exclusive OR, left: on a single bit, right: on two arbitrary 8-bit values

The **xor** operator is commutative:

$$a \textbf{ xor } b = b \textbf{ xor } a \qquad (1)$$

and associative:

$$(a \textbf{ xor } b) \textbf{ xor } c = a \textbf{ xor } (b \textbf{ xor } c).$$

Furthermore:

$$a \textbf{ xor } a = 0$$

and:

$$a \textbf{ xor } 0 = a$$

and therefore:

$$(a \textbf{ xor } c) \textbf{ xor } (b \textbf{ xor } c) = a \textbf{ xor } b. \qquad (2)$$

Properties (1) and (2) will be helpful when separately handling two entities sharing a common boundary.

2.2 Prefix and Infix

Given a sequence $A = a_0, a_1, .. a_n$, we define the infix of an operator **O** on A as the result of applying **O** to all pairs of neighbouring elements of A:

$$\textbf{O} \text{ infix } A = a_0 \textbf{ O } a_1, a_1 \textbf{ O } a_2, .. , a_{n-1} \textbf{ O } a_n$$

The prefix of the operator **O** on A is defined as the result of applying **O** on all contiguous sub-sequences $a_0, .. a_i$ $(0 \le i \le n)$.

O prefix $A = (a_0,), (a_0\,\mathbf{O}\,a_1), (a_0\,\mathbf{O}\,a_1\,\mathbf{O}\,a_2), .. , (a_0\,\mathbf{O}\,..\,\mathbf{O}\,a_n)$

After properly defining 'sum', 'box', 'xor', 'infix' and 'prefix' in the programming language J (www.jsoftware.com), we can have a dialogue as shown in Fig.3.

```
   box infix 'Rasterize'
┌──┬──┬──┬──┬──┬──┬──┬──┐
│Ra│as│st│te│er│ri│iz│ze│
└──┴──┴──┴──┴──┴──┴──┴──┘

   box prefix 'Rasterize'
┌─┬──┬───┬────┬─────┬──────┬───────┬────────┬─────────┐
│R│Ra│Ras│Rast│Raste│Raster│Rasteri│Rasteriz│Rasterize│
└─┴──┴───┴────┴─────┴──────┴───────┴────────┴─────────┘

   sum infix 1 2 3 4 5
3 5 7 9

   sum prefix 1 2 3 4 5
1 3 6 10 15

   xor infix 0 0 0 3 3 3 3 3 3 2 2 2 2 1 1 1 0 0 0 0
0 0 3 0 0 0 0 0 1 0 0 0 3 0 0 1 0 0 0

   xor prefix 0 0 3 0 0 0 0 0 1 0 0 0 3 0 0 1 0 0 0
0 0 3 3 3 3 3 3 2 2 2 2 1 1 1 0 0 0 0
```

Fig 3. Dialogue in the J programming language illustrating infix and prefix on 'box', 'sum' and 'xor'

The last two fragments of the dialogue in Fig. 3 illustrate that **infix** and **prefix** on the operator **xor** are each others inverse – almost, that is, because **infix** reduces the number of elements in the sequence by 1. This is repaired in the example in Fig. 4, where the first element of the top line is put in front of the second line (the result of **infix**). Applying **prefix** now completely restores it.

A	1 1 1 1 1 3 3 3 3 1 1 1 1 2 2 2 2 1 1 1 1 1 1 1
B = 1,xor infix A	1 0 0 0 0 2 0 0 0 2 0 0 0 3 0 0 0 3 0 0 0 0 0 0
xor prefix B	1 1 1 1 1 3 3 3 3 1 1 1 1 2 2 2 2 1 1 1 1 1 1 1

Fig. 4. Using **xor infix** to convert a full grid line into sparse boundary representation, and **xor prefix** to go back.

When applying **xor infix** to each line of a two-dimensional 'full pixel' grid (i.e. a grid where each pixel of an area-object has the value of that object), we obtain what we call the sparse boundary grid. The first element (pixel) of each 'full grid' line is placed in front of the 'sparse grid', maintaining the total number of pixels, and allowing for restoration by **xor prefix**. When extending infix and prefix to two dimensions, we can choose between applying them to the columns of the grid (Fig 5, centre), or to the rows (fig. 5, right). The default in J would be the first; for the second we have to apply a modifier, here defined as **inrows**.

```
   M;(infix M);(infix inrows M)
┌─────────────────┬─────────────────┬─────────────────┐
│1 1 1 1 1 1 1 1 1│1 1 1 1 1 1 1 1 1│1 0 0 0 0 0 0 0 0│
│1 1 1 1 2 2 2 1 1│0 0 0 0 3 3 3 0 0│1 0 0 0 3 0 0 3 0│
│1 1 1 2 2 2 2 1 1│0 0 0 3 0 0 0 0 0│1 0 0 3 0 0 0 3 0│
│1 1 3 3 3 3 3 1 1│0 0 2 1 1 1 1 0 0│1 0 2 0 0 0 0 2 0│
│1 3 3 3 3 3 3 1 1│0 2 0 0 0 0 0 0 0│1 2 0 0 0 0 0 2 0│
│1 1 1 1 3 3 3 1 1│0 2 2 2 0 0 0 0 0│1 0 0 0 2 0 0 2 0│
│1 1 1 1 1 1 1 1 1│0 0 0 0 2 2 2 0 0│1 0 0 0 0 0 0 0 0│
│1 1 1 1 1 1 1 1 1│0 0 0 0 0 0 0 0 0│1 0 0 0 0 0 0 0 0│
└─────────────────┴─────────────────┴─────────────────┘
```

Fig. 5. Transforming a full area grid (left) into a sparse boundary grid by **xor infix** , either column wise (centre) or row wise (right).

The boundary grids are called sparse, because the created boundaries are not completely closed – the non-zero pixels do not form a connected path around each area.

The corresponding **xor prefix** operation (either column wise or row wise) will restore the full area grid from the sparse boundary grids (Fig. 6).

```
   M;N;xor prefix N =.(first, xor infix) M
┌─────────────────┬─────────────────┐
│1 1 1 1 1 1 1 1 1│1 1 1 1 1 1 1 1 1│1 1 1 1 1 1 1 1 1│
│1 1 1 1 2 2 2 1 1│0 0 0 0 3 3 3 0 0│1 1 1 1 2 2 2 1 1│
│1 1 1 2 2 2 2 1 1│0 0 0 3 0 0 0 0 0│1 1 1 2 2 2 2 1 1│
│1 1 3 3 3 3 3 1 1│0 0 2 1 1 1 1 0 0│1 1 3 3 3 3 3 1 1│
│1 3 3 3 3 3 3 1 1│0 2 0 0 0 0 0 0 0│1 3 3 3 3 3 3 1 1│
│1 1 1 1 3 3 3 1 1│0 2 2 2 0 0 0 0 0│1 1 1 1 3 3 3 1 1│
│1 1 1 1 1 1 1 1 1│0 0 0 0 2 2 2 0 0│1 1 1 1 1 1 1 1 1│
│1 1 1 1 1 1 1 1 1│0 0 0 0 0 0 0 0 0│1 1 1 1 1 1 1 1 1│
└─────────────────┴─────────────────┴─────────────────┘
   M;N;xor prefix inrows N =.(first, xor infix) inrows M
┌─────────────────┬─────────────────┐
│1 1 1 1 1 1 1 1 1│1 0 0 0 0 0 0 0 0│1 1 1 1 1 1 1 1 1│
│1 1 1 1 2 2 2 1 1│1 0 0 0 3 0 0 3 0│1 1 1 1 2 2 2 1 1│
│1 1 1 2 2 2 2 1 1│1 0 0 3 0 0 0 3 0│1 1 1 2 2 2 2 1 1│
│1 1 3 3 3 3 3 1 1│1 0 2 0 0 0 0 2 0│1 1 3 3 3 3 3 1 1│
│1 3 3 3 3 3 3 1 1│1 2 0 0 0 0 0 2 0│1 3 3 3 3 3 3 1 1│
│1 1 1 1 3 3 3 1 1│1 0 0 0 2 0 0 2 0│1 1 1 1 3 3 3 1 1│
│1 1 1 1 1 1 1 1 1│1 0 0 0 0 0 0 0 0│1 1 1 1 1 1 1 1 1│
│1 1 1 1 1 1 1 1 1│1 0 0 0 0 0 0 0 0│1 1 1 1 1 1 1 1 1│
└─────────────────┴─────────────────┴─────────────────┘
```

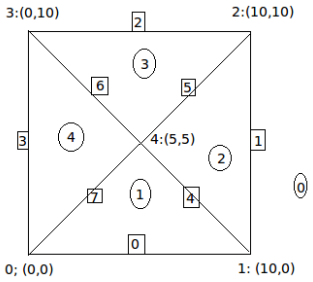
Fig 6. Illustration of xor infix converting a full area grid (left) to a sparse boundary grid (centre), either by columns (top) or by row (bottom). The corresponding **xor prefix** converts in the other direction.

2.3 Stage 1: Vector edge to sparse boundary

During the first stage of a 2D polygon-to-raster conversion the edges of the input dataset are converted to pixels in the sparse boundary grid. This process can be entirely completed on the bases of the edges (that make up the polygons) alone, without taking the interiors of the polygons into account.

When preparing for a row-wise **xor prefix** in stage 2, for each edge of the dataset the process is as follows: first the minimum and the maximum row of the gridded edge are determined, according to the y-coordinates of the end-points of the edge, taking the resolution into account, as well as the extent of the entire area). Then, for each row in the range between this minimum and maximum, one column number is computed: the (vertical) column at which the edge intersects the (horizontal) row under consideration, based on the parameters of the edge as given by the two end-points. At the (row, column)-index obtained one value is placed into the grid: the **xor** of the two values of the polygons on either side of the edge.

The process is illustrated in the example of Fig. 7, showing four areas (triangles 1 – 4), surrounded by a background with value 0. There are five vertices, numbered from 0 to 4, and eight edges, numbered from 0 to 7. In each edge the two vertices it connects are stored, as well as the two areas it separates. The two edges are allowed to be in either order, independent of their positions in the map. The same holds for the two area values. The algorithm does not need to know about left and tight, or about clockwise and anticlockwise. From the pairs of area values of all the edges, **xor** values are computed.



vertices		edges		xor
0:	0 0	0:	0 1 0 1	1
1:	10 0	1:	1 2 0 2	2
2:	10 10	2:	2 3 0 3	3
3:	0 10	3:	3 0 0 4	4
4:	5 5	4:	1 4 1 2	3
		5:	2 4 2 3	1
		6:	3 4 3 4	7
		7:	0 4 4 1	5

Fig. 7: Two-dimensional vector input dataset: graphical representation (left) and the necessary data structure (right).

Fig. 8 shows at its left how the **xor** values from Fig. 7 are distributed in the sparse boundary grid, which had initially been filled with zero values.

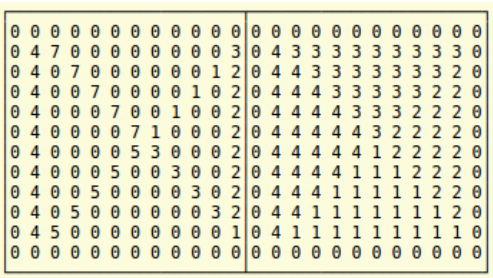


Fig. 8: Sparse boundary grid generated from the vector dataset in Fig. 7 (left), and the resulting full area grid after applying **xor prefix** (right)

Inside the algorithm, extra care is taken of the exact positioning of the values in the sparse boundary grid, for the fact that the entire grid seems to be shifted half a pixel to the right. Another concern is about which area "wins" when multiple ones meet at one vertex: inserting a value (being the **xor** of two area numbers) into the grid is taking the **xor** of that value and the value that was already there. Therefore the edges of a certain polygon at one vertex will either cancel out each other, or occupy two pixels next to each other (leading to a single pixel being affected after stage 2).

In case of having input data with separate polygons, i.e. without topology, the vertices with numbers 4-7 would appear twice, adding edges 8-11 to the set: each edge comes once with one area number plus background, and once with the other area number plus background, and the **xor**-ing takes place when inserting the second of these into the grid.

A real-life example is shown in Figure 9. It is a historical land use map of the centre of the City of Adelaide in Australia, containing 192 polygons with 24 classes, 1296 vertices (332 nodes and 964 break points) and 424 boundary lines. Here, nodes are vertices where three or more boundaries meet, whereas break points separate boundary lines into multiple polygons edges. Note that the road network in this dataset is a single polygon.

This dataset is rasterized here into a grid of 708 x 929 pixels.

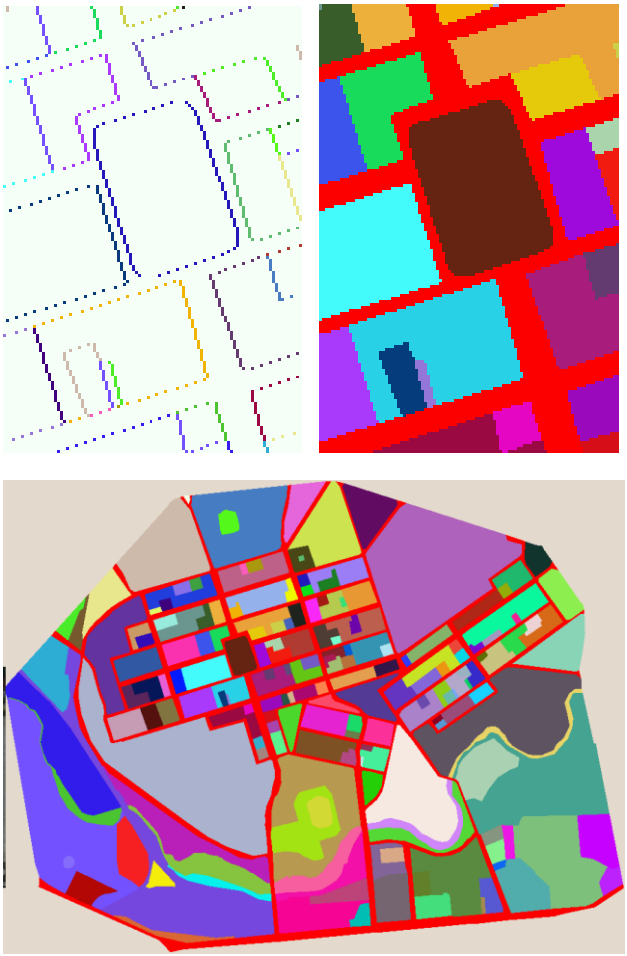


Fig. 9 Historical land use map of the City centre of Adelaide (AUS) (below), with a detail (upper right) and its preceding sparse boundary grid (upper left).

3. POLYHEDRON TO VOXEL CONVERSION (VOXELIZATION)

The methodology for converting 2D polygons to raster, described above, can be extended to the 3D conversion of polyhedra to voxel grids. Whereas is the 2D case the algorithm is relying on edges, forming the boundaries between area's, this role is now given to faces that form the boundary surfaces between solids in 3D space. Therefore, each object (solid) is thought to be enclosed by a polyhedron that can be subdivided into planar faces. Like an edge in 2D, a face can be part of a boundary between two identified solids, or between a solid and the background, such as the air in a building model – or the outside air if the indoor empty spaces are being modelled explicitly. Also when all faces are designated to be bounding just single objects (having 'background' at the other side), those faces that actually separate the two identified objects will be dealt with properly by the **xor** operation during the gridding process.

Faces are 3D polygons, made up of vertices having 3D (*x,y,z*) coordinates. Faces must be planar, also when having more than three vertices.

The entire process again consists of two stages. At the end of stage 1, a sparse 3D boundary grid has been created, which is

converted to a full 3D solids grid during stage 2. This stage 2 is very similar to the 2D case, except that there are now three directions to choose from: left to right, front to back or top to bottom. (Note that the opposite directions would be possible as well, but we haven't mentioned this in the 2D case either). We will describe the top-to-bottom case. Of course, this also determines the appearance of the sparse boundary grid. Notably it implies that vertical walls will not explicitly appear in the sparse boundary grid, but are solely represented by their top and bottom xor values (Fig. 10 left and Fig. 13).

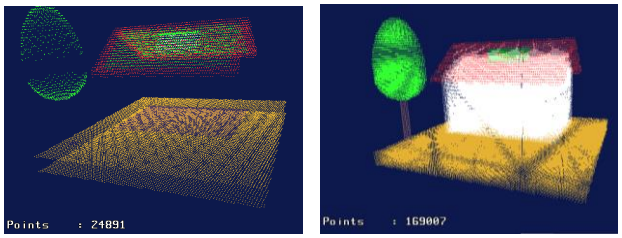


Fig. 10: 3D sparse boundary grid with different **xor** values after voxelization stage 1 (left) and full voxel grid after top-to-bottom **xor prefix** in stage 2 (right)

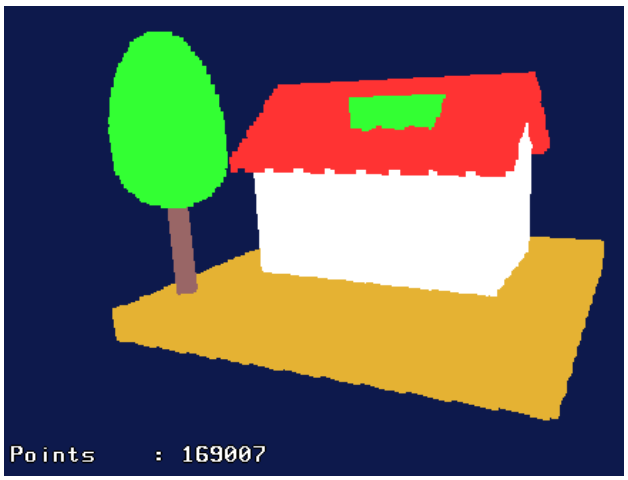


Fig 11. The same result as the right hand side of Fig. 10, shown with a larger voxel point size.

As for stage 1: after setting up an empty 3D grid (i.e. filled with zero values) of the right size, depending on the extent of the dataset and the desired resolution, stage 1 proceeds working face by face. Given the (x,y) coordinates of the face's vertices, the extent of the horizontal grid coordinates in the resulting space is determined, and the smallest possible 2D grid surrounding these coordinates is formed. Into this small 2D grid a raster representation of the projection of the face into the (x,y)-plane is created, exactly following the description of Section 2. The result is a polygonal area filled with 1's surrounded by a background of 0's. The (row, column) coordinates of the 1's are shifted to the horizontal grid coordinates of the 3D grid, after which they are completed with a third (vertical) grid coordinate on the basis of the plane equation of the face, as determined from the 3D vertex coordinate set.

At the index thus created a value is inserted into the 3D grid, being the **xor** of the values given by the two solids (or of one solid and 'background'). Insertion means taking the **xor** of the new value and the one that might be already present in the grid (or 0)

The visualisations in Fig. 10 (right) and 11 clearly show different voxel values by different colours. Note that in the full solid grid also the 'invisible' voxels are present, having the value 'background' = 0.

The crown of the tree to the left of the house is not a collection of planar faces. Instead, it consists of two hemispheres, which are parametrized by x_0, y_0, z_0, R and e and v. Here, x_0, y_0 and z_0 determine the centre of a (hemi)sphere in grid coordinates, and R its radius. For (x,y) pairs within distance R from (x_0,y_0) a z is determined according to the equation of a hemisphere with radius R, scaled vertically by e (which may also be negative). At the positions in the grid at index (x,y,z) the value v, **xor**-ed with 'background', is inserted by using another **xor**.

Another example, showing the interior of a building too, is presented in Fig. 12, 13, 14 and 15. The furniture inside the building is represented by polyhedra too, in this case the minimum maximum extend of the furniture. The polyhedral touch but again do not overlap. Here, the only distinction in voxel values is between 'object' and 'background' (air) values. In the full solids grid these are represented by 1's and 0's respectively. In the visualisation only the 1's are shown, made semi-transparent in the final result.

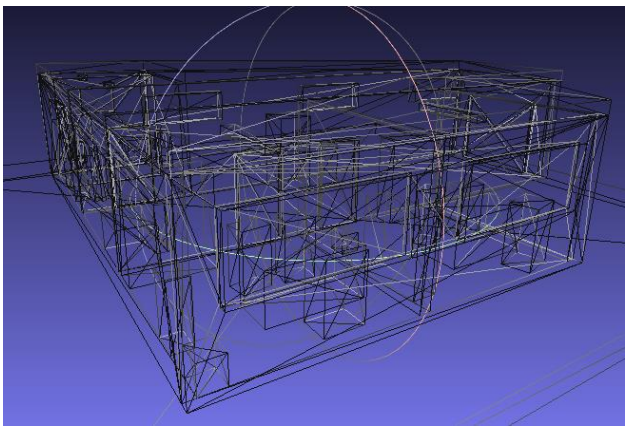


Fig. 12. Vector input for a building with interior rooms and furniture

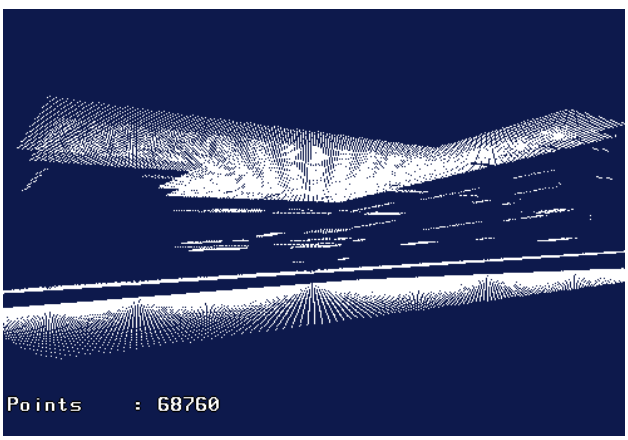


Fig. 13. Sparse boundary grid of model of Fig. 12 at 10cm grid resolution

This contribution has been peer-reviewed.
doi:10.5194/isprsarchives-XLI-B4-283-2016

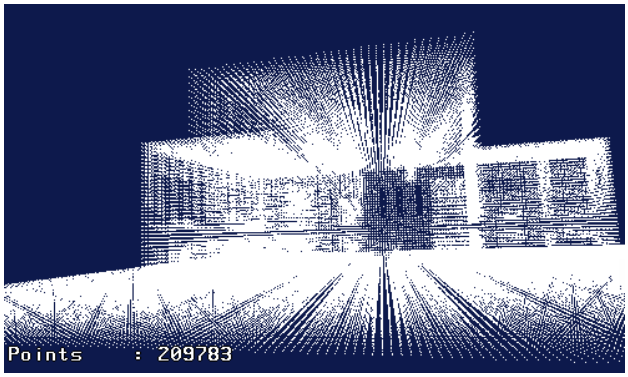


Fig. 14. Voxel cloud of the model in Fig.12. The voxels are represented as points.

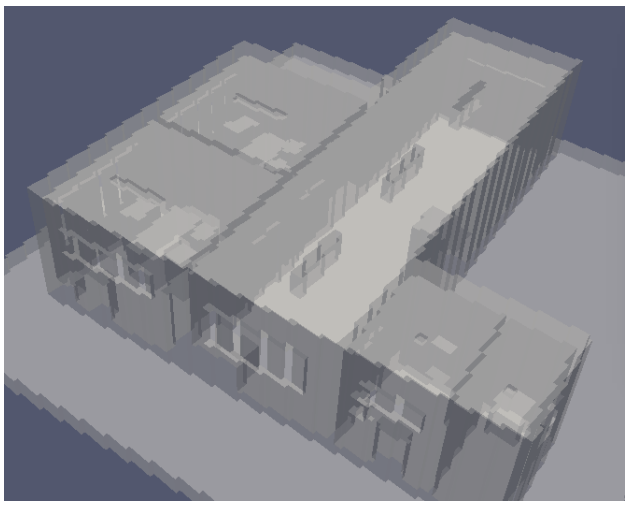


Fig. 15: Voxelized model

4. CONCLUSION

We present a new approach for the conversion of 2D and 3D spatial data from vector representation into grids of pixels and voxels, respectively. Input are sets of vector boundaries between adjacent areas (in 2D) or solids (in 3D). Each area (solid) has a value, which will determine the value of the contained pixels/voxels of the result. 2D boundaries are piecewise linear (edges between vertices), whereas 3D boundaries are piecewise planar (plane polygons defined by 3D vertices). Polygons and polyhedral may be nested. Topological information in input datasets is not required, but its presence increases the efficiency of the operation.

The approach is conceptually straightforward and can be easily re-implemented in other programming environments. It consists of two stages, the first being object (boundary) based, the second scan-line based. Input formats containing other primitives than planar faces, such as ellipsoids, can be accommodated in the first stage without affecting the second.

During the first stage a minimal set of voxels have to be updated in a buffer in main memory. The indices are computed by straightforward operations, such as applying a plane equation. The performance is therefore considered to be optimal within the boundaries of the task under consideration. Further enhancement by GPU processing seems adequate, but has to be developed. The second stage is a simple operation based on the xor-instruction that is implemented in any CPU hardware, and is currently significantly faster than the first.

REFERENCES

Eisemann, E. and Décoret, X., 2008, Single-pass GPU solid voxelization for real-time applications, *Proceeding GI '08 Proceedings of Graphics Interface 2008*, pp. 73-80

Gorte, B., R. Liem, J. Wind, 1988, The ILWIS software kernel, *ITC Journal* 1, pp. 15-22

Huang, J, Yage R, Filippov V and Kurzion Y, 1998. An Accmate Method for Voxetiting Polygon Meshes. In: *IEEE Symposium on Volume Visualization* (Cat. No.989EX300). s.l.:IEEE, p. 119–126.

Kaufman, A. & Shimony, E., 1987. *3D scan-conversion algorithms for voxel-based graphics*. New York, ACM Press, p. 45–75.

Kaufman, A., 1987. *Efficient algorithms for 3D scan-conversion of parametric curves, surfaces, and volumes*. New York, ACM Press, pp. 171-179.

Laine, S., 2013. A Topological Approach to Voxelization. *Computer Graphics Forum*, 32(4), p. 77–86.

Min, P, 2016, BinVox Binvox, available at http://www.google.com/search?q=binvox

Milossramek, 2013, Voxelisation toolkit, available at http://voxelization-toolkit.soft112.com/

Nourian, P., R. Gonçalves, S. Zlatanova, K. Arroyo Ohori and A-V Vo, 2016, Voxelization algorithms for geospatial applications: computational methods for voxelizing spatial datasets of 3D city models containing 3D surface, curve and point data models. *MethodsX* 3, January 2016, pp. 69–86.

Wang, S.W. & Kaufman, A.E., 1993. Volume sampled voxelization of geometric primitives. s.l., *IEEE Comput. Soc. Press*, p. 78–84.

Xie, X., Liu, X. & Lin, Y., 2009. The investigation of data voxelization for a three-dimensional volumetric display system. *Journal of Optics A: Pure and Applied Optics*, 11(4), p. 45707.