Extruding building footprints to create topologically consistent 3D city models

Hugo Ledoux and Martijn Meijers
Delft University of Technology (OTB—section GIS Technology)
Jaffalaan 9, 2628BX Delft, the Netherlands
{h.ledoux-b.m.meijers}@tudelft.nl

February 23, 2009

One of the simplest methods to construct a 3D city model is to extrude building footprints, to obtain "block-shaped" buildings. While the method is well-known and easy to implement, if the topological relationships between the footprints are not taken into account, the resulting city models will not necessarily be *topologically consistent*. As a result, the model will be of little use for most applications, besides visualisation that is. In this paper, we present a new extrusion algorithm to construct topologically correct 3D city models. It is conceptually simple and permits us to create city models in different formats (e.g. CityGML). We have implemented the algorithm, tested it for the creation of the model of our university campus and validated it by constructing the constrained Delaunay tetrahedralization.

1 Introduction

The Open Geospatial Consortium (OGC) has recently adopted CityGML (OGC, 2008) as a standard for representing three-dimensional (3D) city models. CityGML, a GML-based (Geography Markup Language) format, "not only represents the shape and graphical appearance of city models but specifically addresses the object semantics and the representation of the thematic properties, taxonomies and aggregations" (Kolbe, 2008). It also offers a multi-scale representation since five levels of details (LODs) for the same city can be stored: from LOD0 where only the terrain is stored, to LOD4 where buildings have detailed roofs structures, windows, rooms and even pieces of furniture. While the adoption of an international standard will certainly foster the use of CityGML by practitioners and also foster the development of new tools to create and process city models, it should also be said that at this moment the creation of 3D city models is still very much a problem in practice. Many 3D data from diverse sources are being collected (photogrammetry, LIDAR, GPS, terrestrial laserscanning, etc.), but not enough tools to process and structure these data are available. Most of the available models have indeed been built semi-automatically, and for the ones with high LODs heavy manual intervention is often the norm.

We discuss in this paper one way to automatically construct a 3D city model: by extruding the building footprints (ground plans) to create "block-shaped" buildings. This representation of a city, where roofs have no structure and are formed by horizontal planes, is referred to as LOD1 in CityGML. This is arguably the simplest way to create 3D city models as footprints of buildings are (usually) readily available, and the height at which buildings are extruded can be obtained by different means, although airbone laser scanners are nowadays used (Rottensteiner et al., 2005). Here, one might say that the extrusion task is an easy one. We argue that it is indeed easy if the only thing you want to do with your city model is to look at it! As explained in Section 2, many commercial software can extrude polygons to obtain buildings, but the results will in most cases only *look* nice, but not be *topologically consistent*. In a nutshell, that means that the result of an extrusion contains for example duplicate points, overlapping faces, faces intersecting where there are no points, etc. (we define formally topological consistency in Section 3). Figure 1 illustrates a simple case where five building footprints are being extruded. If the adjacency relationships between the building footprints are not taken into account

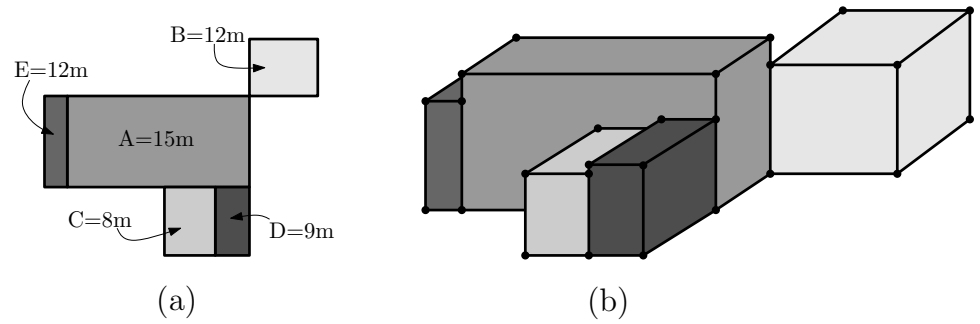

Figure 1: **(a)** Top view of 5 buildings A, B, C, D and E, and their respective elevation. **(b)** Perspective view of the result, obtained by extrusion.

while extruding (if the five are "independently" extruded), it is easy to see that the resulting model will not be consistent. And if one wants to *process* and *manipulate* such a city model (using the topological relationships between the spatial objects in 3D), and not only visualise it, then all these inconsistencies have to be fixed. Examples of applications where consistent city models are needed are plenty: noise modelling in 3D (CityGML has an application extension for noise mapping), flood modelling (Schulte and Coors, 2008), 3D navigation (Lee and Zlatanova, 2008), disaster management (Kolbe et al., 2008) and urban planning (Köninger and Bartel, 1998).

In this paper, we present a new algorithm to construct topologically consistent 3D city models by extrusion of building footprints. The algorithm, which is described in Section 4, is conceptually simple and straightforward to implement. It takes as input a topologically consistent dataset in 2D, and can output to different formats, including CityGML. We have put strict requirements on the input because cleaning a 2D dataset is a straightforward task for which many tools exist. In 3D, the same task is titanic, for, to the best of our knowledge, there are simply no tools available. Furthermore, we report in Section 5 on our implementation of the algorithm to create the city model of the campus of our university (TU Delft). In that section, we also describe the tools we used to obtain topological consistency out of a "spaghetti" input, show examples of the CityGML output, and describe how we validated our approach by creating a constrained Delaunay tetrahedralization of the model.

2 Related work

As mentioned in the Introduction, the "simple" extrusion of footprints, that is without considering other footprints, is a straightforward task and has been implemented in many commercial products, for instance Oracle Spatial 11g (Beinat et al., 2007), ArcGIS[1], or even Google Earth[2]. Each resulting building is valid, but there are no guarantees that a set of footprints will yield a topologically consistent city models.

Tse et al. (2005) extrude footprints and output topologically consistent models, but take a totally different approach. They start by constructing the constrained Delaunay triangulation of the footprints, and then extrude the buildings by always keeping a single triangulated surface for the whole area; the surface has vertical walls so their approach is referred to as "2.75D". With their method, many applications are possible since topological relationships between the buildings are explicitly stored, but it is unfortunately impossible to represent the vertical walls between two connected buildings (e.g. the five buildings in Figure 1 are stored without the internal walls) as the whole dataset must be represented by single surface.

One way to obtain topologically consistent 3D city models is to use constructive solid geometry (CSG). With that method, polyhedra are represented as Boolean combinations (union, intersection and difference) of simpler objects such as cylinders, cubes, spheres or pyramids. CSG objects are by definition topologically consistent, and it is possible to convert them a boundary representation (see Requicha (1982) for more details). Haala and Brenner (1999) create extruded 3D city models by first decomposing footprints into rectangles, and from these rectangles create CSG polyhedra, which are then combined to reconstruct the building. The main problem of that approach is that not every building footprint can be decomposed into rectangles; as can be seen in Section 5, some of the footprints of our campus have for instance circular shapes.

[1] `www.esri.com/arcgis`
[2] `earth.google.com`

Another approach to obtain topological consistency is that of *cell decomposition*, as explained in Haala et al. (2007). Each edge of a building becomes a half-space plane (an infinite vertical plane defined mathematically), and intersections of planes permits us to reconstruct the building. It is not clear how this method would scale up to big datasets since it appears that every half-space plane has to be tested with every other one in the dataset.

3 Topological concepts related to extrusion

This section introduces definitions and concepts needed for the next section. The concepts are related to topological relationships in 2D and 3D, and what happens when one creates a 3D dataset from a 2D dataset.

3.1 Topological concepts in 2D

Let M be a set of spatial objects in $\mathbb{R}^2$, the two-dimensional Euclidean space. Spatial objects in M are formed by three geometric primitives σ: (i) a point (0-dimensional object); (ii) a straight line segment (1-dimension), which is referred to simply as a line in the following; and (iii) a polygon (2-dimensional object). We call a sequence of lines connected by their end-points a *polyline*. Each point in a polyline is part of two lines, except the first and the last ones (we call them the *extremes* of the polyline).

Observe that a spatial object σ of dimensionality higher than 0 is formed by lower dimensionality primitives that define its boundary, denoted $\partial\sigma$. If σ is a line, then $\partial\sigma$ is formed by both end-points of σ; if σ is a polygon, then $\partial\sigma$ is formed by a polyline forming a *loop* (whose extremes are coincident).

We say that M is *topologically consistent* if the following rules are valid:

1. every line in M is formed by two points also in M;
2. the intersection of two lines σ_1 and σ_2, denoted $\sigma_1 \cap \sigma_2$, is either empty or is a point in M;
3. the intersection of the interior of a polygon σ_1, denoted σ_1°, with another primitive in M, $\sigma_1^\circ \cap \sigma_2$, is empty.

Rule #3 implies that every polygon in M is topologically equivalent to a *unit disk*, i.e. it does not contain any holes. Also, observe that rule #2 implies that the intersection of two polygons σ_1 and σ_2, is either empty or a finite set of primitives in M (a point or a line).

Connectedness. Consider two geometric primitives σ_1 and σ_2. We say that σ_1 is *incident* to σ_2 if σ_2 is a primitives forming $\partial\sigma_1$ (the dimensionality of the two primitives is different), and we say that σ_1 and σ_2 are *adjacent* if they share a lower-dimensionality primitives (here σ_1 and σ_2 are of the same dimensionality). If the set of shared primitives contains at least one line, σ_1 and σ_2 are *strongly* connected, and if it contains only points, then they are *weakly* connected. (In Figure 1, polygon A and D are strongly connected, while A and B are weakly connected).

Representing topology. Once a set of spatial objects is topologically consistent, different data structures can be used to represent explicitly and store the topological relationships of the set M of objects. Observe that if M is stored in a computer the names of the geometric primitives change to *node*, *edges* and *faces* (or areas), but these have the same definition. The well-known *node-edge-face* data structure (NEF) can for instance be used. With it, each edge is directed (it has a start and an end node), and the faces left and right of each edge are also stored (most GIS textbook, such as Longley et al. (2001) and Worboys and Duckham (2004), give detailed description of the structure). This permits us to explicitly represent the concept of strong connectedness, but not weak connectedness. To represent the latter, another data structure where the ordering of the edges incident to nodes must be used; the DCEL structure (Muller and Preparata, 1978) or the *half-edge* (Mäntylä, 1988) are two examples. It should however be noticed that, if not explicitly stored, weak connectedness can always be derived.

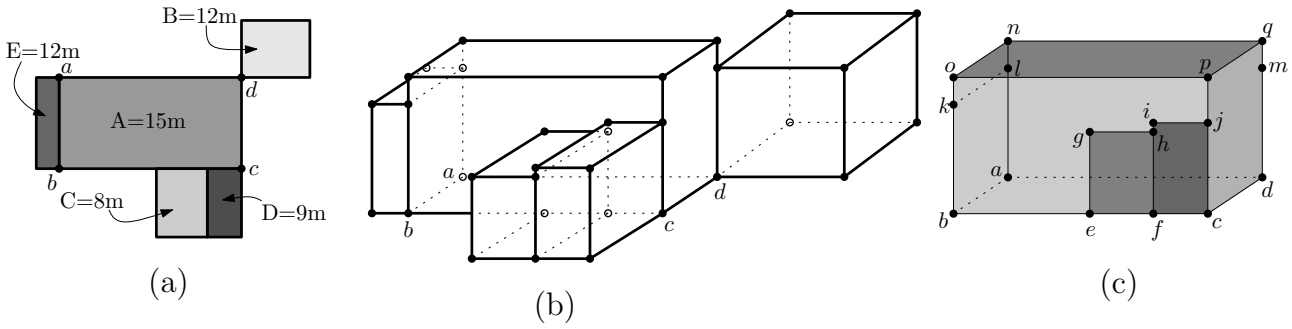

Figure 2: Overview of the extraction of building A. **(a)** The footprint of A is the polygon $abcd$. **(b)** Perspective view of the result (with connected buildings). **(c)** A is formed of several faces.

3.2 3D topological concepts

When moving to $\mathbb{R}^3$, one more geometric primitive has to be introduced: the *polyhedron*. As is the case in 2D, a polyhedron is formed by lower dimensionality primitives. The concept of topological consistency in $\mathbb{R}^3$ is a straightforward generalisation of the three rules previously defined. The only differences are that: (i) all the primitives are embedded in $\mathbb{R}^3$; (ii) another rule has to be added: the intersection of the interior of a polyhedron σ_1 with another primitive in M, $\sigma_1^\circ \cap \sigma_2$, is empty.

3.3 From 2D to 3D with extrusion

In the context of 3D city models, extrusion means that a building (a polyhedron) is constructed by "pushing upwards" its footprint (a polygon). For a footprint with n lines, the resulting building is formed of $n + 2$ surfaces:

- the floor surface corresponds to the footprint;
- the roof surface has the same geometry as the floor but all points are at the extruded height;
- every line becomes a wall (a vertical surface).

If the building footprints are considered independently, or if a footprint does not have any other footprint adjacent, then as discussed previously creating a building is an easy task. However, if there are adjacent footprints (weakly and/or strongly connected), the result will not be topologically consistent. Indeed, consider the case in Figure 2a where five connected footprints are extruded to different heights, yielding the set of objects shown in Figure 2b.

Let us focus on the building A whose footprint is $abcd$. Figure 2c shows the resulting extruded building, with different shades of grey for every extruded surface. First of all, notice that A has 17 points and 9 surfaces; a "simple extrusion" would have created here 8 points and 6 surfaces. Other observations:

- the floor surface is not the polygon $abcd$, but $abefcd$, because C and D are strongly connected to A;
- the roof surface is however formed only of the four points forming the footprint, for the buildings adjacent to it are not as high;
- line ab is extruded to two surfaces $abkl$ and $klno$ since A and E do not have the same height. Observe also that surface $abkl$ will be a surface of both A and E;
- buildings A and B are weakly connected and have different heights, which means that the edges incident to d in 2D (cd and ad) are extruded to surfaces containing a point at the elevation of building B (here surfaces $cdmqpj$ and $dmqnla$). Notice here that point l is in the second surface because E is adjacent to A;
- line bc in the footprint is the most complex one to extrude since three surfaces must be extruded ($beghijpok$, $efhg$ and $fcjih$).

As can be seen from these observations, two factors are to be considered when extruding building footprints: (i) the relative heights of two connected buildings, (ii) their type of connectedness. But it should also be said that we cannot simply consider pair-wise the building footprints as more complex situations arise, for instance

in Figure 2c the point f has 3 incident buildings (A, C and D) that are extruded to different heights. When creating the surfaces containing f, all its adjacent footprints have to be taken into account. It is worth pointing out that many cases are also very simple to handle, for instance extruding a line whose end-points have only one other incident line, as is the case for two lines of building B (the two top-right lines). Here, no special cases arise and the simple extrusion is sufficient.

4 Our approach

This section presents our new algorithm to extrude 2D footprints, called EXTRUDE and detailed in Figure 3. The input of the algorithm is a topologically consistent 2D dataset, and the output is a list of topologically consistent buildings (a building object is a container for a set of 3D surfaces describing the polyhedron). In practice, only a few datasets are topologically consistent. We could go for extruding a non-consistent 2D dataset to a 3D geometric model and try to clean the resulting model by creating 3D topology and correcting errors after the extrusion process. However, since there are currently no tools, to our knowledge, to do this validation in 3D, we chose to use existing tools to create a consistent 2D dataset.

In the previous section we have shown that finding the geometry of the (vertical) wall surfaces is complex due to the different heights and given connectedness of the buildings. Our approach tackles this problem elegantly by introducing the concept of *node columns*. At the location of each 2D node, a node column is erected. Such a column consists of a sorted list of all the different heights of the buildings incident to this node (plus the floor height, which we assume to be zero). In our case, using a NEF data structure, we could find the different heights by navigating from the nodes to the related edges, and then to the adjacent faces (and so obtain the associated heights). Figure 5a illustrates a part of the node columns related to building A (shown in Figure 1a).

Input: a topologically consistent 2D footprint dataset D, with extrusion height for every footprint
Output: a topologically consistent list B of 3D buildings

```
1:  B ← init an empty list
2:  N ← init an empty list
3:  for all faces in D as f do
4:     b ← init a building {with height associated to f}
5:     b ← CreateFloorSurface
6:     b ← CreateRoofSurface
7:     B ← add b
8:  for all nodes in D as n do
9:     nc ← compute a node column
10:    N ← add nc
11: for all edges in D as e do
12:    nc_start, nc_end ← from N {related to e}
13:    if e is adjacent to the universe face then
14:       b ← from B {face not being the universe}
15:       b ← CreateWallSurfaces (e, nc_start, nc_end, b)
16:    else
17:       b_left, b_right ← from B {the two adjacent faces}
18:       b_left, b_right ← CreateWallSurfaces (e, nc_start, nc_end, b_left, b_right)
```

Figure 3: The EXTRUDE algorithm.

The main steps of the algorithm are as follows: First all buildings are initialized. As shown in line 3 of Figure 3, the floor and the roof surfaces are both added instantly after initialising such a container object. Then, after initialising the node columns (line 8), the wall surfaces are pulled up per edge. For this, we visit all edges in the given dataset and apply the CREATEWALLSURFACES algorithm (line 11 and onwards). As we do not have a full area partitioning, edges may be adjacent to empty space. The concept of the universe face models this empty space.

The CREATEWALLSURFACES function, given in Figure 4, outputs the correct number of wall surfaces for each building, with the help of the EXTRUDESEGMENT function, which does the 'heavy lifting' of extrusion. Its input is an edge (for which we can derive the node columns), and the start and end heights of the surface to be created. A ring of coordinates, that describes the geometry of the wall surface, is formed as follows: the

Input: an edge e, node column nc_{start}, node column nc_{end}, one or two buildings (b_{left} and/or b_{right})
Output: building(s) with wall surfaces

$W \leftarrow$ init a list
if one building **then**
  building $\leftarrow$ ExtrudeSegment (e, 0, height of building, nc_{start}, nc_{end})
else {two buildings}
  if height of b_{left} = height of b_{right} **then** {equal height}
    b_{left}, $b_{right} \leftarrow$ ExtrudeSegment (e, 0, height of b_{left}, nc_{start}, nc_{end})
  else
    if height of b_{left} < height of b_{right} **then** {b_{right} is tallest}
      b_{left}, $b_{right} \leftarrow$ ExtrudeSegment (e, 0, height of b_{left}, nc_{start}, nc_{end})
      $b_{right} \leftarrow$ ExtrudeSegment (e, height of b_{left}, height of b_{right}, nc_{start}, nc_{end})
    else {b_{left} is tallest}
      b_{left}, $b_{right} \leftarrow$ ExtrudeSegment (e, 0, height of b_{right}, nc_{start}, nc_{end})
      $b_{left} \leftarrow$ ExtrudeSegment (e, height of b_{right}, height of b_{left}, nc_{start}, nc_{end})

Figure 4: The CreateWallSurfaces algorithm.

given start node column is ascended from the start height, until the end height is reached. Subsequently the end node column is descended from that height, until the start height is encountered. The resulting geometry of the surface is correctly added to the building. Here, correctly means that the normal vector to the surface described points outward of the building to which the surface is added.

Figure 5b shows an illustration of how ExtrudeSegment uses the node columns: the node columns at location e and f are given to the ExtrudeSegment function when edge ef is processed. Edge ef is adjacent to building A and building C, for which building A is the tallest of the two. First, the groundheight (0m) and the height of building C (8m) are given. This way a loop is formed over the two given node columns and results in surface $eghf$. Second, the height of building C (8m) and the height of building A (15m) are given, together with the two node columns and surface $grsih$ is the result. Surface $eghf$ is added to the buildings A and C, as both are adjacent to edge ef. Surface $grsih$ is only added to building A (as this is the tallest of the two buildings). Notice here that since we extrude buildings by processing edges, the resulting surface between building A and buildings C and D is modelled with five surfaces, and not three as in Figure 2c.

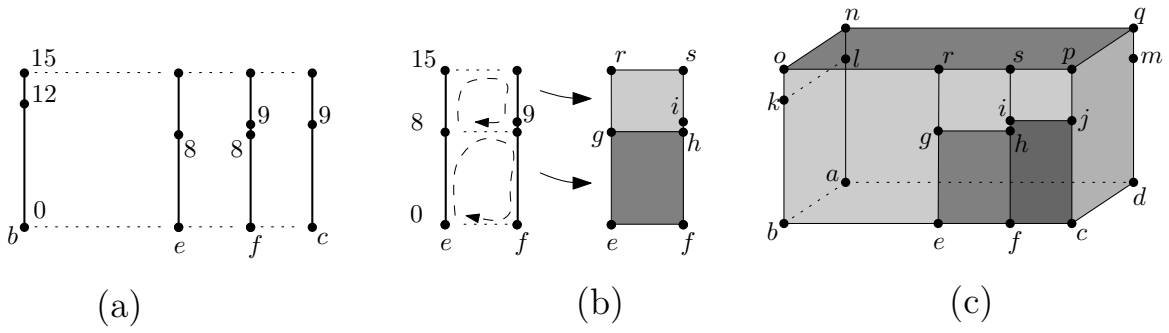

Figure 5: **(a)** Node columns generated for the front side of building A (same configuration as in Figure 2). **(b)** Extrusion of edge ef results in two surfaces **(c)** Extruded result for the footprint of building A.

Observe that if edges were modelled as polylines (compared to straight lines), the ExtrudeSegment function would have to be modified, but could still output correct geometry. With this approach, each edge is entirely processed, but segment per segment. If the start or end vertex of a segment is a node, a given node column (either the start node column or the end column) is used for creating the geometry of the wall surface. If this is not the case—a segment thus starts or ends in a vertex not being a node—a virtual node column is generated, consisting of the start and end height given. This way per segment two node columns can be used and a ring of coordinates can still be formed.

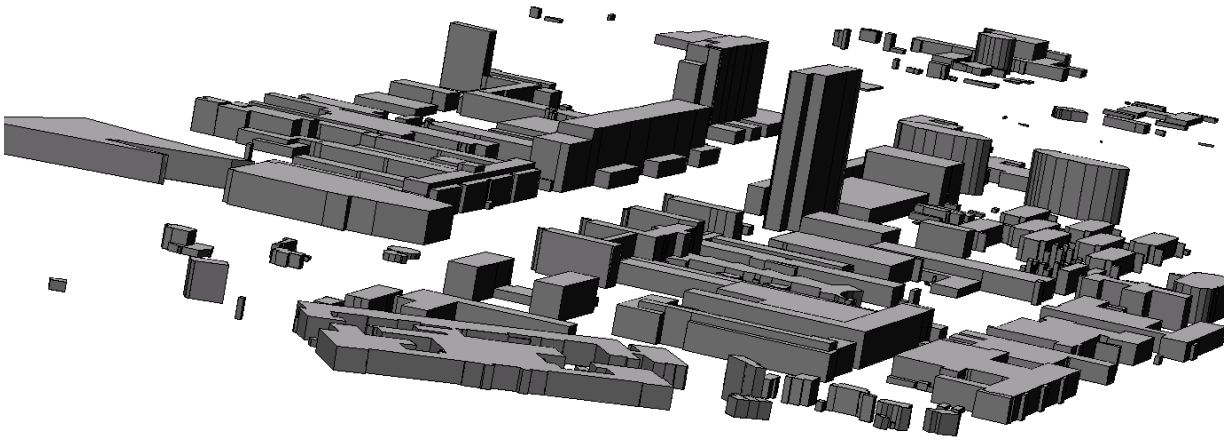

Figure 6: Perspective view of the CityGML file (LOD1) of the TU Delft campus.

5 Experiments and results

To test our approach, we have setup an experiment with real world data of the TU Delft campus. For this area, covering 2.3 km^2 with 470 buildings, we obtained the Large Scale Base map of Delft (Grootschalige Basiskaart in Dutch). This map was a line-oriented dataset, consequently the first step was to form footprint polygons out of the lines.

As the footprints appeared to be too coarse to get an accurate city model with respect to the height of the buildings, it was decided to split all building footprints into multiple, with respect to height, homogeneous parts. This was a manual work and was accomplished by using interpretation of additional aerial photographs of the area. After creating the footprint parts, a height was assigned to each footprint. This value was obtained by using the median z value of all LIDAR points within a given part.

Having a complete footprint dataset, we continued our experiment by creating an explicit topological model of the campus. For this, we used FME's topology builder[3] to convert our geometric building footprints to a node-edge-face data structure. This topological model was loaded into Oracle Topolgy with some handwritten SQL scripts (as the Oracle Topology model uses the more extensive Winged-Edge topology model (Baumgart, 1975)). Oracle Topology provides the ValidateTopoMap function, which can validate a stored topology according to a given tolerance. The original polygon geometry was also loaded into Oracle Spatial and was validated with the supplied ValidateGeometryWithinContext function. Both validation functions highlighted different errors in the source dataset, including overlapping polygons, vertices that were too close to each other and spikes in the dataset. In an iterative way, all errors present in the original dataset were solved by the process of topology creation, validating the topology and geometry and correcting the remaining errors. This way we obtained a topologically consistent 2D dataset that we used as an input for our algorithm. Note that our initial dataset was far from being topologically consistent, while there are also known cases from practice, e.g. the Large Scale Base map of the municipality of Amsterdam, for which a topological structure is used to maintain the dataset. If this had been the case, our algorithm could have run right away.

We implemented the Extrude algorithm in the Python scripting language[4]. Since our program creates topologically consistent datasets in 3D, it is easy to convert to different formats used in practice. We first created a CityGML file of the whole campus, the result can be seen in Figure 6. Each building (LOD1) is stored as a `gml:Solid`, which means that it is a volume with a "watertight" boundary. Notice that CityGML does not offer (yet) mechanisms to represent topological relationships between primitives, which means that nodes and edges shared by higher-dimensionality primitives are repeated. The only mechanism available at the time of writing is the possibility to store only once surfaces shared by two adjacent polyhedra A and B: the surface is represented only once, e.g. A contains that surface, and B has a pointer to the surface (an `xlink` in XML language).

It should be noticed here that CityGML does not permit us to really validate our approach as the XML-

[3] `www.safe.com`
[4] `www.python.org`

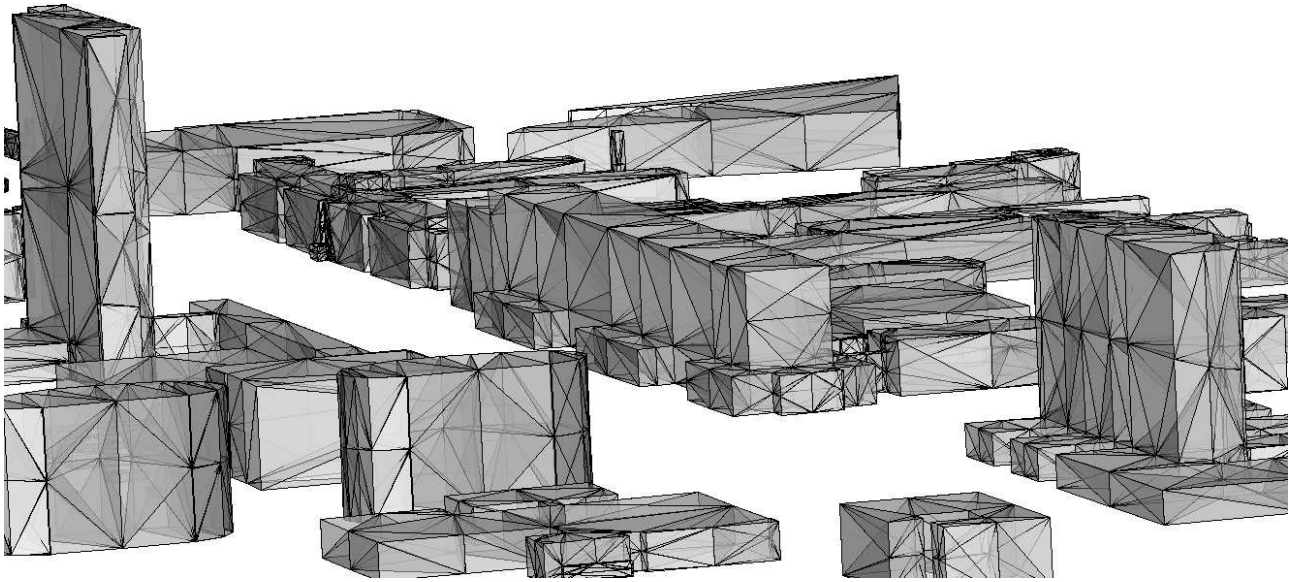

Figure 7: Part of the TU Delft campus tetrahedralized.

based validation (with XML schemas) simply validates the XML tags, and not the geometry or the topological relationships. To validate our approach, we created the constrained Delaunay tetrahedralization (CDT) of our set of buildings. As explained in Shewchuk (2002), CDTs have strong mathematical foundations, and are used in the generation of meshes in engineering. The input of a CDT algorithm is a *piecewise linear complex* (PLC), as first introduced by Miller et al. (1996). A PLC, which is a general boundary description for 3D shapes, contains a set of points, together with a set of straight line segments and polygons. Segments and polygons in a PLC are allowed to intersect at a shared point, and two polygons may intersect only at a finite number of shared points or lines. PLCs are more general than polyhedra, i.e. every polyhedra as defined in Section 3 is a PLC, but not vice versa. If the input of a CDT algorithm is not a valid PLC, then it is impossible to compute the CDT. We therefore converted our set of buildings to a PLC format, and used TetGen (Si, 2004) to perform the tetrahedralization. The PLC format used was the *POLY* format (as explained in Si (2004)), which is a simple text file listing unique points in 3D, and polygons are formed by references to the points. For our set of 470 buildings, the PLC contains 6105 points, and 4331 polygons (which act as constrained faces for the tetrahedralization). The result of this operation is a set of 21700 tetrahedra, part of it shown in Figure 7.

6 Conclusions

We have shown that a problem that appears to be rather simple at first glance—the extrusion of building footprints—turns out to be considerably more complex if we put requirements on the resulting datasets. We have solved the problem by detailing an algorithm that is conceptually simple, and easy to implement. We have introduced the concept of a node column, which permits us to simply construct surfaces and assign them to the correct buildings (modelled as polyhedra), which yields a topologically consistent 3D dataset. Our implementation of the algorithm has been used to create the 3D city model of the TU Delft campus, at a level of detail of LOD1. It should also be said that this work is part of a larger project at TU Delft to build the CityGML model of the whole campus (with the five LODs), store it in a spatial DBMS, and use that for further analysis and for teaching students about 3D GIS. The LOD2 and LOD3 of the campus have already been done (the work was done mostly manually), and we are planning to construct LOD4 by using when possible the digital blue prints of the buildings (as in Brenner et al. (2005)).

As for the future work, we also plan to extend the algorithm to consider holes in the footprints, so that inner courts and/or blocks on roofs can be modelled.

Acknowledgements

We thank our all colleagues for fruitful discussions on the topic, especially Sisi Zlatanova and Friso Penninga for posing the problem, and Theo Tijssen for providing the scripts to validate topology in Oracle.

References

B. G. Baumgart. A polyhedron representation for computer vision. In *National Computer Conference*. AFIPS, 1975.

E. Beinat, A. Godfrind, and R. V. Kothuri. *Pro Oracle Spatial for Oracle Database 11g*. Apress, 2007.

C. Brenner, A. Geiger, and K. Leinemann. Flexible generation of semantic 3D building models. In G. Gröger and T. H. Kolbe, editors, *Proceedings 1st International Workshop on Next Generation 3D City Models*, pages 17–22, Bonn, Germany, 2005.

N. Haala and C. Brenner. Virtual city models from laser altimeter and 2D map data. *Photogrammetric Engineering & Remote Sensing*, 65:787–795, 1999.

N. Haala, S. Becker, and M. Kada. Cell decomposition for building model generation at different scales. In *Proceedings Urban Remote Sensing Joint Event*, pages 1–6, Paris, France, 2007.

T. H. Kolbe. Representing and exchanging 3d city models with CityGmL. In *Proceedings 3rd International Workshop on 3D Geo-Information: Requirements, Acquisition, Modeling, Analysis, Visualization*, Seoul, Korea, 2008. Upcoming, November 2008.

T. H. Kolbe, G. Gröger, and L. Plümer. CityGML—3D city models and their potential for emergency response. In S. Zlatanova and J. Li, editors, *Geospatial Information Technology for Emergency Response*. Taylor & Francis, 2008.

A. Köninger and S. Bartel. 3d-GIS for urban purposes. *Geoinformatica*, 2(1):79–103, 1998. ISSN 1384-6175.

J. Lee and S. Zlatanova. A 3D data model and topological analyses for emergency response in urban areas. In S. Zlatanova and J. Li, editors, *Geospatial Information Technology for Emergency Response*, pages 143–168. Taylor & Francis, 2008.

P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. *Geographic information systems and science*. Wiley, London, 2001.

M. Mäntylä. *An introduction to solid modeling*. Computer Science Press, New York, USA, 1988.

G. L. Miller, D. Talmor, S. hua Teng, N. Walkington, and H. Wang. Control volume meshes using sphere packing: Generation, refinement and coarsening. In *Proceedings 5th International Meshing Roundtable*, pages 47–61, Pittsburgh, USA, 1996.

D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

OGC. City geography markup language (CityGML) encoding standard. Open Geospatial Consortium inc., 2008. Document 08-007r1, version 1.0.0.

A. A. G. Requicha. Representation of rigid solids—theory, methods and systems. *ACM Computing Surveys*, 12 (4):437–464, 1982.

F. Rottensteiner, J. Trinder, and S. Clode. Data acquisition for 3D city models from LIDAR extracting buildings and roads. In *Proceedings IEEE Geoscience and Remote Sensing Symposium (IGARSS '05)*., volume 1, Seoul, Korea, 2005.

C. Schulte and V. Coors. Development of a CityGML ADE for dynamic 3D flood information. In *Proceedings Joint ISCRAM-CHINA and GI4DM Conference on Information Systems for Crisis Management*, Harbin, China, 2008.

J. R. Shewchuk. Constrained Delaunay tetrahedralization and provably good boundary recovery. In *Proceedings 11th International Meshing Roundtable*, pages 193–204, Ithaca, New York, USA, 2002.

H. Si. Tetgen: A quality tetrahedral mesh generator and three-dimensional Delaunay triangulator. User's manual v1.3 9, WIAS, Berlin, Germany, 2004.

R. O. C. Tse, M. Dakowicz, C. M. Gold, and D. Kidner. Building reconstruction using LIDAR data. In *Proceedings 4th ISPRS Workshop on Dynamic and Multi-dimensional GIS*, pages 156–161, Pontypridd, Wales, UK, 2005.

M. F. Worboys and M. Duckham. *GIS: A computing perspective*. CRC Press, second edition edition, 2004.