A1 Simple Polyhedron Processing

3D MODELLING OF THE BUILT ENVIRONMENT

GEO1004

Authors:

Hongyu Ye (6286240)
Viktor Bozev ()

May 12, 2025

1 Methodology

1.1 .OBJ File Reading

Open the file with `std::ifstream`, read it line by line, and inspect the first word (*line_type*): If it is `v`, then it is a vertex line; if it is `f`, then it is a face line.

Our added code performs the following action—if exactly three vertices were read, we finally create the triangle.

1.2 Generate Expanded BBoxes

First, call `Triangle_3::bbox()` to obtain a three-dimensional bounding box$(x_{\min},\, x_{\max},\, y_{\min},\, y_{\max},\, z_{\min},\, z_{\max})$.

Next, project the box onto the xy-plane by taking $(x_{\min}, x_{\max})$ and $(y_{\min}, y_{\max})$ and *inflate* the resulting rectangle in each of the four directions by adding (or subtracting) the prescribed *threshold*.

Finally, insert the expanded bounds together with *an iterator to the current face* into the `boxes` container:

```
boxes.emplace_back(lo, hi, current_face);
```

Storing the iterator enables CGAL to identify exactly which two faces overlap during later intersection tests.

1.3 Neighbor Finding

Run CGAL's `box_self_intersection_d()` algorithm over the entire `boxes` array to detect every pair of overlapping boxes. Then in the added code, each *box overlap* is interpreted as the fact that *face A and face B are neighbors*.

For every such pair we record the relationship by inserting, into each face's `std::set` $\langle$`std::size_t`$\rangle$ `neighbours`, the index of the other face obtained through its iterator in `input_faces`. This operation incrementally builds an *adjacency table* for the mesh.

Then the given code groups all mutually adjacent faces into connected blocks by performing a BFS over the *adjacency table*.

1.4 Convex Hull & Triangles Constructing

Firstly, collecting vertices and computing heights. We begin by resizing the container so that `blocks.resize(current_block)` matches the number of connected components. Scanning every triangular face, we extract its three 3-D vertices but retain only their (x, y) coordinates. For each component we keep track of the maximum and minimum z–values, denoted `blk.max` and `blk.min`, which later fix the heights of the *roof* and the *floor* respectively.

Then, generating triangles for *roof*, *floor*, and *façade*.Once the two-dimensional convex hull of the retained vertices has been computed by the given code, the mesh is produced in three parts:

1. *Roof* and *floor* use *fan triangulation*. Taking the first hull vertex as the fan origin `roof_origin`, we create triangles $(0, i, i+1)$ in sequence. The roof is oriented CCW and the floor CW so that their normal vectors point outward and inward, respectively.
2. For the *façade*, process every hull edge $(i \to j)$: let (a, b) be the roof-level edge, $z_{\text{top}} = \max(c, d)$ the corresponding upper z, and $z_{\text{bottom}} = \min(c, d)$ the lower z. These four vertices form a rectangle which is split into the two triangles (a, b, c) and (a, c, d).

1.5 .OBJ File Writing

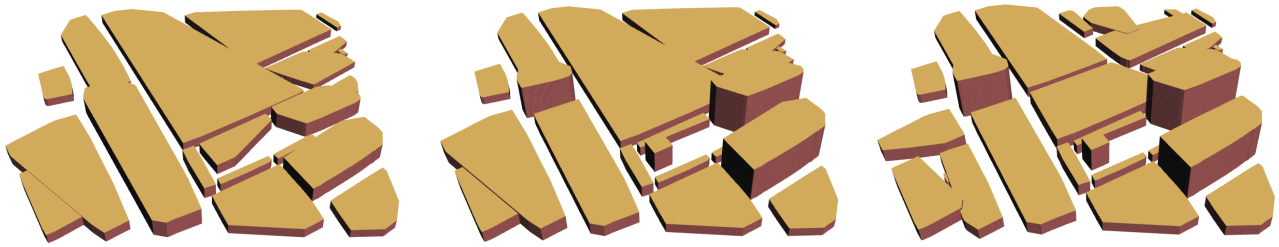
First, write every vertex sequentially in triangle order as `v x y z`, so that the vertex indices automatically coincide with the order of appearance; then, after every three vertices, output `f i j k` (with i, j, k equal to the current vertex numbers plus 1), allowing each triangular face to reference them correctly.

2 Testing, Discussion & Instruction

2.1 Testing

Nine results were generated across different LoDs and thresholds, as summarized in Table Table 1. When the threshold is 1.0, the block count at LoD 1.2 is 23, increasing to 26 and 32 at finer LoDs of 1.3 and 2.2, respectively. For a fixed LoD, the number of blocks decreases as the threshold increases. Moreover, the number of triangles is disproportionately large compared to the number of blocks, contrary to our expectations.

Figure Figure 1 shows the outputs at threshold 1.0 for various LoDs. As the LoD increases, block heights become more varied and shape simplification decreases. Compared to the original data, originally independent buildings or floors merge; the convex hull computation and subsequent extrusion produce large solid entities with incorrect geometric properties, resulting in a loss of the original semantic information.



(a) LoD 1.2 (b) LoD 1.3 (c) LoD 2.2

Figure 1: Output using different LoD data materials

Table 1: Intersections, blocks, and triangles at different thresholds and LoD

	LoD-22			LoD-13			LoD-12		
Threshold	1	1.5	2	1	1.5	2	1	1.5	2
Intersections	5 391 142	7 313 475	9 484 480	2 352 694	2 988 249	3 704 754	1 311 665	1 609 411	1 944 601
Blocks	32	20	14	26	17	13	23	16	12
Triangles	2 104	1 460	1 056	1 656	1 180	916	1 484	1 100	836

2.2 Conclusion & Discussion

Overall, the results perform as expected—at LoD 1.2, the number, shapes, and positions of blocks closely match those shown in the reference images, demonstrating that our algorithm *works well*. Of course, after further reflection on the code and both quantitative and qualitative analysis of the outputs, we identified three potential issues:

Problem 1: Simplistic parsing of face (`f`) lines. The code reads each vertex index as a plain integer in sequence, which neither supports the common "vertex/texture/normal" slash format in `.OBJ` files nor prevents pushing an undefined value into the container on the final failed read of a `while(!eof())` loop. As a result, indices become misaligned or `Face.triangle` remains uninitialized, and subsequent calls to `bbox()` may access out-of-bounds memory, causing crashes or corrupted geometry.

Problem 2: Neighbor face detection relies solely on 2D expanded bounding boxes. The algorithm projects 3D triangles onto the xy-plane, enlarges each rectangle by a fixed threshold, and tests for box intersection, completely ignoring positional differences along the z-axis. In scenarios such as overpasses or eaves, components that should remain separate in upper and lower layers are mistakenly merged into the same block, and subsequent convex-hull extrusion "fuses" multiple independent buildings into one solid.

Problem 3: No deduplication of vertices during output. When writing the `.OBJ`, each encountered triangle causes its three vertices to be appended directly to the list and referenced in order, without reusing existing vertex indices. For large models, this generates massive duplicate vertex entries, bloating the file size, degrading loading and rendering performance, and wasting considerable memory.

2.3 Instructions to Run the Code

Ensure CGAL is installed, then compile the source with:

```
g++ -std=c++17 -O3 main.cpp -lCGAL -lboost_system -o lod_extruder
```

Edit the two constants `input_file` and `output_file` near the top of `main.cpp` to point to your own `.OBJ` input and desired output path.
From the terminal, run:

```
./lod_extruder
```

The program will read the input OBJ, generate simplified block geometry, and write the new OBJ to the specified location.

Work Allocation

GitHub Page: https://github.com/Jackson513ye/GEO1004A1

- Hongyu Ye: developed the implementation logic, completed the code implementation, analyzed the output results and potential issues in the code, and wrote the report.
- Viktor Bozev:

Uses of AI

In this assignment, AI was used to debug portions of the implemented code—saving time in locating errors; translate the author's non-English content into English for inclusion in the report; and convert text-formatted material into LaTeX format and typeset it. The author has proofread all AI-processed content to ensure its accuracy.