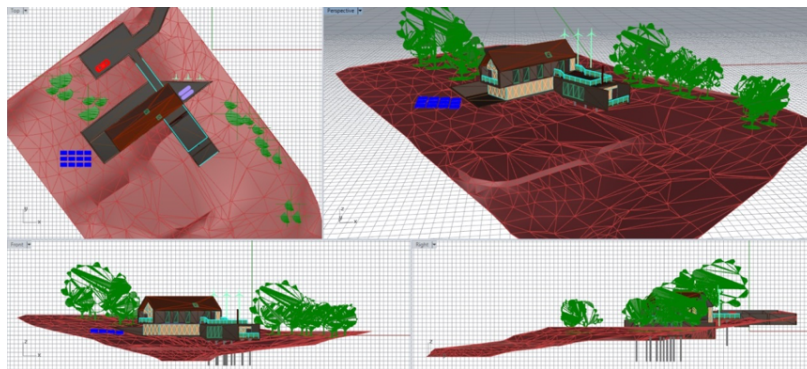


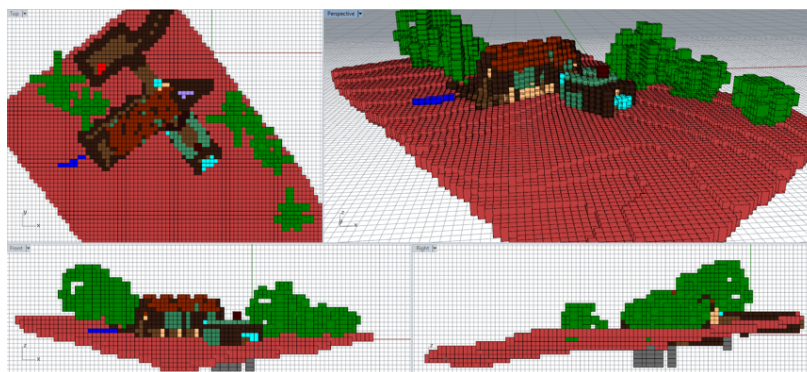
Voxel models, which are the 3D equivalent of 2D rasters, are a common way to store 3D models of the built environment using a regular 3D grid (Figure 1.1). Much like rasters in 2D, they have inherent limits in precision based on the grid size that is used and can easily grow to very large sizes in terms of memory, especially with a small grid size and when compression is not used.

At the same time, voxel models are easy to use and understand, and algorithms to process them are typically much simpler than those using other representations, which also makes them more reliable, robust and easy to parallelise. These characteristics make voxels an important and widely used representation to process 3D information in general.

- 1.1 Exhaustive enumeration models . . . . . 1
- 1.2 Hierarchical subdivision models . . . . . 3
- 1.3 Voxelisation . . . . . 4
- 1.4 Exercises . . . . . 9
- 1.5 Notes and comments . . . . . 9



(a)



(b)

Figure 1.1: (a) A mesh model of a house with surrounding terrain and trees and (b) a corresponding voxel model with the same elements.

## 1.1 Exhaustive enumeration models

Voxels might appear to be quite a unique data model in terms of 3D representations, but they are actually only the most used among a type of related representations, which are together usually referred to as *exhaustive enumeration*. The specifics of these data models differ, but in

exhaustive enumeration

general they represent objects by:

voxel domain

voxel cells

1. **defining the shape of a domain** in which the objects to be represented fit, or alternatively in which the region of interest of a field fits, eg a bounding box defined by their minimum and maximum coordinates along each axis;
2. **dividing the domain** using a structure of many *cells*, usually following a regular or semi-regular pattern that can be defined programmatically (as opposed to explicitly representing the shape of each individual cell), eg a grid defined by the number of cells along each axis;
3. **specifying a well-defined order** passing once through each cell of the subdivision, usually also programmatically (as opposed to explicitly numbering each cell), eg the order and direction of iteration of the axes in a grid;
4. **labelling each cell** with values that indicate the object(s) that are in it, or in the case of fields, the values of variable(s) at that location. The values can then be *encoded linearly* using the order defined in the previous step.

We can thus say that what is represented in an exhaustive enumeration is usually composed of four elements: (i) a set of rules defining the shape of a domain, (ii) a set of rules on how to divide the domain into cells, (iii) a set of rules that define an order of the cells, and (iv) an encoded linear representation that represents objects or values for all cells. However, out of these four elements, the first three are sets of rules that are generally very simple, and thus they are stored encoded in a minimal way or not at all (ie only implied by the context). For instance, the rules might be part of the specification of a particular data format.

Based on these standard characteristics, we can see that exhaustive enumeration representations use space differently from other data models. In most geometric representations, much (or most) of the space and complexity of a data structure is devoted to creating a custom structure that individually describes the shape of the objects being represented. By contrast, in exhaustive enumeration, objects' shapes are instead approximated using simple rules on a predefined structure, and the vast majority of the space is thus devoted to specifying which objects are present in which cells (or the values of a field in each cell).

That being said, the statements described above—which in the steps correspond to the actions that are done for a standard regular 3D grid with voxels—can all differ substantially. By analysing different possibilities at each step, it is easy to see how the approach can be adapted and extended to form other types of representations. For instance, consider the following example, which is arbitrarily chosen to be completely different from a typical voxel grid.

We can start by describing a space using an alternative method, eg a b-rep representation of a domain with an arbitrarily complex shape. Cells could then be specified using a *constrained Delaunay tetrahedralisation* of the domain (using predefined rules for the addition of Steiner points). The order of the cells could be specified based on the lexicographical order of the vertices of each tetrahedron. Finally, the values of each tetrahedral cell are then encoded linearly as in a grid.

While the previous example is perfectly possible, it is worth noting that exhaustive enumeration schemes are well-liked largely because of their simplicity, which means that simpler representations are usually preferred. Using a complex representation where the geometry is not trivial to compute on the fly (eg a CDT) thus defeats many of the advantages of the exhaustive enumeration approach.

Most examples that are found in practice are thus relatively minor variations of voxel grids. For instance, cells can have varying sizes according to their place in the grid (eg when more details are desired in a particular region), the domains of grids can be stretched in some directions (such that the domain is oblique), or the cells can be of different shapes (eg octahedra).

Among the variations of voxel models, **sparse voxel models** are used widely in practice and are thus worth describing in more detail here. These representations opt to encode only the voxels containing something (rather than all voxels in the domain). In order to do so, they usually specify simple objects consisting of: (i) a voxel position, eg using integer coordinates for its position along each axis, and (ii) the voxel's variables. While this is undoubtedly more space-intensive per voxel than the standard encode-all approach, it works well for 3D models consist of largely empty space, which occurs frequently in 3D city models and where the objects we want to represent do not fit neatly into a box-shaped domain.

sparse voxel model

It is also worth pointing out that most variations of voxel models can be processed with basically the same methods as standard voxel grids.

## 1.2 Hierarchical subdivision models

In addition to exhaustive enumeration, there are also related data models where the structure is not entirely predefined, but it is instead defined hierarchically using space-partitioning trees. The root of the tree thus refers to a predefined space that will be subdivided, which corresponds to the entirety of the domain that is represented in an exhaustive enumeration model. Each node then specifies a subset of the space defined by its parent node, and nodes (usually but not necessarily at the leaf level) are then labelled to specify the object(s) or value(s) present in the space represented by it.

Since different branches of a tree do not need to have the same depth, hierarchical subdivision models can have different resolutions in different parts of the model, and can thus adapt to the shape of the objects being represented. This allows them to act as more compact alternatives to exhaustive enumeration models in certain cases, usually where there are large objects that occupy many adjoining cells. Note however that the tree structure of a hierarchical representation can occupy a significant amount of space.

Hierarchical subdivisions are also a good way to encode the sparse models described in the previous section, where large areas of empty space will be efficiently represented by leaf nodes that are generally close to the root of the tree.

The most common structures used by hierarchical subdivision models are:

**octrees** subdivide space evenly along the  $x$ ,  $y$  and  $z$  axes into eight equal-size *octants*. They are analogous to quadtrees in 2D, which subdivide space evenly along the  $x$  and  $y$  axes into four equal-size quadrants.

**bintrees** are similar to octrees, but they subdivide space in halves along only one axis per node, then switching to a different axis for the next level of the tree, eg  $x$ , then  $y$ , then  $z$ , then  $x$  again, etc.

**$k$ -d trees** are similar to bintrees, but they subdivide space using an arbitrary plane per node, which can be defined by a single coordinate included in the node.

## 1.3 Voxelisation

voxelisation

The process through which other data models are converted into voxels is called *voxelisation*. It is analogous to rasterisation in 2D. In most cases, the data being voxelised consists of vector objects, either as a point cloud or a b-rep mesh. We will thus explain a method to voxelise 0D, 1D, 2D and 3D vector objects. In principle, it can be applied to arbitrary curves and surfaces, but in most instances they will be line segments (or polylines), as well as triangular and polygonal meshes.

### 1.3.1 Connectivity

When rasterising a curve in 2D, different algorithms aim to obtain a pixellated curve that is connected according to either 4-connectivity or 8-connectivity (Figure 1.2). These are as follows:

**4-connectivity** means that pixels are connected to their four horizontally and vertically adjacent neighbours.

**8-connectivity** means that pixels are connected their four 4-connected neighbours and to their four diagonally incident neighbours.

In 3D, the equivalent concepts are 6-connectivity, 18-connectivity and 26-connectivity. These are as follows:

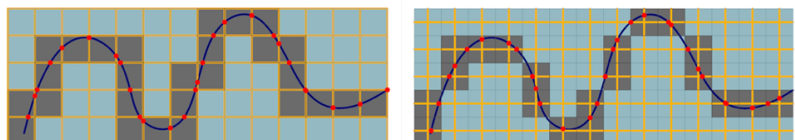
**6-connectivity** thus means that voxels are connected to their six adjacent neighbours (ie on their left, right, front, back, bottom and top).

**18-connectivity** means that voxels are connected to their six 6-connected neighbours and to their twelve incident neighbours that touch them diagonally along an edge (ie top left, top right, top front, top back, bottom left, bottom right, bottom front, bottom back, front left, front right, back left and back right).

6-connectivity

18-connectivity

26-connectivity



**Figure 1.2:** Rasterising a line to achieve 4-connectivity (left) and 8-connectivity (right). The algorithm uses line targets (orange) that are intersected with the curve.

**26-connectivity** means that voxels are connected to their eighteen 18-connected neighbours and to their eight incident neighbours that touch them diagonally along a vertex (ie top front left, top front right, top back left, top back right, bottom front left, bottom front right, bottom back left and bottom back right).

An alternative way to think about these connectivities is that they are defined based on the dimensionality of the common boundary of the pixels or voxels. 6-connectivity means that two neighbouring voxels have a common 2D face. 18-connectivity means that they have at least a common 1D edge (which covers having a common 2D face). 26-connectivity means that they have at least a common 0D vertex (which covers having a common 1D edge or 2D face).

18-connectivity is an interesting concept that shows that there is a consistent logic for every dimension, but it is not really used in practice. We will thus not discuss it further.

### 1.3.2 Intersections with targets (2D)

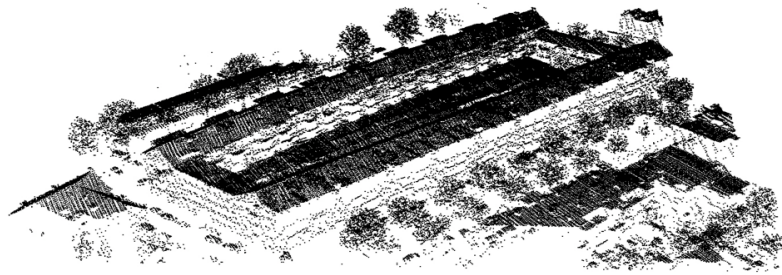
In the example from Figure 1.2, the pixellated curve is obtained by calculating intersections between the original 1D curve and a set of *intersection targets* that are 1D line segments. For 4-connectivity, the targets consist of the four line segments that bound every pixel. For 8-connectivity, the targets are line segments that bisect the pixel horizontally and vertically and their midpoints. The intersections with the targets give us a set of points, and the pixels in which these points are tell us the pixels that are part of the pixellated curve. When a point lies on an edge between two pixels or a vertex between four pixels, we consider that all of the pixels are part of the curve.

intersection target

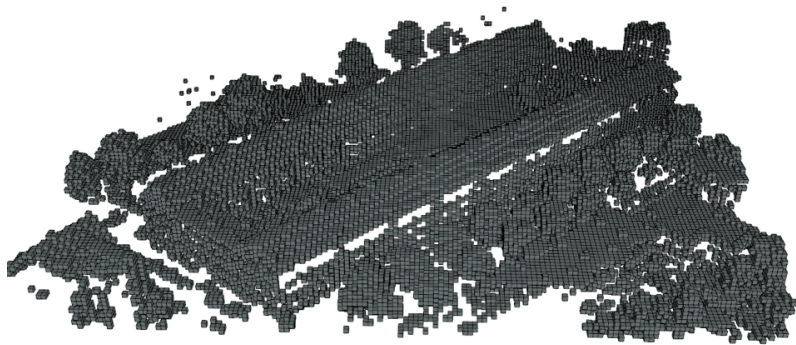
In order to understand the logic of the targets, it is important to consider two aspects: (i) where the intersections will lie and (ii) whether they will detect lines when they do not cross the midpoint of a pixel. For 4-connectivity, the targets simply detect when a line exits the pixel through the left, right, bottom or top edges on the *boundary of the pixel*. Since all intersections will be between pixels, the 2 or 4 pixels incident to the points will be part of the pixellated curve. For 8-connectivity, the targets detect when they pass through the middle of the pixel either vertically or horizontally, which happens in the *interior of the pixel*. Crucially, note that they might do not detect when a line cuts through a corner of the pixel without crossing its middle vertically or horizontally.

Having covered the rasterisation of a 1D curve, let us discuss the two other cases: rasterising 0D points and 2D areas. Since vector points are not connected, they do not need to be connected when rasterised either. Since areas are always connected, they should also be connected when rasterised. Connectivity is thus not an issue, which makes their rasterisation simpler.

An important observation for this method is that we used 1D targets to rasterise a 1D curve. In order to rasterise a set of 0D points, we would use intersections with 2D targets, of which the optimal choice would consist of the whole area of each 2D pixel. In order to rasterise a set of 2D areas, we would use intersections with 0D targets, of which the obvious choice



(a)



(b)

**Figure 1.3:** A point cloud (a) before and (b) after voxelisation. AHN data from Rotterdam.

is the midpoint of a pixel (although others are possible). It is possible to see a duality property here: in order to rasterise  $i$ -dimensional objects, we use  $(2 - i)$ -dimensional targets.

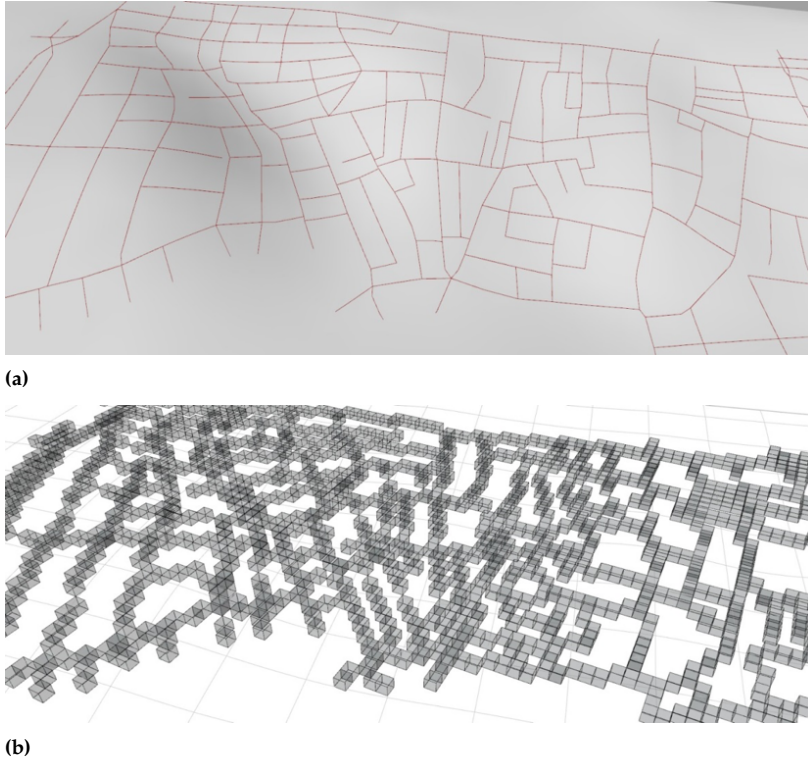
### 1.3.3 Intersections with targets (3D)

At this point, we should point out that the method described in the previous section is not the absolute fastest or the most common to rasterise objects in 2D. However, it is a method with good performance with a logic that works perfectly in 3D, which is the reason why we will now explain how it works for voxelisation.

Let us start backwards, with the equivalent duality property for voxelisation, which states that we can use  $(3 - i)$ -dimensional targets to voxelise  $i$ -dimensional objects. Using this formula directly, we can discuss the most obvious cases first: voxelising 0D points and 3D volumes, in which connectivity also does not matter.

In order to voxelise **0D points** (eg a point cloud), we can thus simply use 3D targets that consist of the whole voxel (Figure 1.3). That is, we can compute for each point which voxel it is in, or for each voxel the points that are in it. This is a trivial operation using ranges of  $x$ ,  $y$  and  $z$  coordinates.

Similar to the previous case, in order to voxelise **3D volumes**, we can use a 0D target with the midpoint of the voxel. The exact form of this operation depends on the input data. For instance, if we have tetrahedra



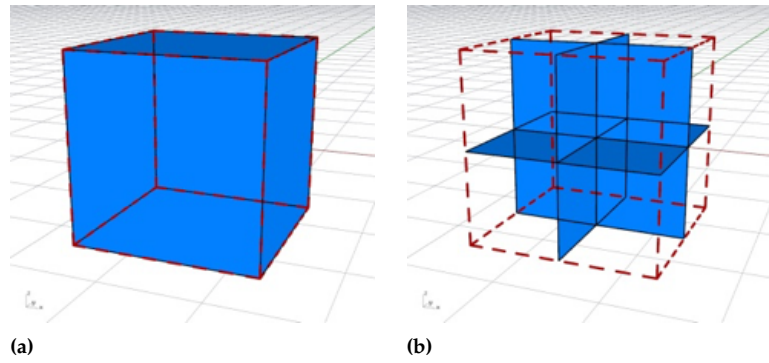
**Figure 1.4:** A set of lines (a) before and (b) after voxelisation. OpenStreetMap data from Istanbul.

as input, it would be a point in tetrahedron operation, which could be done using barycentric coordinates.

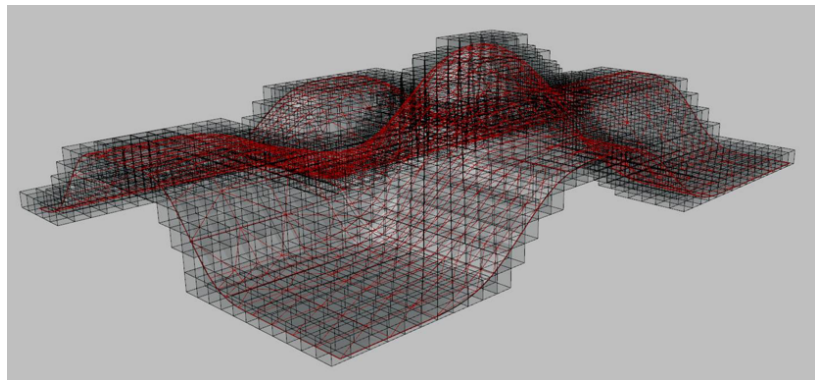
Now, let us discuss the more challenging cases: 1D and 2D objects. As with 1D curves in rasterisation, connectivity is important for these, so we will give targets that can be used in order to achieve 6-connectivity and 26-connectivity for each.

In order to voxelise **1D curves** with 6-connectivity (Figure 1.4), we could detect when these pass through the top, bottom, left, right, front or back faces of the voxel using 2D targets (Figure 1.5a). For 26-connectivity, we could detect when these pass through the middle of the voxel using three bisecting faces (Figure 1.5b).

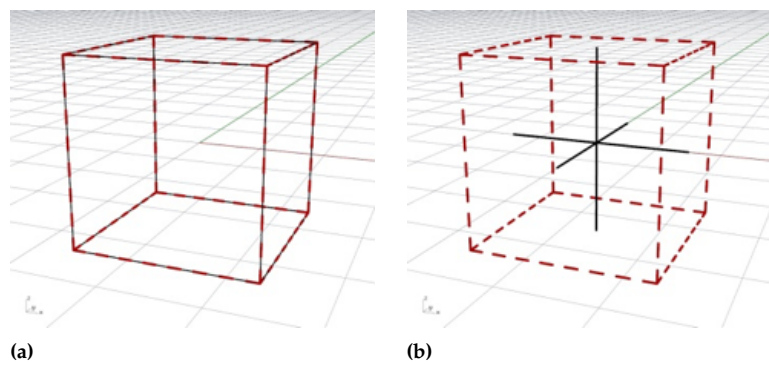
Now, in order to voxelise **2D surfaces** (Figure 1.6) with 6-connectivity, we can use 1D targets that detect when we pass through any of the 12 edges on the boundary of the voxel (Figure 1.7a). For 26-connectivity, we can use 1D targets that detect when we pass through the middle of the voxel (Figure 1.7b).



**Figure 1.5:** Intersection targets (blue) for 1D curves for (a) 6-connectivity and (b) 26-connectivity.



**Figure 1.6:** Voxelising a surface



**Figure 1.7:** Intersection targets (black lines) for 2D surfaces for (a) 6-connectivity and (b) 26-connectivity.



## 1.4 Exercises

1. Can you devise a formula to compare the space occupied by:
  - a) encoding all voxels in a grid linearly
  - b) using a sparse encoding with individual voxels
  - c) using a sparse encoding with an octree
2. Can you think of cases where the rasterisation targets for 1D lines do not work? Hint: think of short curves.
3. What kind of connectivity is used in the example of Figure 1.4?

## 1.5 Notes and comments

Voxels are widely used in areas other than geographic information. For instance, both medical magnetic resonance (MRI) and computer tomography (CT) scans produce voxel models. Physical simulations also use voxels since many calculations are easy to do using regular grid structures, eg finite-element analysis. Games sometimes use voxels as well, both for calculations and to render graphics. It is worth noting that many of the techniques developed in these fields are just as applicable to geographic information as well.

4D grids using 3D+time are also sometimes used, both in geographic information and elsewhere. Some of the earliest papers to mention this are: Mason et al. (1994), who implemented a system using a 4D grid of ocean temperatures with support for interpolation and generalisation operations, and Bernard et al. (1998), who implemented a 4D grid of atmospheric variables (eg temperature, wind or pollution), which can be used for simulations.

A common use of the representations covered here, especially voxel grids and octrees, is spatial indexing. Cells can thus be used to store other kinds of data, eg ids of objects, memory addresses with data, or a subset of a point cloud.

The original paper describing quadtrees is Finkel and Bentley (1974), whereas that for octrees is Meagher (1980). Bintrees (Samet and Tamminen, 1985) are an alternative that split dimensions alternately rather than all at once. If you are curious about more types of trees used in hierarchical subdivisions, have a look at the section titled 'Spatial data partitioning trees' in this Wikipedia template: [https://en.wikipedia.org/wiki/Template:CS\\_trees](https://en.wikipedia.org/wiki/Template:CS_trees).

The voxelisation algorithm covered here is described by Laine (2013), although it might be easier to understand the implementation described in Nourian et al. (2016). Alternative targets to the ones described in this lesson are shown in both papers.

## Bibliography

- Bernard, L., B. Schmidt, and U. Streit (1998). AtmoGIS — Integration of atmospheric models and GIS. *Proceedings of the 8th International Symposium on Spatial Data Handling*. Ed. by T. Poiker and N. Chrisman.
- Finkel, R. and J. Bentley (1974). Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica* 4.1, pp. 1–9.
- Laine, S. (2013). A Topological Approach to Voxelization. *Computer Graphics Forum* 32.4, pp. 77–86.
- Mason, N. C., M. A. O’Conaill, and S. B. M. Bell (1994). Handling four-dimensional geo-referenced data in environmental GIS. *International Journal of Geographical Information Systems* 8.2, pp. 191–215.
- Meagher, D. (1980). *Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Tech. rep. Rensselaer Polytechnic Institute.
- Nourian, P., R. Gonçalves, S. Zlatanova, K. Arroyo Ohori, and A. V. Vo (Jan. 2016). Voxelization Algorithms for Geospatial Applications: Computational methods for voxelating spatial datasets of 3D city models containing 3D surface, curve and point data models. *MethodsX* 3. ISSN: 2215-0161, pp. 69–86.
- Samet, H. and M. Tamminen (1985). Bintree, CSG trees, and time. *SIGGRAPH ’85*. Ed. by P. Cole, R. Heilman, and B. A. Barsky. Vol. 19. 3. ACM, pp. 121–130.