

GEO-1004 Assignment-1: Simple Polyhedron Processing

Ming-Chieh Hu (6186416)

May 3, 2025

1 Introduction

In this assignment, I implemented a method to simplify a 3D city model by merging neighboring buildings into blocks. The process begins by reading the triangles from an `.obj` file. For each triangle, its expanded bounding box is computed to identify pairs of neighboring triangles. These triangles will then be grouped into blocks. Within each block, all the points from the triangles' vertices are collected to compute the block's convex hull and determine its z-span. Finally, using the convex hull and z-span, triangles representing the block's geometries are generated and wrote into output file. The experiment results (for tile [10-282-562](#)), source code and the compiled binary execution file are in this [GitHub repository](#).

2 Methodology

Algorithm 1 describes the overall structure of this approach. The parts which I implemented in this assignment are mainly:

1. Arguments parsing
2. Simple fan triangulation function
3. Block data fetching and z-span calculation
4. Output files writing function (including the `.mtl` file)

Arguments parsing I exposed its three main parameters (input file, output file and expansion threshold) as part of the command-line API, allowing users to run it without having to recompile the code or manually modify parameters. Users can use the `-h` tag to check the options.

Fan triangulation From the definition of the given code base, the triangulated faces are stored in `output_faces` as triangles. However, during the output file writing phase vertices have to be separated from the triangle objects again, which is more complex since one will have to deal with the vertex duplicates. Instead, I changed the type of `output_faces` to make it only stores indices of each block's convex hull (vector of 3D vertices), as shown in the code listing below.

- `std::vector<Kernel::Triangle_3>> output_faces; // Given definition`
- `std::vector<std::vector<std::array<int, 3>>> output_faces; // My definition`

The fan triangulation function is hence simpler, as shown in Algorithm 2. By making use of the known orientation from `CGAL::convex_hull_2()` function definition and the amount of points, one can already determine the triangle faces without knowing the vertices' actual position. In my implementation, In my implementation, I combined two 2D convex hull rings to construct a 3D convex hull. The data structure is `std::vector<Kernel::Point_3>`, where the first half stores the lower ring and the second half stores the upper ring.

Calculate z-value span To calculate the z-span for each block, the algorithm iterates over all faces in the input mesh. For each face, it computes the minimum and maximum z-values of the face's triangle using the `std::minmax()` function. The block's minimum z is updated if a smaller value is found, and similarly, the maximum z is updated if a larger value is found.

Output files writing As mentioned above, the `.obj` file writing process was simplified by preventing duplicates beforehand. Most of the code in this function is only used to writing strings to the output file stream. Note that this function takes blocks and `output_faces` as input to keep tracking the number of points processed. Besides `.obj` file writing, I made another function to generate random color as material for each block. This function takes the amount of blocks as its input instead of all the block objects.

Algorithm 1: Block Assignment and Mesh Processing

Input: Faces $F = \{f_i\}$ from .obj file
Output: Triangulated mesh `output.obj`
begin
 foreach $f_i \in F$ **do**
 | Compute and expand bounding box \mathcal{B}_i ;
 foreach pair $(\mathcal{B}_i, \mathcal{B}_j)$ **do**
 | **if** $\mathcal{B}_i \cap \mathcal{B}_j \neq \emptyset$ **then**
 | Mark f_i, f_j as neighbors;
 Initialize block counter $c \leftarrow 0$;
 foreach *unassigned* f_i **do**
 | Assign f_i to block c ; queue $Q \leftarrow \{f_i\}$;
 while Q not empty **do**
 | Pop f from Q ; assign unassigned neighbors to c and enqueue;
 | $c \leftarrow c + 1$;
 foreach block B_c **do**
 | Get vertices P_c , compute z_{min}, z_{max} , 2D convex hull H_c ;
 | Lift H_c to z_{min}, z_{max} ; triangulate hull;

Algorithm 2: Fan Triangulation of 3D Convex Hull

Input: 3D convex hull points indices $H = \{h_0, \dots, h_{2m-1}\}$, where the first half of H is the bottom ring (h_0, \dots, h_{m-1}) and the second half is the upper ring (h_m, \dots, h_{2m-1}). Both the rings ordered counterclockwise (CCW) when viewed from above, their face normals point upward.
Output: List of triangular faces F (also represented using indices)
begin
 Set $m \leftarrow |H|/2$;
 /* Bottom face (lower ring) fan triangulation */
 for $i \leftarrow 1$ **to** $m - 2$ **do**
 | Add triangle $(0, i + 1, i)$ to F ;
 /* Bottom face (upper ring) fan triangulation */
 for $i \leftarrow m + 1$ **to** $2m - 2$ **do**
 | Add triangle $(m, i, i + 1)$ to F ;
 /* Wall (side) faces */
 for $i \leftarrow 0$ **to** $m - 2$ **do**
 | Add triangle $(i, i + 1, m + i)$ to F ;
 | Add triangle $(i + 1, m + i + 1, m + i)$ to F ;
 Add triangle $(m - 1, 0, 2m - 1)$ to F ;
 Add triangle $(0, m, 2m - 1)$ to F ;

3 Results

This section presents the results of various experiments conducted on tile 10-282-562 to evaluate the performance of the proposed method. It was tested with different expansion thresholds and Levels of Detail (LoDs). Issues and key observations are identified in following paragraphs through visualization and statistical analysis. Figure 3 showed the test result on other two tiles 9-304-520 and tile 9-308-520 with LoD 2.2 input and different expansion threshold values.

Iso-oriented bounding box Figure 2a showed that, using iso-oriented bounding boxes may make the behavior of algorithm unpredictable, as they sometimes make the bounding boxes bigger than the Faces actually are. Depending on the orientation of the input buildings, the program may produce very different results for the same set of buildings.

Effects of different LoDs It is shown in Table 1 that, given a reasonable threshold value, the file with higher LoD almost guaranteed to separate more blocks from same tile. This is expected, as higher LoD models typically use smaller, more detailed faces. As a result, the bounding boxes of these individual faces are generally smaller than those of larger, aggregated faces, as illustrated in Figure 2b.

Common challenges of threshold-based methods This method relies entirely on the expansion threshold, much like density-based clustering methods or region-growing algorithms. And fundamentally, these methods almost always involve a trade-off: too small a threshold can fragment the results, while too large a threshold can over-merge or blur meaningful boundaries. It’s generally impossible to find a single “perfect” threshold value that works universally, since the ideal setting depends on the specific characteristics of the input data and the desired output. Comparison between outcomes with different threshold values is displayed in Table 1.

Hidden and touching blocks In Table 1, one can also see that based on the buildings’ arrangement, some blocks may be touching each other (yellow) or fully hidden inside another (red). This is mainly due to the use of the convex hull to represent blocks, which also reveals a trade-off between computational complexity and output quality. The effect is especially noticeable when buildings have C-shaped contours, as the convex hull can significantly misrepresent the shape (and hence volume) of the block.

Unoccupied block volume As shown in Figure 1, some of the merged blocks cannot represent the buildings inside very well, leaving plenty of unoccupied block volume. This is especially obvious in the output from LoD 1.3 and LoD 2.2 models. LoD 1.2 models did not have this problem since the towers are harmonized.

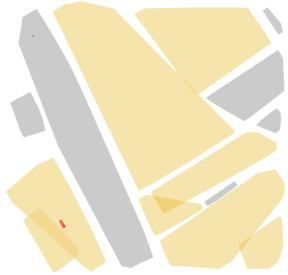
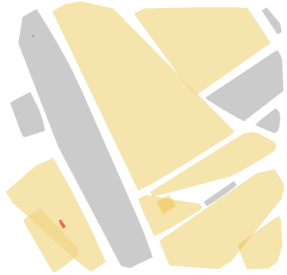

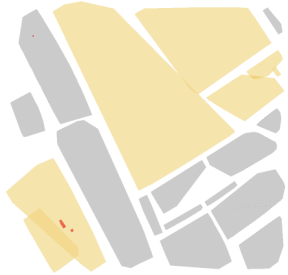
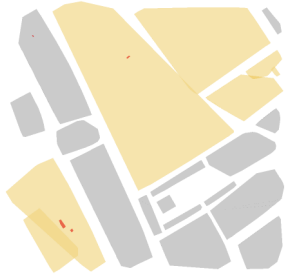

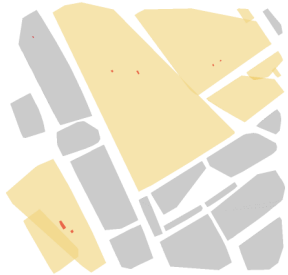


	LoD 1.2	LoD 1.3	LoD 2.2
Threshold=1.5			
	Total: 16, Hidden: 2	Total: 17, Hidden: 2	Total: 20, Hidden: 1
Threshold=1.0			
	Total: 23, Hidden: 3	Total: 25, Hidden: 4	Total: 32, Hidden: 4
Threshold=0.5			
	Total: 30, Hidden: 7	Total: 33, Hidden: 8	Total: 41, Hidden: 9

Table 1: Summary of thresholds, LoDs, and block statistics. Red blocks are hidden; yellow blocks are touching.

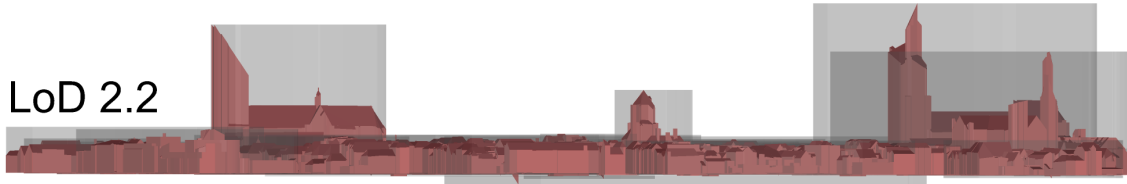
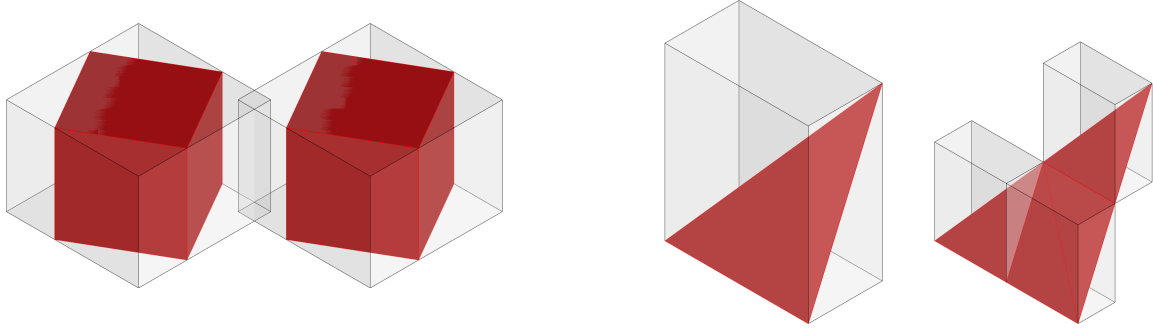


Figure 1: The simple block model does not reflect the true height of buildings inside.



(a) An intersection for iso-oriented bounding boxes.

(b) Bounding box volume for different LoD faces.

Figure 2: Detailed scenarios for iso-oriented bounding boxes.

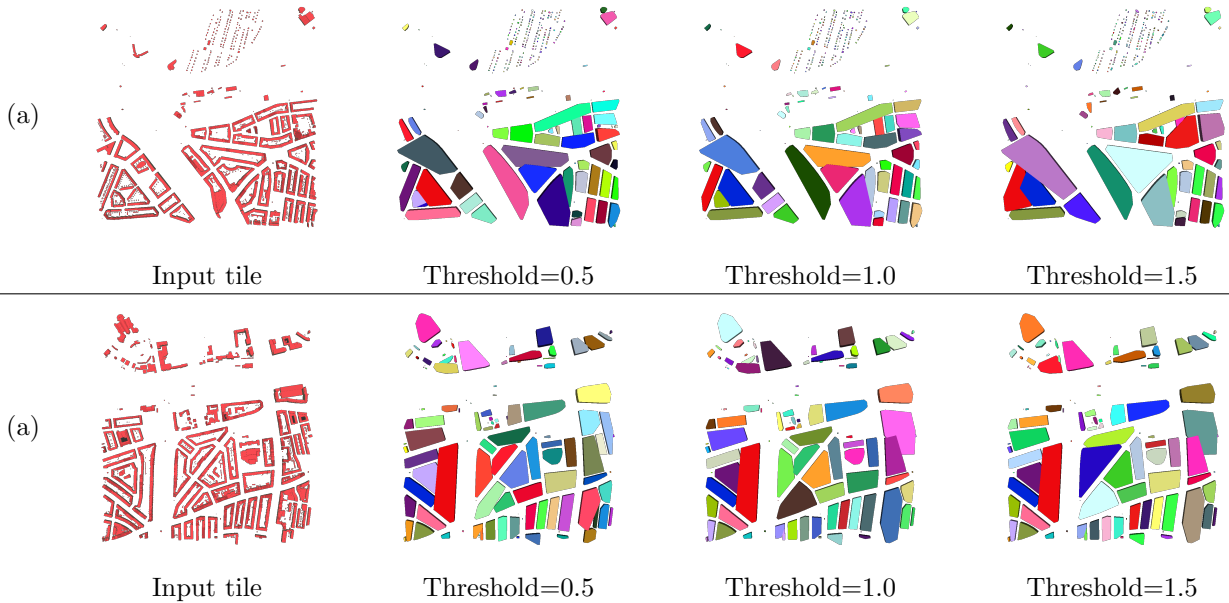


Figure 3: Results using LoD 2.2 and: (a) tile 9-304-520; (b) tile 9-308-520 results.

4 Use of AI

All the methods mentioned above are my own ideas (except for the provided parts in the sample code). Since the operation here is relatively simple, I didn't explicitly use complex functions. Most of the methods I used could be easily found on either cplusplus.com or [CGAL documentation](http://CGAL.org). For those C++ or CGAL methods that were difficult to find or hard to understand, I use ChatGPT to generate explanations and examples. And besides coding, I also use ChatGPT to format some of the tables and figures in this report.