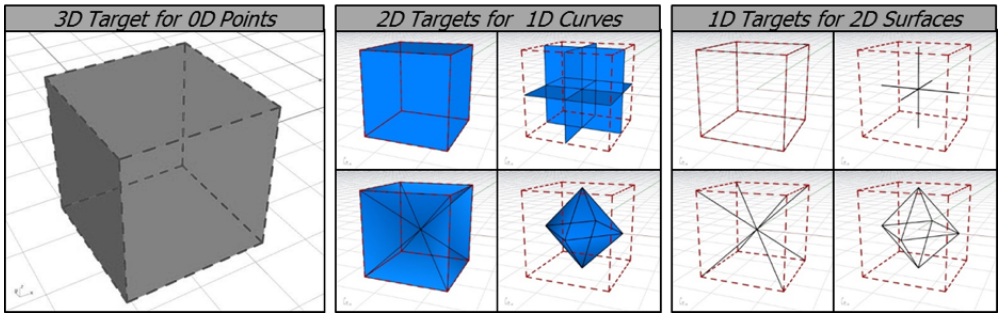
Voxelization Algorithms for Geospatial Applications: Computational methods for voxelating spatial datasets of 3D city models containing 3D surface, curve and point data models

Pirouz Nourian Romulo Gonçalves Sisi Zlatanova
Ken Arroyo Ohori Anh Vu Vo

This is an author's version of the paper. The authoritative version is:

Voxelization Algorithms for Geospatial Applications. Pirouz Nourian, Romulo Gonçalves, Sisi Zlatanova, Ken Arroyo Ohori and Anh Vu Vo. *MethodsX* 3, January 2016, pp. 69–86. ISSN: 2215-0161.
DOI: 10.1016/j.mex.2016.01.001

Related source code is available at
`https://github.com/NLeSC/geospatial-voxels/tree/master/software/voxelGen`



Voxel representations have been used for years in scientific computation and medical imaging. They are the main focus of our research where we provide easy access to methods for making large-scale voxel models of built environment for environmental modelling studies that are spatially correct, meaning they correctly represent topological and semantic relations among objects. In this article we present algorithms that generate voxels (volumetric pixels) out of point cloud, curve, or surface objects. The algorithms for voxelization of surfaces and curves are a customization of the Topological Voxelization approach [Laine, 2013]; we additionally provide an extension of this method for voxelization of point clouds.

1 Introduction

We present three methods for voxelating point, curve, and surface objects. For curve (1D) and surface (2D) objects we present algorithms for the methods mathematically described by Laine [2013] and for voxelizing points (0D) we devise our own algorithms. Note that 0D, 1D, and 2D refer to the topological dimension of inputs; this, however does not contradict the fact that all inputs are embedded in three-dimensional Euclidean space of $\mathbb{R}^3$, i.e. a Cartesian product of X, Y and Z coordinates represented in the domain of real numbers $\mathbb{R}$. Full scripts of algorithms are available in the abovementioned repository. Laine [2013] mathematically proves that using the so-called intersection targets desired connectivity levels such as 6 or 26 connected voxel results could be achieved. He does not present an algorithm in detail though. In developing an algorithm for dealing with large datasets, we had to come up with an efficient way of iteration. In general, one can either iterate over all possible voxels in a bounding box or iterate over objects. It sounds obvious that iterating over objects is more efficient as the whole space is not usually filled with objects. This approach can also lead to inefficiency when it comes to a large mesh object such as a TIN (Triangular Irregular Network) terrain model. However, this issue can be resolved if all objects are decomposed into primitives such as triangles (in case of surface input) or line segments (in case of curve inputs). In such cases the algorithms can iterate over triangles or line segments. In our implementation of the topological voxelization method, we have chosen four intersection targets from those defined by Laine [2013], namely two hairline targets for surface inputs and two mesh targets for curve inputs as shown in Table 1. In the following section we give an overview of topological voxelization.

2 Topological Voxelization in Brief

Laine [2013] mathematically proves that using the so-called "intersection targets" desired connectivity levels such as 6 or 26 connected voxel results could be achieved. The idea can be best understood by asking the following question based on the definitions of connectivity given below:

- 6-Connected Voxel Collection: a voxel collection in which every voxel has at least one face- neighbour, i.e. by virtue of having adjacent faces
- 18-Connected Voxel Collection: a voxel collection in which every voxel has at least one edge- neighbour, i.e. by virtue of having adjacent edges
- 26-Connected Voxel Collection: a voxel collection which every voxel has at least one vertex- neighbour, i.e. by virtue of having adjacent vertices

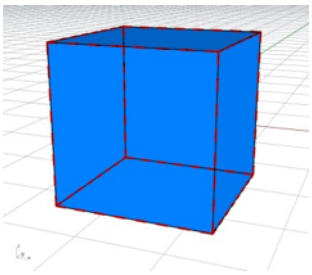
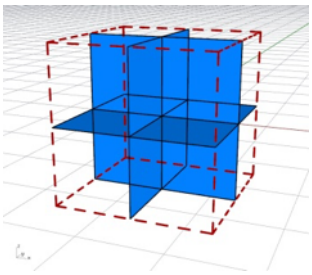
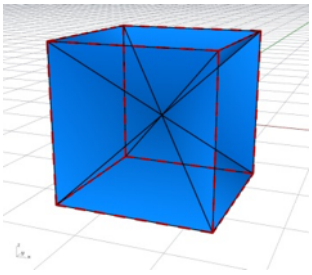
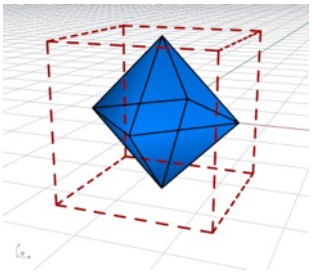
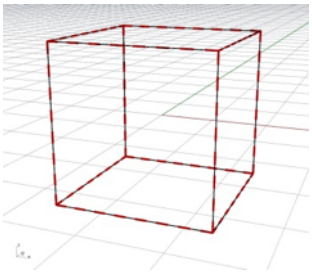
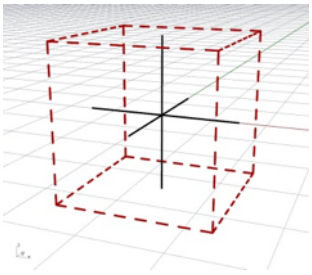
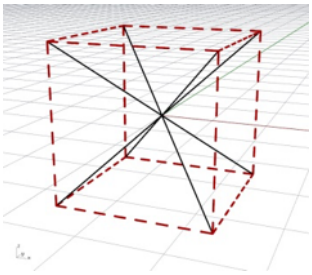
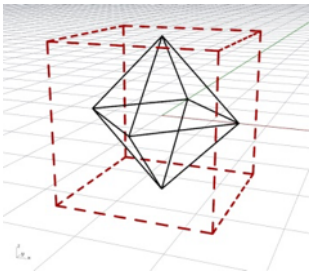
Question 1 given a curve (1D manifold input, as vector data model) how can we obtain a 'thin' voxelated curve (i.e. a voxel collection as a raster representing the curve in question without unnecessary voxels) that is 6 connected or 26 connected?

Question 2 given a surface (2D manifold input, as vector data model) how can we obtain a 'thin' voxelated surface (i.e. a voxel collection as a raster representing the surface in question without unnecessary voxels) that is 6 connected or 26 connected?

Formally, considering adjacency between voxels in a voxel collection can be in the sense of face- adjacency, edge-adjacency or vertex adjacency. Analogous to these definitions, we can mention edge-adjacency or vertex adjacency between pixels which correspond to 4-neighbourhoods (a.k.a. von Newman neighbourhoods[1]) and 8-neighbourhoods (a.k.a. Moore neighbour-

[1] http://mathworld.wolfram.com/vonNeumannNeighborhood.html

Table 1: Intersection targets for topological voxelization reproduced after Laine [2013]

Output connectivity →	6-connected	26-connected
1D (curve) inputs		
		
2D (surface) inputs		
		

hoods[2]) respectively. These neighbourhood definitions are mostly known in definition of Cellular Automata[3] (CA) models. The term 'connected' refers to the graph that represents the adjacencies between voxels or pixels with the given definition of adjacency. It is easier to conceive of the principle of guaranteeing connectivity in raster (pixel or voxel) output by looking at the 2D case in Figure 1.

1D Input (Curves)

- **(saleable) 6-Connected Intersection Target**: The 6 faces of a voxel cube
- **(saleable) 26-Connected Intersection Target**: 3 plane faces cutting a voxel cube in halves along X,Y and Z, meeting at the centre of the voxel cube

[2] http://mathworld.wolfram.com/MooreNeighborhood.html
[3] http://mathworld.wolfram.com/CellularAutomaton.html

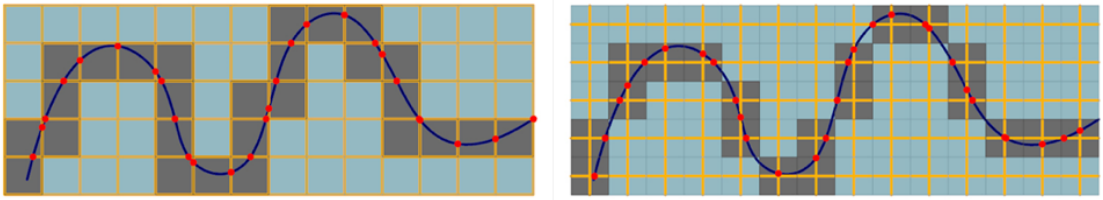


Figure 1: topological pixilation after Laine [2013] the rasterization at the left is based on an 'outline' intersection target which ensures a 4-connected rasterized curve in which every pixel has at least one neighbour sharing an edge with; the rasterization at the right is based on a 'crosshair' intersection target which ensures an 8-connected rasterized curve in which every pixel has at least one neighbour sharing a vertex with. Resulting pixels are highlighted in dark grey. Intersection Targets are highlighted in yellow and intersections with them in red dots. Pixels are shown as blue squares.

2D Input (Surfaces): voxelization

- **(saleable) 6-Connected Intersection Target**: The 12 edges of a voxel cube
- **(saleable) 26-Connected Intersection Target**: A 3D crosshair centred at a voxel cube made of lines going through face centres along X,Y and Z axes

3 What is an Intersection Target?

The fundamental idea behind using Intersection Targets is based on what is called Poincaré Duality Theorem [Munkres, 1984] or similar to the approach of Pigot [1991] and Lee [2001], which establishes a primal-dual pairing between 'primal' k-dimensional features and their $(n-k)$-dimensional 'duals' in an n-dimensional space ($\mathbb{R}^n$), within which these objects are 'embedded'. In 3D space, we can think of dualities as shown in Table 4. For a clearer picture we also show potential dual pairs in 1D and 2D spaces respectively in Table 2 and Table 3. Simply put, an Intersection Target is a dual feature of dimension $n-k$ ($3-k$ in case of the 3D space of $\mathbb{R}^3$) that must be intersected with the input feature (primal) to determine whether a voxel should be added to the voxelated output object or not.

A well-known example of duality between 2D maps is the duality between a Delaunay triangulation and a Voronoi tessellation.

An example of dual relationships in 3D is shown in Figure 2. A face in the left image can be considered as the element through which two 3D cells are connected; this is why representing the same face with an edge in the dual graph makes sense as it connects two vertices representing the respective 3D cells.

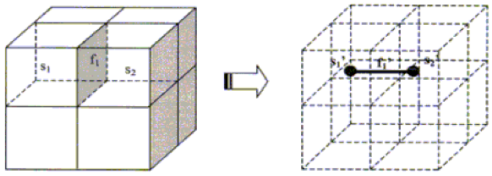


Figure 2: representing adjacencies between 3D cells or bodies via their dual vertices [Lee, 2001]

4 What is special about topological approach over alternatives?

In many geospatial applications it is important to preserve topological relations among objects [Egenhofer and Herring, 1990; Zlatanova, 2000]. This set of topological relations is used for many purposes

Table 2: Duality of features in 1D space

Primal Features	Dual Feature
0D vertex (e.g. a point)	1D edge
1D edge (e.g. a line segment)	0D vertex

Table 3: Duality of features in 2D space

Primal Features	Dual Feature
0D vertex (e.g. a point)	2D face
1D edge (e.g. a line segment)	1D edge
2D face (e.g. a triangle or a pixel)	0D vertex

such as to analyse relationships between geographical objects [Louwsma et al., 2006], to ensure validity of geospatial datasets [Arroyo Ohori et al., 2012], or to construct 'graphs', as in navigation and path planning [OGC, 2014]. In the raster domain, connectivity levels in the output voxels of a voxelation process are essentially topological properties, which are best handled through an explicitly topological approach. The topological approach brings added elegance and clarity to the voxelization process and allows for obtaining desired connectivity levels in a systematic way.

In the topological approach to voxelization [Laine, 2013] the idea is that if we are to decide whether a voxel 'needs to be' in the output (so as to ensure preserving the topological properties of the input features); we need to ensure its relevant Intersection Target intersects with the input. The nature of these Intersection Targets is dual to the nature of the primal inputs. This way, there will be no doubt on the necessity of having a voxel in the output; i.e. the results are guaranteed to be 'minimally thin' as to representing a raster version of the original vector features. We refer the reader to the mathematical proofs given in Laine [2013]. Here we give an intuitive explanation for the 2D example shown in Figure 1. This figure shows a 2D version of topological rasterization (pixelization) of a 1D input curve. If we want to ensure that every necessary pixel is added to the results, we need to focus on the connectivity level desired (4 or 8 neighbours for each pixel). If we want our pixelated curve to be a 4-Connected path so as to best represent the connected underlying curve, then we need a determining factor for including a pixel in the outcome. Let us figuratively imagine an ant going through that curve somewhere, we want to know from which spaces (pixels) it has passed; and that we want to reconstruct its path as a sequence of interconnected pixels sharing a wall with one another. If the ant in question crosses a wall-edge in one pixel it definitely enters to a 4- neighbouring cell. Therefore, checking the crossings of the path curve with the pixel boundaries would be enough to decide for including a pixel in the rasterized path.

The prevalent alternative approaches, which are quite common as to their efficiency can be seen as variants of Scan Conversion [Kaufman and Shimony, 1986], which works by interacting rays in 3D directions passing through objects. If all voxels inside are needed then voxels coinciding with intersection points between odd and even intersections will be added to the rasterized (voxelated) output. In our approach this operation corresponds to finding those 0D voxel centre points which happen to be intersecting with the 3D volume (inside or on the volume)[4]. Observe that the last pair of dual features in Table 4 actually corresponds to voxelating a volume in this way.

The topological approach can also eventually be adapted to be implemented based on rays, so as to make it more efficient, but that

[4] We have not included an algorithm for this case, as it seemed quite straightforward.

Table 4: duality of features in 3D space

Primal Features	Dual Feature
0D vertex (e.g. a point)	3D body
1D edge (e.g. a line segment)	2D face
2D face (e.g. a triangle or a pixel)	1D edge
3D body (e.g. a tetrahedron or a voxel)	0D vertex

subject would fall out of scope of the current paper. We can simply say that such a scaling approach would be based on considering Meta Intersection Targets such as rays for meshes and half-planes for curves and finding intersection parameters along these Meta Intersection Targets to locate meeting voxels.

5 Point Voxelization (0D Inputs, 3D Targets)

In this section the voxelization algorithm for 0D data inputs, i.e. point clouds, is described in detail. The algorithm has been devised and implemented in C# and its implementation is available in our GitHub repository. Its pseudo code is in Algorithm 1. In the initialization phase, the algorithm creates a bounding box for $\mathbb{Z}^3$ coordinates that is larger than or equal to the size of the bounding box of the point cloud in $\mathbb{R}^3$. It then finds how many voxels could be in each direction by finding the floor integer closest to the size of bounding box in each direction divided by the voxel size in the corresponding direction. It then initializes a 3-dimensional array of Boolean (1 bit) values of respective X, Y, and Z sizes plus one (i.e. minimal in size for it only stores bits). This array will be used to keep track of voxels already visited as tuples of $[i, j, k]$ in $\mathbb{Z}^3$. If not yet visited, it marks it as visited (true). It then continues by 'embedding' the voxel in $\mathbb{R}^3$, that is through mapping the $[i, j, k]$ voxel in $\mathbb{Z}^3$ to $\mathbb{R}^3$ by first creating a point of respective voxel size and then shifting it first for half of the voxel size vector and then for the point at the minimum corner of the $\mathbb{Z}^3$ bounding box.

The RBBox_to_ZBBox algorithm adjusts a bounding box in $\mathbb{R}^3$ to a larger or equal size bounding box in $\mathbb{Z}^3$. Its pseudo code, described in Algorithm 2, uses MinBoundRP_to_ZP and MaxBoundRP_to_ZP to ensure the new bounding box contains the $\mathbb{R}^3$ bounding box. The time complexity of the algorithm is linear, i.e., $O(n)$ where n is the number of points in the point cloud.

Figure 4 and Figure 5 show the application of this algorithm on a sample point cloud data from AHN (Actual Height of Netherlands dataset) of Noordereiland in Rotterdam, shown in Figure 3. This point cloud is consisted of 136828 points as X, Y, and Z coordinate tuples, we have provided this point cloud in our GitHub repository. Figure 5 depicts voxels with higher resolution than those shown in Figure 4.

Figure 3: Rotterdam AHN dataset, site in Noordereiland

6 Curve Voxelization (1D Inputs, 2D Targets)

According to the topological voxelization approach [Laine, 2013] if an effectively one dimensional input such as a curve is intersected with voxels replaced by relevant intersection targets mentioned in the Table 1,

Algorithm 1: VOXELIZEPOINTCLOUD

Input : `Point3d[]` *PointCloud*, `double` *sx*, `double` *sy*, `double` *sz*
Output: `List<Point3d>` voxels

Initialize **begin**
    form a bounding box `BBox` in $\mathbb{R}^3$ for the *PointCloud*, i.e. composed of six intervals between the *MaximumPoint* with largest X, Y, Z coordinates and the *MinimumPoint* with the smallest;
    define voxels as new list of 3D points;
    define voxel size as `Vector3d` $vSize = [sx, sy, sz]$;
    `/* generate a bounding box equal or larger than the current bounding box in` $\mathbb{Z}^3$ `(c.f., Algorithm 2) */`
    $ZBBox$ = RBBox_TO_ZBBox($RBBox, vSize$);
    compute $CX = \lfloor (ZBBox.Max.X - ZBBox.Min.X)/Sx \rfloor$; `/* number of voxels in` X `direction */`
    compute $CY = \lfloor (ZBBox.Max.Y - ZBBox.Min.Y)/Sy \rfloor$; `/* number of voxels in` Y `direction */`
    compute $CZ = \lfloor (ZBBox.Max.Z - ZBBox.Min.Z)/Sz \rfloor$; `/* number of voxels in` Z `direction */`
    define `bool[,,]` $Voxels3I$ as a 3D array of Booleans of size $[CX, CY, CZ]$;
    define `List<Point3d>` $VoxelsOut$ as new list of 3D points;
Compute voxels **begin**
    foreach *vertex as* `Point3d` *in the point cloud* **do**
        compute integer $i = \lfloor (vertex.X - ZBBox.Min.X)/Sx \rfloor$;
        compute integer $j = \lfloor (vertex.Y - ZBBox.Min.Y)/Sy \rfloor$;
        compute integer $k = \lfloor (vertex.Z - ZBBox.Min.Z)/Sz \rfloor$;
        `/* Check if a voxel was already generated for the` i, j, k `location in` $\mathbb{Z}^3$`, if not generate one */`
    if $Voxels3I[i, j, k] = false$ **then**
        $Voxels3I[i, j, k] = true$ `/* mark it as visited */`
        `Point3d` $VoxelOut$ = new `Point3d`$(i \times Sx, j \times Sy, k \times Sz)$;
        $VoxelOut = VoxelOut + VSize/2 + Zbbox.Min$; `/* change its reference as to the` Z `*/`
        $VoxelsOut.Append(VoxelOut)$; `/* and add it to the voxel output */`
return voxels;

Algorithm 2: RBBox_TO_ZBBox

Input : A bounding box in $\mathbb{R}^3$, i.e. composed of three intervals in X, Y, and Z directions
Output: A bounding box embedded in $\mathbb{R}^3$, Corresponding to the voxels in $\mathbb{Z}^3$

RBBox_to_ZBBox **begin**
    `/* to ensure the bounding box for voxels covers the whole bounding box in` $\mathbb{R}^3$ `*/`
    compute `Point3d` $ZMinP$ = MINBOUNDRP_TO_ZP($RBbox.Min, vSize$); `/* min box corner */`
    compute `Point3d` $ZMaxP$ = MAXBOUNDRP_TO_ZP($RBbox.Max, vSize$); `/* max box corner */`
    return new `BBox`($ZMinP, ZMaxP$);
Function MINBOUNDRP_TO_ZP($RPoint, VSize$) **begin**
    compute $x = RPoint.X; y = RPoint.Y; z = RPoint.Z$;
    compute $u = VSize.X; v = VSize.Y; w = VSize.Z$;
    compute $ZP_x = u.\lceil x/u \rceil$; `/* ceiling integer nearest to the` X `coordinate of the point */`
    compute $ZP_y = v.\lceil y/v \rceil$; `/* ceiling integer nearest to the` Y `coordinate of the point */`
    compute $ZP_z = w.\lceil z/w \rceil$; `/* ceiling integer nearest to the` Z `coordinate of the point */`
    return ZP = new `Point3d`(ZP_x, ZP_y, ZP_z);
Function MAXBOUNDRP_TO_ZP($RPoint, VSize$) **begin**
    compute $x = RPoint.X; y = RPoint.Y; z = RPoint.Z$;
    compute $u = VSize.X; v = VSize.Y; w = VSize.Z$;
    compute $ZP_x = u.\lfloor x/u \rfloor$; `/* floor integer nearest to the` X `coordinate of the point */`
    compute $ZP_y = v.\lfloor y/v \rfloor$; `/* floor integer nearest to the` Y `coordinate of the point */`
    compute $ZP_z = w.\lfloor z/w \rfloor$; `/* floor integer nearest to the` Z `coordinate of the point */`
    return ZP = new `Point3d`(ZP_x, ZP_y, ZP_z);

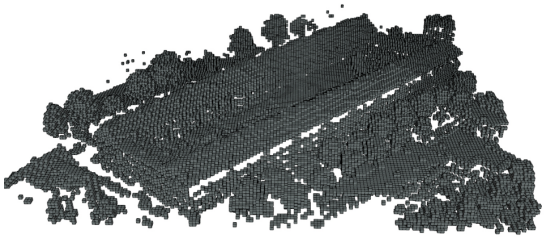


Figure 4: Rotterdam AHN-2 dataset, Noordereiland, voxel count 31880, $1 \times 1 \times 1$ m

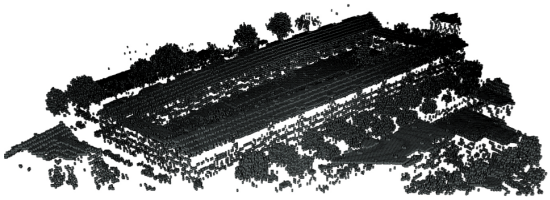


Figure 5: Rotterdam AHN-2 dataset, Noordereiland, voxel count 81588, $0.4 \times 0.4 \times 1$ m

then the resulting set of voxels is guaranteed to have the desired connectivity level, namely 6 or 26. The idea is to construct intersection targets as triangle meshes and then:

1. form a bounding box for the curve in question;
2. adjust the bounding box to ensure its size is an integer multiple of the voxel size and that it covers the whole curve;
3. fill in the bounding box with voxels; find out voxels which potentially intersect with the curve in question, i.e. those closer than half of the size of a virtual vector representing the diagonal of a voxel cube;
4. for each voxel that is near enough to be possibly included in the result, find out if its relevant connectivity target intersects with the curve in question; if yes then add it to the voxels.

Here we explain our voxelization algorithm for 1D data inputs, i.e. lines, polylines or curves that can be approximated as such[5].

The idea behnid the algorithm TriangleIntersectsLine is to represent an imaginatry intersection point in the triangle using two barycentric coordinates based on the axes defined along two vectors corresponding to two sides of the triangle. If the barycentric parameters termed s & t can be found and they are both positive and add up to one, then there is an intersection inside the triangle in question. An algorithm for TriangleIntesectsLine is available at `http://geomalgorithms.com` (by Dan Sunday, subject: Intersections of Rays, Segments, Planes and Triangles (3D)). The function 1D_26_intersectionTarget described by the pseudo code in Algorithm 4 creates mesh objects that ensure 26 connectivity if used as intersection target for voxelizing 1D input.

The above algorithm is currently implemented such that it produces 12 quadrangular faces and thus 24 triangles. It will be therefore more efficient to implement the above function instead. The function 1D_6_intersectionTarget described by the pseudo code in Algorithm 5 creates 1D mesh ensuring 6-connectivity. Note that this intersection target produces 12 triangles and that this number cannot be reduced, even if the other alternative for a 6-connected result were implemented (i.e. the outline faces of a voxel cube). This implies that if we want to create a better-connected raster with more voxels (6-connected), then we need to do more computation.

The function BboxToVoxels, defined in Algorithm 6, can be used to fill a bounding box with voxels.

The voxelizeCurve algorithm has been applied to a dataset consisting of street centrelines of a neighbourhood in Istanbul called

[5]Note that in our current implementation of the curve voxelization algorithm we are using the Curve-Mesh intersection algorithm from Rhinocommon. This algorithm can be easily rewritten using the line-triangle intersection algorithm (Algorithm 10), provided that the input curve is approximated as a polyline composed of line segments. This is a normal procedure as such approximation is needed for all kinds of curves such as conic sections and NURBS prior to rendering.

Algorithm 3: VOXELIZECURVE

```
Input  : int connectivity, list<polyline> curves, double sx, double sy, double sz
Output: list<Point3d> voxels
1  foreach curve in curves do
2      form a bounding box in ℝ³ for curve;
3      define voxels as list of 3D points;
       /* generate a bounding box equal or larger than the current bounding box in ℤ³ (c.f.,
          Algorithm 2)                                                              */
4      ZBBox = RBBox_to_ZBBox(RBBox, vSize);
5      vSize = [sx, sy, sz];
       /* Pseudo code described by Algorithm 6                                      */
6      VoxelPoints = BBoxToVoxels(ZBBox, vSize);
       /* voxel centre points closer than the length of vSize to the curve which possibly
          intersect;                                                                */
7      near_voxel_points = VoxelPoints.FindAll(lambda|lambda.DistanceTo(Curve) < |vSize|/2);
8      foreach voxel_point in near_voxel_points do
9          if connectivity = 26 then
               /* Pseudo code described by Algorithm 4                              */
10             intersectionTarget = 1D_26_intersectionTarget(voxel_point, vSize);
11         else if connectivity = 6 then
               /* Pseudo code described by Algorithm 5                              */
12             intersectionTarget = 1D_6_intersectionTarget(voxel_point, vSize);
13         else
14             throw exception and prompt: "other connectivity levels undefined";
15         if TriangleIntersectLine(intersectionTarget, curve) then
16             voxels.add(voxel_point);
17     return voxels;
```

Tarlabasi, shown in Figure 5, available on OpenStreetMap[6]. The results are shown in Figure 6. It is important to note that for 6-connectivity the both of the targets mentioned in Table 1 guarantee 6-connectivity and in that sense are equally effective. However, the diagonal mesh target is only composed of 8 triangles whereas the cube outline target will be practically composed of 12 triangles. It is important to note that for solving intersections unambiguously an intersection between a ray or a half-line and a triangle should be found. This means that in practice the diagonal target will 12/8 = 1.5 times faster than a corresponding cube outline target. This is not the final conclusion on efficiency however. As for implementing Meta Intersection Targets, straight faces would be advantageous as they can be eventually replaced by finding intersections to half-planes.

Note that every line in the the voxelized curve network in Figure 6 is 6-connected but the whole network is not. Ensuring such connectivity for the network would require

Figure 6: A set of curves extracted from OpenStreetMap representing a street network Tarlabasi, Istanbul

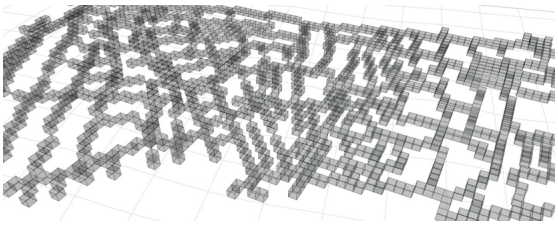
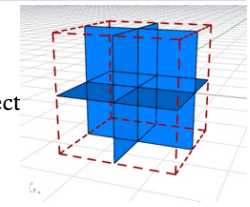


Figure 7: Voxelized curves (street centrelines) with 6-connectivity from Tarlabasi dataset with the resolution $10 \times 10 \times 10$ m

[6] https://www.openstreetmap.org

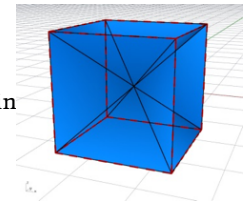
Algorithm 4: 1D_26_INTERSECTIONTARGET



Input : voxel size as `Vector3d`
Output : an intersection target for 1D input ensuring 26 connectivity as a mesh object as in the figure

Initialize **begin**
    `Mesh` *IntersectionTarget* = new *Mesh*();
    `Point3d[]` *Vertices* = new `Point3d`[12];
    `double` $u = (vSize.X/2), v = (vSize.Y/2), w = (vSize.Z/2)$;
Create Vertices **begin**
    Vertices[00] = *VoxelPoint*+ new `Vector3d`$(+u, +v, 0)$; `/* parallel to` XY `*/`
    Vertices[01] = *VoxelPoint*+ new `Vector3d`$(-u, +v, 0)$; `/* parallel to` XY `*/`
    Vertices[02] = *VoxelPoint*+ new `Vector3d`$(-u, -v, 0)$; `/* parallel to` XY `*/`
    Vertices[03] = *VoxelPoint*+ new `Vector3d`$(+u, -v, 0)$; `/* parallel to` XY `*/`
    Vertices[04] = *VoxelPoint*− new `Vector3d`$(0, +v, +w)$; `/* parallel to` YZ `*/`
    Vertices[05] = *VoxelPoint*− new `Vector3d`$(0, -v, +w)$; `/* parallel to` YZ `*/`
    Vertices[06] = *VoxelPoint*+ new `Vector3d`$(0, -v, -w)$; `/* parallel to` YZ `*/`
    Vertices[07] = *VoxelPoint*+ new `Vector3d`$(0, +v, -w)$; `/* parallel to` YZ `*/`
    Vertices[08] = *VoxelPoint*+ new `Vector3d`$(+u, 0, +w)$; `/* parallel to` ZX `*/`
    Vertices[09] = *VoxelPoint*+ new `Vector3d`$(-u, 0, +w)$; `/* parallel to` ZX `*/`
    Vertices[10] = *VoxelPoint*+ new `Vector3d`$(-u, 0, -w)$; `/* parallel to` ZX `*/`
    Vertices[11] = *VoxelPoint*− new `Vector3d`$(+u, 0, -w)$; `/* parallel to` ZX `*/`
Create Faces **begin**
    `List<MeshFace>` *Faces* = *IntersectionTarget.Faces*;
    Faces.AddFace(00, 01, 02, 03); `/* parallel to` XY `*/`
    Faces.AddFace(04, 05, 06, 07); `/* parallel to` YZ `*/`
    Faces.AddFace(08, 09, 10, 11); `/* parallel to` ZX `*/`
    IntersectionTarget.Vertices.AddVertices(*Vertices*);
    IntersectionTarget.Faces.AddFaces(*Faces*);
    return *IntersectionTarget*;

Algorithm 5: 1D_6_IntersectionTarget



Input : voxel size as `Vector3d`
Output: an intersection target for 1D input ensuring 6 connectivity as a mesh object as in the figure

Initialize **begin**
    `Mesh` $IntersectionTarget$ = new `Mesh`();
    `double` $u = (vSize.X/2), v = (vSize.Y/2), w = (vSize.Z/2)$;
    `Point3d[]` $Vertices$ = new `Point3d`[9]; $Vertices[0] = VoxelPoint$;
Create Vertices **begin**
    `Vertices`[1] = $VoxelPoint-$ new $Vector3d(+u, +v, +w)$;
    `Vertices`[2] = $VoxelPoint-$ new $Vector3d(-u, +v, +w)$;
    `Vertices`[3] = $VoxelPoint-$ new $Vector3d(-u, -v, +w)$;
    `Vertices`[4] = $VoxelPoint-$ new $Vector3d(+u, -v, +w)$;
    `Vertices`[5] = $VoxelPoint+$ new $Vector3d(-u, -v, +w)$;
    `Vertices`[6] = $VoxelPoint+$ new $Vector3d(+u, -v, +w)$;
    `Vertices`[7] = $VoxelPoint+$ new $Vector3d(+u, +v, +w)$;
    `Vertices`[8] = $VoxelPoint+$ new $Vector3d(-u, +v, +w)$;
Create Faces **begin**
    `List<MeshFace>` $Faces$ = $IntersectionTarget.Faces$; ;
    $Faces.AddFace(0, 1, 2)$;
    $Faces.AddFace(0, 6, 5)$;
    $Faces.AddFace(0, 4, 1)$;
    $Faces.AddFace(0, 8, 7)$;
    $Faces.AddFace(0, 3, 4)$;
    $Faces.AddFace(0, 7, 6)$;
    $IntersectionTarget.Vertices.AddVertices(Vertices)$;
    $IntersectionTarget.Faces.AddFaces(Faces)$;
return $IntersectionTarget$;

Algorithm 6: BBoxToVoxels

Input : bounding box adjusted to voxel size ($ZBBox$), voxel size as `Vector3d` ($vSize$)
Output: a bounding box filled with voxels
Initialize **begin**
    `int` $i = 0, j = 0, k = 0$;
    `int` $Imax = [|ZBBox.Diagonal.X/Vsize.X|]$;
    `int` $Jmax = [|ZBBox.Diagonal.Y/Vsize.Y|]$;
    `int` $Kmax = [|ZBBox.Diagonal.Z/Vsize.Z|]$;
    `List<Point3d>` $VoxelPoints$ = new `List<Point3d>`();
Create Voxels **begin**
    for $0 \leq k < Kmax$ **do**
        for $0 \leq j < Jmax$ **do**
            for $0 \leq i < Imax$ **do**
                `Point3d` $RelPoint$ = new `Point3d`$(i \times vSize.X, j \times vSize.Y, k \times vSize.Z)$;
                $RelPoint = RelPoint+$ new `Vector3d`$(ZBBox.Min) + 0.5 \times VSize$;
                $VoxelPoints.Add(RelPoint)$;
return $VoxelPoints$;

extra measures and techniques.

7 Surface Voxelization (2D Inputs, 1D Targets)

Function VOXELIZESURFACE is used for topological voxelization of 2D surfaces, which are represented as TIN or triangular polygon mesh objects. Note that other surfaces such as NURBS surfaces are also approximated as such for rendering purposes and so they can be the input of this method. The algorithm works by iterating over triangle faces of the surface in question by checking whether they intersect with the relevant intersection target of each voxel that could possibly be in resulted 3D raster. The points that possibly intersect with an intersection target are those whose distance to the input object is less than half of the length of the voxel size vector, i.e. a virtual vector comprised of the voxel size in X, Y and Z directions. Intersection between a connectivity target and the input surface is eventually determined by checking intersections between individual line segments and triangles. A standard algorithm for determining whether a triangle and a line intersect has been used for this purpose (see Curve Voxelization (1D Inputs, 2D Targets)).

Users can choose a level of connectivity (6 or 26), because of which a corresponding connectivity target will be chosen here to produce appropriate results. The pseudo code is described in Algorithm 7.

The function 2D_26_INTERSECTIONTARGET described by the pseudo code in Algorithm 8 creates 2D mesh ensuring 26 connectivity.

The function 2D_26_INTERSECTIONTARGETDI described by the pseudo code in Algorithm 9 creates a 2D mesh ensuring 26 connectivity with a diagonal target.

The function 2D_6_INTERSECTIONTARGET described by the pseudo code in Algorithm 10 creates a 2D mesh ensuring 6 connectivity with outline target.

Similar to the case of mesh targets for curve inputs, it is notable that each diagonal line target is only composed of 8 lines (which can also be made with 4 lines) whereas the outline target composed of cube edges is consisted of 12 lines. This means that the diagonal target is 12/8 = 1.5 times faster for computing voxels in the output compared to the cube outline edges. This is because for each voxel to be included in the output in the case of cube outline edges there must be 12 intersections solved in the worst case whereas the maximum number of line triangle intersections in the case of diagonal targets is 8. However, it might be advantageous to work with the cube edges because in that case 2D scan conversions parallel to X, Y, or Z axes might be combined (in a Meta Intersection Target) to find out voxels in the output. This is important because 2D scan conversion is a standard procedure for graphical processors that pixelate vector inputs for visualization on 2D digital monitors. Treating this matter in depth falls outside of the scope of this paper though.

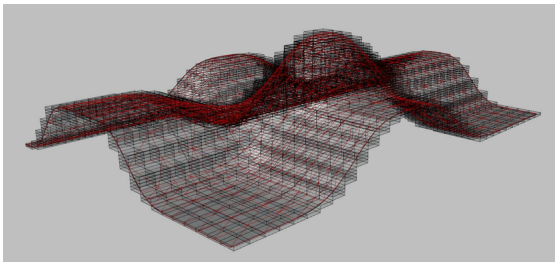


Figure 8: A surface (triangular polygon mesh) voxelized with connectivity level 26

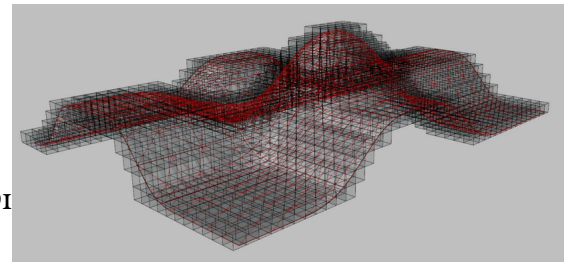


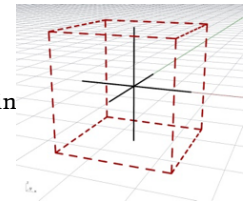
Figure 9: A surface (triangular polygon mesh) voxelized with connectivity level 6

Algorithm 7: VOXELIZESURFACE

Input : `int` *connectivity*, `mesh` *surfaces*, `double` *sx*, `double` *sy*, `double` *sz*
Output: `list<3D_points>` *voxels*

foreach *surface in surfaces* **do**
    form a bounding box in $\mathbb{R}^3$ for curve;
    define voxels as list of 3D points;
    `/* generate a bounding box equal or larger than the current bounding box in` $\mathbb{Z}^3$ `(c.f., Algorithm 2) */`
    $ZBBox$ = RBBox_to_ZBBox($RBBox, VSize$);
    define voxel size as $vSize = [sx, sy, sz]$;
    `/* Pseudo code described by Algorithm 6 */`
    $VoxelPoints$ = BBoxToVoxels($ZBBox, VSize$);
    `/* voxel centre points closer than half of the length vSize to the surface which possibly intersect with the input surface */`
    $near\_voxel\_points = VoxelPoints.FindAll(lambda|lambda.DistanceTo(Surface) < |VSize|/2)$
    foreach *voxel_point in near_voxel_points* **do**
        if *connectivity* = 26 **then**
            `/* Pseudo code described by Algorithm 8 */`
            $intersectionTarget$ = 2D_26_intersectionTarget($voxel\_point, vSize$);
        else if *connectivity* = 6 **then**
            `/* Pseudo code described by Algorithm 10 */`
            $intersectionTarget$ = 2D_6_intersectionTarget($voxel\_point, vSize$);
        else
            throw exception and prompt: "other connectivity levels undefined";
            if TriangleIntersectsLine($intersectionTarget, surface$) **then**
                $voxels.add(voxel\_point)$;
return voxels;

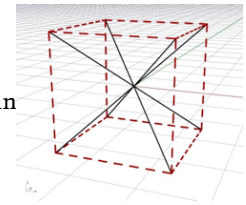
Algorithm 8: 2D_26_intersectionTarget



Input : voxel size as `Vector3d`
Output: an intersection target for 2D input ensuring 26 connectivity as a line array as in the figure

Initialize **begin**
    `Line[]` $IntersecTarget$ = new `Line`[3];
    `Point3d[]` $Vertices$ = new `Point3d`[6];
    `double` $u = (vSize.X/2), v = (vSize.Y/2), w = (vSize.Z/2)$;
Create Vertices **begin**
    $Vertices[0] = VoxelPoint +$ new `Vector3d`$(+u, 0, 0)$;
    $Vertices[1] = VoxelPoint +$ new `Vector3d`$(0, +v, 0)$;
    $Vertices[2] = VoxelPoint +$ new `Vector3d`$(0, 0, +w)$;
    $Vertices[3] = VoxelPoint +$ new `Vector3d`$(-u, 0, 0)$;
    $Vertices[4] = VoxelPoint +$ new `Vector3d`$(0, -v, 0)$;
    $Vertices[5] = VoxelPoint +$ new `Vector3d`$(0, 0, -w)$;
Create Edges **begin**
    for $0 \leq i \leq 2$ **do**
        $IntersecTarget[i]$ = new `Line`$(Vertices[i], Vertices[i+3])$;
    return $IntersectionTarget$;

Algorithm 9: 2D_26_INTERSECTIONTARGETDIAGONAL



Input : voxel size as `Vector3d`
Output: an intersection target for 2D input ensuring 26 connectivity as a line array as in the figure

```
Initialize begin
    Line[] IT = new Line[4];
    Point3d[] VX = new Point3d[8];
    double u = (vSize.X/2); v = (vSize.Y/2); w = (vSize.Z/2);
Create Vertices begin
    VX[0] = VoxelPoint+ new Vector3d(+u, +v, +w);
    VX[1] = VoxelPoint+ new Vector3d(−u, +v, +w);
    VX[2] = VoxelPoint+ new Vector3d(−u, −v, +w);
    VX[3] = VoxelPoint+ new Vector3d(+u, −v, +w);
    VX[4] = VoxelPoint+ new Vector3d(−u, −v, −w);
    VX[5] = VoxelPoint+ new Vector3d(+u, −v, −w);
    VX[6] = VoxelPoint+ new Vector3d(+u, +v, −w);
    VX[7] = VoxelPoint+ new Vector3d(−u, +v, −w);
Create Edges begin
    IT[0] = new Line(VX[0], VX[4]);
    IT[1] = new Line(VX[1], VX[5]);
    IT[2] = new Line(VX[2], VX[6]);
    IT[3] = new Line(VX[3], VX[7]);
return IT;
```

8 Notes on Implementation and Application

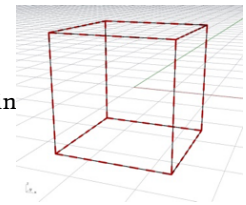
Our implementation of point cloud voxelization algorithm and curve voxelization algorithm are using rhinocommon.dll[7] that is a library provided by McNeel[8], the vendor of Rhino3D[9]. The dependencies however are mostly because of using geometric type definitions such as Point3D, Line and Mesh, which can be replaced straightforwardly. An example code without such dependencies is the surface voxelization code provided in the repository. Codes dependant on Rhinocommon can be complied and tested in an environment such as Grasshopper3D[10] also developed by McNeel.

As can be seen in curve and surface voxelization algorithms, we are currently forming a 3D grid full of voxels for a bounding box of the input object. This can be problematic in case this bounding box is big. In case of surfaces, it is more efficient to iterate over individual triangle faces to avoid this problem. Similarly, it will be more efficient to iterate over individual line segments in a polyline approximating the input curve compared to iterating over curve objects directly[11]. This, however, means that there will be many duplicate voxels in the raster output. To avoid this problem, a 3D raster data model similar to the model implemented in the voxelization algorithm for point clouds needs to be implemented to keep track of voxels already created and to ensure there will not be duplicates. Such a 3D raster data model will be possibly very space efficient as it should only store one bit per voxel. Such an implementation then will bring other advantages such as the

[7] `http://4.rhino3d.com/5/rhinocommon/` See also: `https://github.com/mcneel/rhinocommon`
[8] `http://www.en.na.mcneel.com/`
[9] `https://www.rhino3d.com/`
[10] `http://www.grasshopper3d.com/`

[11] This is not the case in our current implementation now. We are iterating over 'generic curve objects' and use a function of Rhinocommon to intersect a curve and a mesh. However, deep inside, this function is also based on intersecting line segments and triangles. Note that free-form curve objects such as NURBS curves are rendered as polylines with fine line segments.

Algorithm 10: 2D_6_intersectionTarget



Input : voxel size as `Vector3d`
Output: an intersection target for 2D input ensuring 6 connectivity as a line array as in the figure

Initialize **begin**
    `Line[]` IT = new `Line`[12];
    `Point3d[]` VX = new `Point3d`[8];
    `double` $u = (vSize.X/2); v = (vSize.Y/2); w = (vSize.Z/2)$;
Create Vertices **begin**
    $VX[0] = VoxelPoint +$ new `Vector3d`$(+u, +v, +w)$;
    $VX[1] = VoxelPoint +$ new `Vector3d`$(-u, +v, +w)$;
    $VX[2] = VoxelPoint +$ new `Vector3d`$(-u, -v, +w)$;
    $VX[3] = VoxelPoint +$ new `Vector3d`$(+u, -v, +w)$;
    $VX[4] = VoxelPoint +$ new `Vector3d`$(-u, -v, -w)$;
    $VX[5] = VoxelPoint +$ new `Vector3d`$(+u, -v, -w)$;
    $VX[6] = VoxelPoint +$ new `Vector3d`$(+u, +v, -w)$;
    $VX[7] = VoxelPoint +$ new `Vector3d`$(-u, +v, -w)$;
Create Edges **begin**
    $IT[00]$ = new `Line`$(VX[0], VX[1])$;
    $IT[01]$ = new `Line`$(VX[1], VX[2])$;
    $IT[02]$ = new `Line`$(VX[2], VX[3])$;
    $IT[03]$ = new `Line`$(VX[3], VX[0])$;
    $IT[04]$ = new `Line`$(VX[0], VX[6])$;
    $IT[05]$ = new `Line`$(VX[6], VX[5])$;
    $IT[06]$ = new `Line`$(VX[5], VX[4])$;
    $IT[07]$ = new `Line`$(VX[4], VX[7])$;
    $IT[08]$ = new `Line`$(VX[5], VX[3])$;
    $IT[09]$ = new `Line`$(VX[4], VX[2])$;
    $IT[10]$ = new `Line`$(VX[1], VX[7])$;
    $IT[11]$ = new `Line`$(VX[6], VX[7])$;
    return IT;

ease of Boolean operations on multiple 3D raster objects such as Boolean operations (union, intersection, etc.) and mathematical morphology operations (erosion, dilation, gradient, etc.) as it will only involve bitwise operations, provided that every two 3D raster models are defined in reference to the same bounding box so that their $\mathbb{Z}^3$ representations are compatible. Then such a requirement imposes another requirement that regards the existence of many zeros in any such 3D raster data model. Consider a house model voxelized in reference to a bounding box over the whole neighbourhood or city. It is obvious that most of the voxels represented in the 3D array of bits are zero. This suggests implementation of a data structure like sparse matrix so that such 3D raster models can be compressed in memory.

What is interesting about a binary representation of a voxel collection (as 3D raster) is that the same way 2D raster data models (2D images) can be processed and analysed, 3D raster models (3D images if you like) can be processed and analysed. What is common in both cases is the binary representation of coloured picture elements (pixels) or coloured volume elements (voxels). In short, we can enter a completely new world of possibilities inspired by digital image processing, merely by representing voxel grids as Boolean tensors (3D matrices).

9 Acknowledgements

The work reported here was partly funded by the Netherlands eScience Center (NLeSC) project, Big Data Analytics in the Geo-Spatial Domain (https://www.esciencecenter.nl/?/project/big-data-analytics-in-the-geo-spatial-domain), project code: 027.013.703. We hereby thank the reviewers for their constrcutive and detailed comments, which improved the paper significantly.

10 Additional Information and Supplementary Materials

10.1 Rasterizing CAD models

Rasterization of a building model (Bentley building) can be done using the C# version of the code in Rhinoceros; as shown in the following images (see our exemplary immplementation of the C# source for Rhino3D and Grasshopper 3D here: https://github.com/NLeSC/geospatial-voxels/tree/master/software/voxelGen/voxelizationTools_Rhino_and_GH):

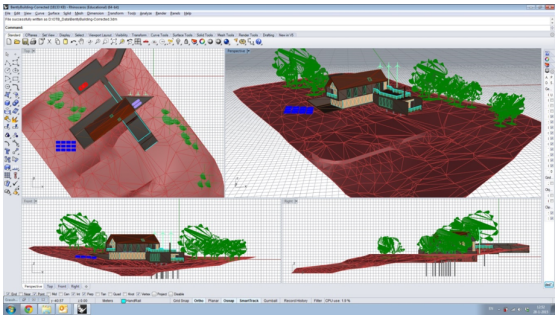


Figure 10: An IFC building mode provided by Bentley Systems, imported as OBJ generated from original BIM model, corrected and coloured to represent objects of different semantics

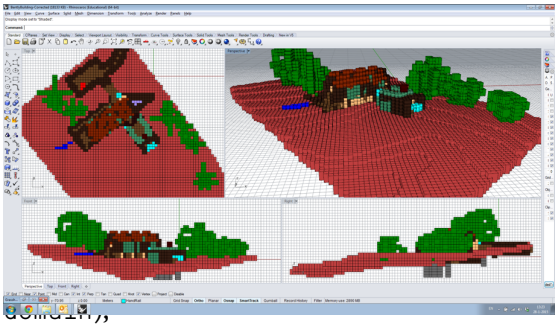


Figure 11: Topological Rasterization with Semantics from a BIM model [0.5 by 0.5 by 0.5 m]

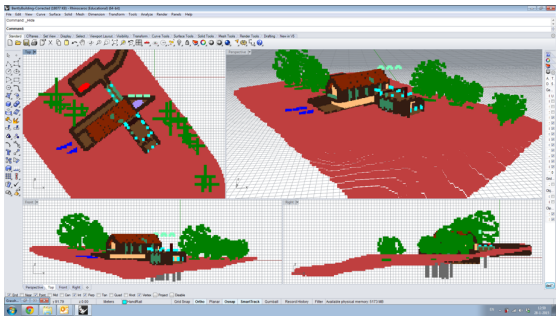


Figure 12: Topological Rasterization with Semantics from a BIM model [0.4 by 0.4 by 0.4 m]

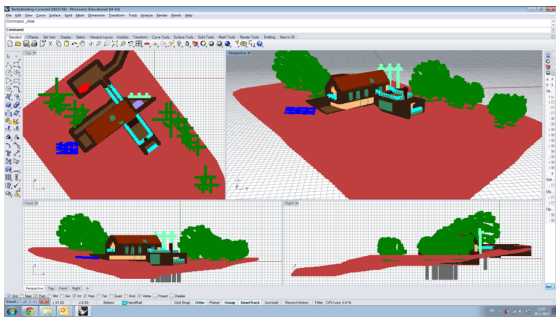


Figure 13: Topological Rasterization with Semantics from a BIM model [0.1 by 0.1 by 0.1 m]

10.2 Rasterizing CityGML LOD2 models (GIS)

In voxelizing CityGML files the main issues are defining a proper schema both for raster 3D and voxels in order to keep semantic information such as the ones modelled in a CityGML. Both of these issues and also the matter of ensuring uniqueness of voxels (avoiding duplicates when joining multiple voxel collections) requires further investigation and implementation of measures such as a 3D binary array to keep track of visited voxels as implemented in our point cloud voxelization algorithm. The following images are test results of the last version of the TUD_voxelizer code in C#, presented in the aforementioned repository. Visualizations are done in CloudCompare.

Figure 14: The area voxelized using CItyGML models provided by the municipality of Rotterdam. See the data sets and a sample voxelated result at: `https://github.com/NLeSC/geospatial-voxels/tree/master/software/voxelGen/data`

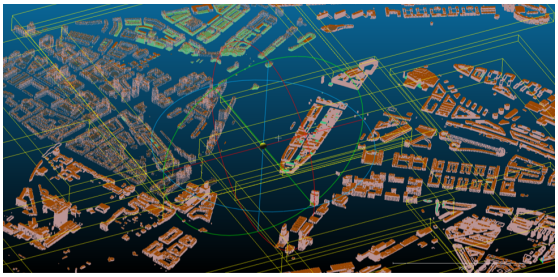


Figure 15: Rotterdam CityGML, Noordereiland, and neighbourhoods around it; voxelized in different resolutions, zoomed out.

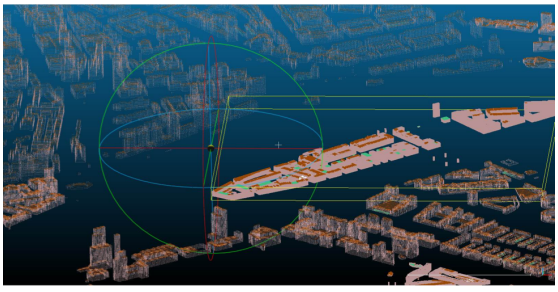


Figure 16: Rotterdam CityGML, Noordereiland and neighbourhoods around it; Noordereiland selected

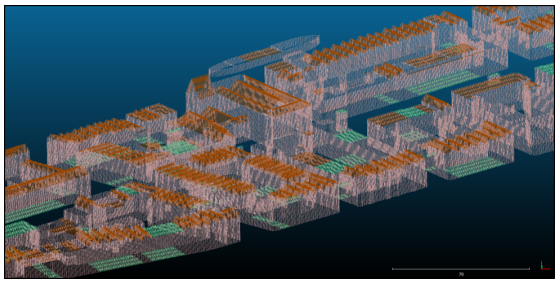


Figure 17: Rotterdam CityGML, Noordereiland, $0.2 \times 0.2 \times 0.2$; zoomed in.

10.3 Background and Literature Review

In this section, we briefly compare (qualitatively) our developments with other 3D rasterization algorithms for curves and surfaces. Since the geospatial objects are commonly represented by multiple surfaces, the review is specifically on voxelizing surfaces. Use of volumetric primitives (cone, cylinder, sphere) to represent buildings, bridges, e.g. in BIM (Building information modelling) are excluded. Research discussing voxelisation of such primitives can be found in [Fang and Chen, 2000] or [Jones and Satherley, 2000].

Two general approaches can be distinguished in voxelisation: object rasterization (e.g. Topological Voxelization) and scan-conversion (e.g. 3D Scan Conversion) in the body of existing methods. Object rasterization concentrates on the object of interest following two steps: boundary rasterization (and possibly interior filling). The scan conversion focuses on the bounding box of the objects and decides that in a given raster volume which voxels get what kind of value. Most of the presented methods in this group are based on extensions of the well-known scan-conversion method, which is well studied and commonly used in the field of 2D computer graphics. After an initial investigation of methods in a rather historical order Table 5, we chose the following methods to implement and compared them according to our quality criteria:

- 3D Scan Conversion [Kaufman and Shimony, 1986]: i.e. extended 2D scan conversion into 3D. This method is very efficient but 6-Connectivity cannot be reached, therefore 26-separability not assured, difficult to generalize the method to different geometric primitives.
- Topological Voxelization [Laine, 2013] NVIDIA: based on the concept of connectivity target, very elegant mathematically; which can be efficiently implemented. The topological approach focuses on matters of Boolean nature such as connectivity, separability, intersections more explicitly compared to former geometrical approaches.

If there is any interest in doing the same comparison, we refer the reader to our implementation of both methods at: `https://github.com/NLeSC/geospatial-voxels/tree/master/software/voxelGen`. However, giving full account of our implementation of 3D scan conversion falls out of the scope of this paper.

References

Ken Arroyo Ohori, Hugo Ledoux, and Martijn Meijers. Validation and automatic repair of planar partitions using a constrained triangulation. *Photogrammetrie, Fernerkundung, Geoinformation*, 5:613–630, October 2012.

Daniel Cohen-Or and Arie Kaufman. Fundamentals of surface voxelization. *Graphical Models and Image Processing*, 57(6): 453–461, November 1995.

Max J. Egenhofer and John R. Herring. A mathematical framework for the definition of topological relations. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, pages 803–813, 1990.

Shiaofen Fang and Hongsheng Chen. Hardware accelerated voxelisation. *Computers & Graphics*, 24(3):433–442, June 2000.

Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *VVS '98 Proceedings of the 1998 IEEE symposium on Volume visualization*. ACM Press, 1998.

Table 5: A qualitative comparison of voxelization algorithms

Year	1997	1995	1998	2003	2010	2013
Authors	Kaufman and Shimony [1986]	Cohen-Or and Kaufman [1995]	Huang et al. [1998]	Varadhan et al. [2003]	Schwarz and Seidel [2010]	Laine [2013]
Method	3D Scan Conversion (lines)	3D Scan Conversion (surfaces)	Distance Based (Spheres)	Max-Norm distance	Parallel voxelization using GPUs	Topological Voxelization
Explicit topology control?	Yes only for 26- connectivity/6 separation	Yes only for 26 connectivity/6 separation	Yes	N/A	Yes	Yes
Notes	6-connected results cannot be produced	6-connected results cannot be produced, multiple algorithms for different types of surfaces	Is not minimal and it is sensitive to tessellation	Produces a 'cover' voxel set, can be eventually used for improving efficiency of other methods	Using an unnecessarily large target results in voxels that are not required for 6-separation, thus not minimal	Elegant method that comes with mathematical proofs for topological properties; algorithm not clearly defined

Mark W. Jones and Richard Satherley. Voxelisation: Modelling for volume graphics. In *Vision, Modeling, and Visualisation*, 2000.

Arie Kaufman and Eyal Shimony. 3D scan-conversion algorithms for voxel-based graphics. In *I3D '86 Proceedings of the 1986 workshop on Interactive 3D graphics*, pages 45–75. ACM Press, 1986.

Samuli Laine. A topological approach to voxelization. *Computer Graphics Forum*, 32 (4):77–86, July 2013.

Jiyeong Lee. 3D data model for representing topological relations of urban features. In *Proceedings of the 21st Annual ESRI International User Conference*, 2001.

Jildou Louwsma, Sisi Zlatanova, Ron van Lammeren, and Peter van Oosterom. Specifications and implementations of constraints in GIS—with examples from a geo-virtual reality system. *GeoInformatica*, 10(4):531–550, 2006.

James R. Munkres. *Elements of Algebraic Topology*. Perseus Books, 1984.

OGC. *OGC IndoorGML*. Open Geospatial Consortium, December 2014.

Simon Pigot. Topological models for 3D spatial information systems. In *Proceedings of the 10th International Symposium on Computer Assisted Cartography (AUTOCARTO 10)*, pages 369–391, 1991.

Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics (TOG) — Proceedings of ACM SIGGRAPH Asia 2010*, 29(6), 2010.

Gokul Varadhan, Shankar Krishnan, Young J. Kim, Suhas Diggavi, and Dinesh Manocha. Efficient max-norm distance computation and reliable voxelization. In *SGP '03 Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. ACM Press, 2003.

Sisi Zlatanova. On 3D topological relationship. In *Proceedings of the 11th International workshop on Database and Expert System Applications (DEXA 2000)*, pages 913–919, 2000.