Geo1004-assignment3
BIM to Geo conversion using voxels

Xiaoluo Gong, 5923476
Haohua Gan, 6007503
Zhuoyue Wang 6093590

April 9, 2024

1 Main Steps in Conversion Methodology

1.1 From IFC to OBJ

We tried to convert IFC to OBJ directly with the IFConvert, but we encountered with some problems when we tried to delete the furniture and parking lots. So we used the BlenderBIM add-on for Blender to transfer it and filtered out all the unnecessary items by hand.

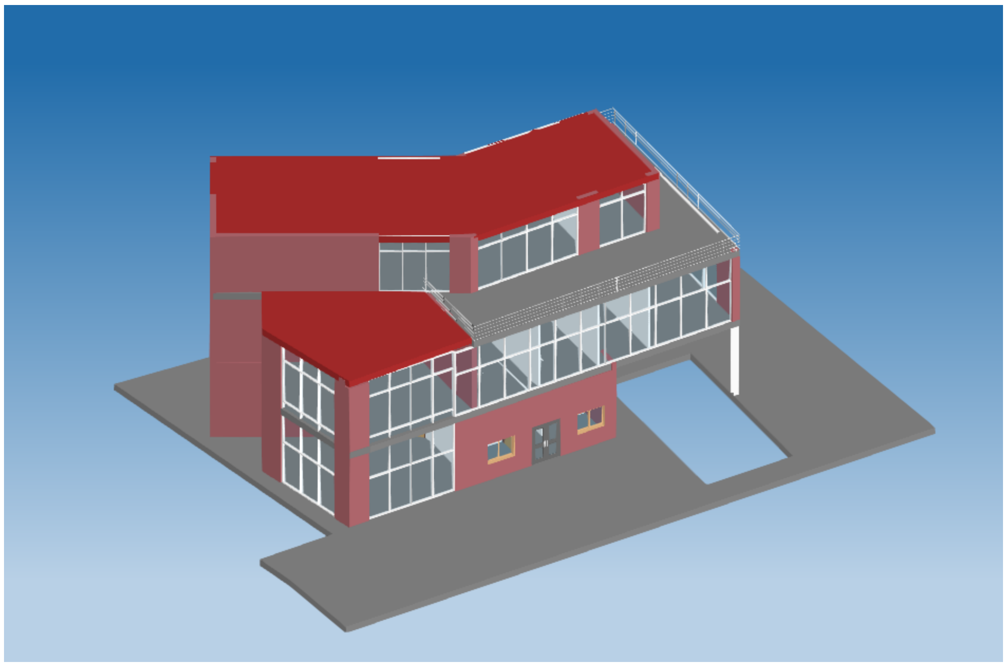


Figure 1: IFC file of wellness center after removing all the furniture and most of the street-level elements (e.g., roads)

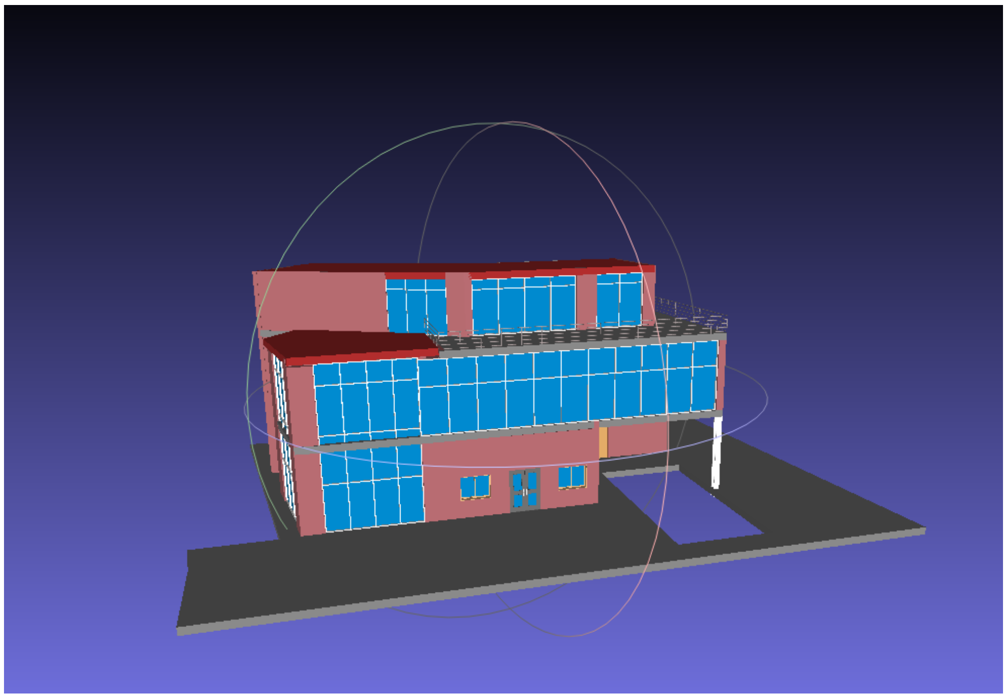


Figure 2: OBJ file transformed by the IFC file in Figure 1.

1.2 From OBJ to Memory

The OBJ files contained a .obj and .mtl which recorded the coordinates of the building points and materials respectively. We loaded all the information into C++ memory by designing the structures including **Vertex, VertexNormal, Face, Group, Materials**. We also made a struct called **ObjInfo** to contain all the information of the input data.

1.3 The voxel grid

We followed the VoxelGrid structure in the assignment instructions and made some adaptions. We create the struct **Voxel** to store the coordinates and row number of each voxel. Also, we added the fields of **value** and **group** to label each voxel in the following steps.

We created the overall voxel grid which covered the whole building with the extended bounding box of the building. The voxel size can be adjusted according to the users' need, and we do the voxel size of 0.1, 0.2, and 0.3 to see if the resolution matters and check the robustness of our code. We extended the 10-voxel size for each dimension in x, y and z to make sure all shells of the building have exteriors to help extract surfaces with the right orientations (see 1.6 for details).

1.4 Voxelisation of triangles

We did the voxelisation of triangles with the function **triangle_voxel_intersection** under the **Voxel** struct. At the beginning, the voxels in the whole empty voxel grid have the default label **value** of 0. We looped through all the triangles in the OBJ file and the voxels inside the boundary box of the triangle to check if they intersected. We marked all the intersected voxels with **value** of

1. For the boundary box, we also extended it for one voxel size for each dimension to avoid a flat boundary box. We made every voxel in the boundary areas as a cube in CGAL and applied the **CGAL::do_intersect** function. According to the documentation of **CGAL::do_intersect**[2], the intersections return type can be point, segment, and triangle. Thus, the connectivity should be 6-connectivity for the intersection targets.

1.5 Marking exterior and room voxels

After the voxelisation of triangles, we now have a voxel grid with two kinds of voxels:

- **voxels with value 1**: indicating that these are the voxels intersected with triangles (building part voxels).
- **voxels with value 0**: these voxels are either exterior voxels or interior voxels (room voxels).

We first start by marking the exterior voxels. Since we've extended 10 voxels along the x, y and z dimensions, we can ensure that the voxels which are located at the boundaries of the voxel grid are exterior voxels. Thus, the exterior voxel marking starts by detecting the boundary voxels, the whole process is:

1. Search the whole voxel grid and find a boundary voxel as the initial start voxel.
2. Change the start voxel's value to 2, representing an exterior voxel.
3. Using 6-connectivity, we validate the 6 neighbour voxels of the start voxel and process them based on the validation result:
   (a) Whether the neighbour voxel is valid (i.e., ensure it is within the voxel grid's boundary), if it is not valid, skip it.
   (b) Whether the neighbour voxel's value is 0 (i.e., it is not a building part voxel), if the value is 0, make this voxel a candidate (i.e., a new start voxel) for future iterations.
4. Repeat steps 2 and 3 until there is no start voxel.

After marking the building part and exterior voxels, all the leftover voxels are room voxels, and the process of labelling room voxels is pretty much the same as labelling exterior voxels. The difference is that the room label starts at 3 and each time the iteration stops (i.e., finish marking a room), we add the room label by 1. So if we have 3 rooms, the voxel value of each room is 3, 4 and 5.

However, not all rooms are real rooms. The geometries of an OBJ file are constructed by surfaces, which are not solid. For example, a wall is actually hollow, and if we set a too-fine voxel size (e.g., a voxel size that is much thinner than the width of a wall), the space inside the wall can be marked as a room (figure 3).

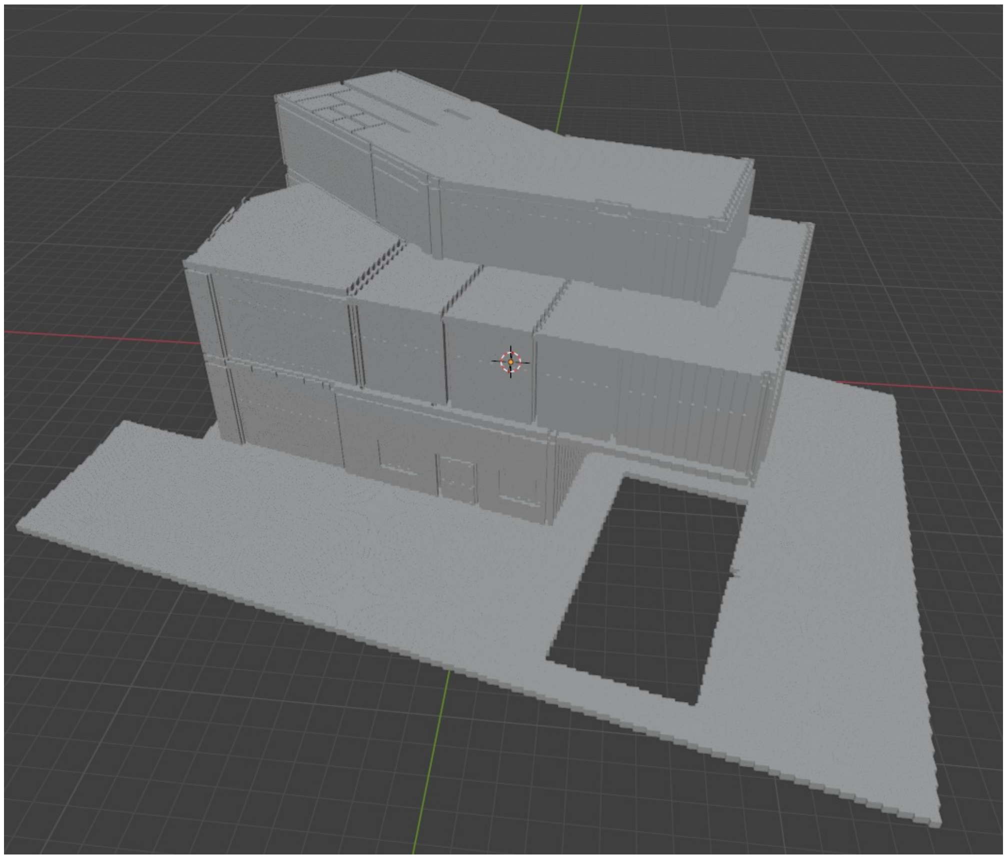


Figure 3: Invalid room voxels. Notice that there are room voxels on the ground, this is because the space between the ground surfaces is being marked as a room.

Therefore, we then define that a room should be greater than 1 meter in **height**, **length** and **width**. Then, after marking a room, we get the vertices of this room and generate the oriented bounding box (OBB) using the **CGAL::oriented_bounding_box**[1] function and we compute the **height**, **length** and **width** of the OBB. If any of these three numbers is lower than 1, we consider this room invalid (i.e., it is the space inside a building part such as a wall) and change the value of this room's voxels to 1, marking them as building part voxels.

1.6 Extracting the surfaces

After marking the exterior and room voxels with different values, we want to extract the faces between the different value's voxels, which represent the boundary of the outer envelope and the rooms. The first step is to identify the faces between the voxels with different IDs. Since the voxels for building parts separate the exterior and the different rooms, it makes sense to traverse each voxel from the building parts to check whether it has the same value as its immediate neighbors across six orthogonal directions (up, down, left, right, front, back). If the value of the building is different from its neighbors', it indicates a transition between two distinct spatial regions, which means that this face should be extracted.

Then, the identified faces are added to the structure that stores the vertices and the indices of the vertex. For each face, it is extracted based on the voxel's position (k, j, i) and its direction (dx,

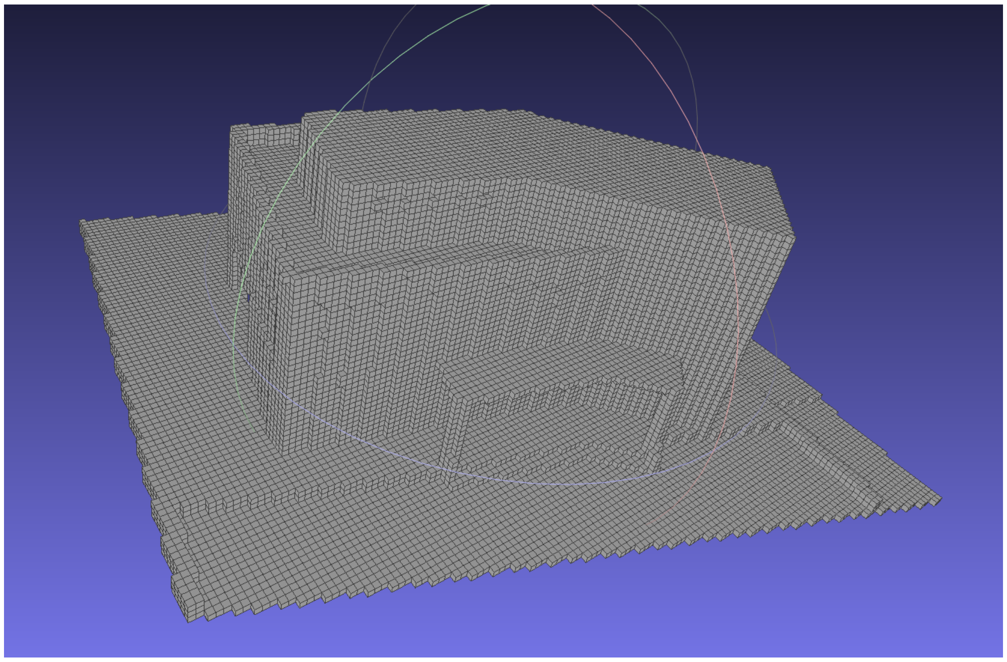


Figure 4: Visualization of the extracted surfaces of the outer envelope for the Wellness Center Sama, viewed in meshlab.

dy, dz) relative to a neighbor with a differing value. Given the positive or negative values of dx, dy, and dz, we can know whether this extracted face is on the left/right, top/bottom, front/back of the original building part's voxel. This way, the orientation of the faces can also be determined, that is for example, if the faces are on the top side of the building part's voxel, the index of the vertices is written in the counterclockwise order (viewing from the top), while it is reversed for the bottom side of the voxel (also viewing from the top). In this way, we can ensure that all the normal vectors are facing out from the building part's voxel, which will give us the faces facing out for the outer envelope, and facing in for all the rooms. However, since we do not know exactly the positive or negative side of the voxel grid, the results are obtained after tests (checking the orientation of faces on the exported obj file). The visualization of the exported obj file of the outer envelope of the wellness center Sama is shown in 4, we can see that the orientation of the surfaces is facing out. The orientation of the rooms are all facing in for all the rooms extracted as seen in 5.

Given the order of the vertex indices, the coordinates are extracted from the voxel grid, and written to the structure. To ensure we only have unique vertices, the vertex is only added if it does not already exist. This step is critical for avoiding redundancy and ensuring efficient data representation.

Another structure **Room** is designed to store all the separate faces in groups. The groups include the outer surface and the different rooms, which were marked with different IDs from the last step. This way, all the faces from the voxels' side are connected to their room number.

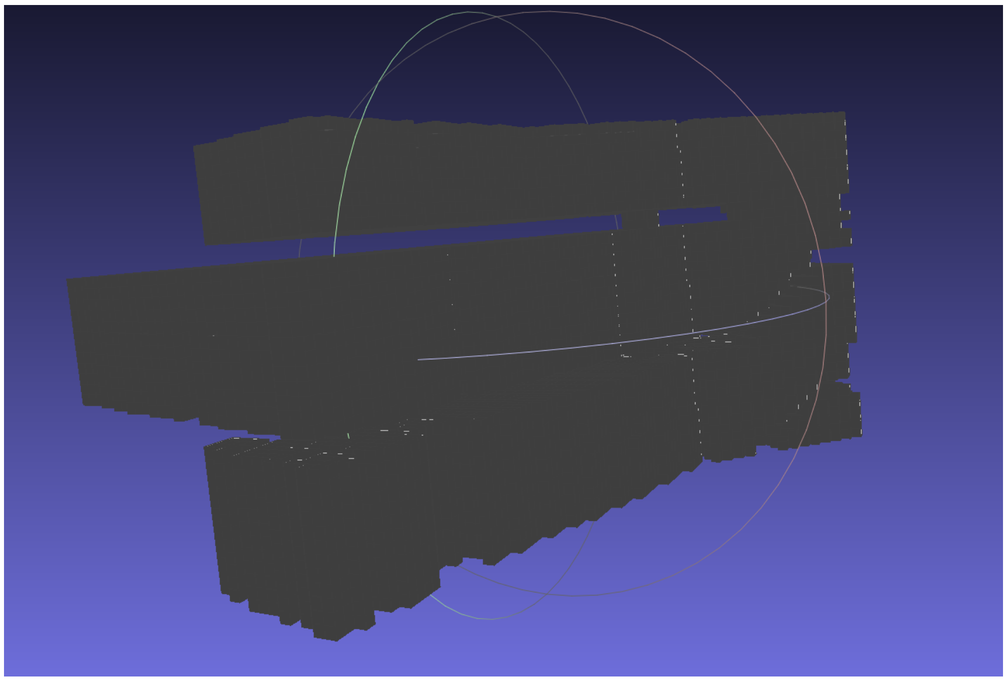


Figure 5: Visualization of the extracted surfaces of the rooms for the Wellness Center Sama, viewed in meshlab

1.7 Writing CityJSON

We wrote the CityJSON referring to the schemas of CityJSON 2.0 and the example code given on the introduction page of the assignment. In our output CityJSON, the overall information is stored in the **BuildingParent** and its children including the **OuterEnvelope** for the building part and the **RoomX** for each room inside the building.

For the children objects, we wrote all the geometry information under the **geometry** and set the type to **MultiSurface**. Because we want to store the outer envelope and each room separately as a child of the BuildingParent, the output of each child doesn't have any interior shell. It is an easy and simple way to store them in the type of **MultiSurface**.

We also specify the scale and translate parameters. The scale parameter is the voxel size, and the translate parameters are the minimum x, y and z coordinates of the voxel grid. We did this to reduce the size of the output CityJSON file.

2 Strength and Weakness Analysis

2.1 Strength

- **Robustness**: Our code can handle the complex buildings like the wellness center. For example, In the room validation process, we first used the room's **height** and **volume** to validate (e.g., $height > 1m \ and \ volume > 2m^3$), but later we found out that this method could not filter out those tall and thin "rooms" (e.g., space inside a wall be marked as a room), thus we decided to compute the oriented bounding box (OBB) of a room and use its **height**, **length** and **width** for the validation. The OBB method can consider the shape of a room without being affected by the room's orientation. In doing so, we successfully filtered out all the invalid rooms.

- **Optimization of loops**: We tried to loop less to improve our code efficiency. For example, when calculating the intersections, we tried to loop over as less voxels as possible so that we only loop through the voxels inside the boundary box of each triangle.
- **Modular Structure**: The code is organized into functions and structures, each with a clear purpose, such as reading OBJ files, creating voxel grids, and extracting surfaces. This modularity aids in readability and maintenance.
- **Efficient Data Structures**: The use of vectors, maps, and custom structs for storing vertices, normals, faces, and materials indicates efficient data management for 3D geometry processing. We load all the information from OBJ file to memory. Although we didn't use all of them for this assignment, we can use them easily if we want to make some adjustments for the future.
- **Error Handling**: There's an attempt to handle errors, such as checking if files can be opened, which makes the code more user-friendly.

2.2 Weakness

- **Hardcoded Parameters**: We didn't design an interface for users to input their own definitions or validation rules for rooms. Because of the OBB method we applied, we have to define a valid room's **height**, **length** and **width**, and we set the threshold to 1 meter. This definition is not robust to all kinds of rooms such as a toilet where the length or width can be lower than 1 meter. Therefore, to avoid wrong validation, the threshold needs to be tuned manually which requires prior knowledge of the data, making this validation not generalized enough.
- **Duplicated Warning**: Although we made our output CityJSON file schema validated, we still get warnings of duplicated vertices. We tried to figure it out by adjusting the scale and transform. Also, we only add a vertex to Room if it doesn't already exist and return its index in lines 580 to 588 to avoid duplicates. However, after all the attempts, we still couldn't completely solve the warning problem.
- **Output Expansion**: Since we didn't merge the grid faces inside a single wall or roof surface, the number of output faces will exponentially grow when the voxel size gets smaller. Thus, the size of output files will grow exponentially. For example, the output for voxel size of 0.2 is only 35Mb, while the output of 0.1 is 192Mb. If we do the merge or simplification of the surface, the output size wouldn't expand so much.

3 Quality Assessment

3.1 Code Performance

We tested out code by different inputs including the IfcOpenHouse and wellness center and all of them works well.For the wellness center, we generate the 0.3 voxel size CityJSON with a Mac laptop in about 10 seconds, while the 0.1 voxel size CityJSON in 8 minutes.

The time complexity of our code depends on the main parts of the code, particularly those that involve iterating through data structures, performing geometric computations, and manipulating collections. Key components include:

- **Triangle-Voxel Intersection**:For each triangle in the OBJ file, the code checks intersection with the voxel grid. The worst-case time complexity is O(T * V), where T is the number of triangles and V is the number of voxels checked for intersection with each triangle. The actual number of voxels checked depends on the size of the triangle and the resolution of the grid, which can vary widely.

- **Marking Exterior and Interior Voxels**: The worst-case time complexity for marking exterior voxels is O(V), where V is the total number of voxels. Identifying interior voxels and rooms involves additional fill operations, potentially visiting each voxel multiple times. The worst-case complexity remains O(V) due to the iterative approach, but with a higher constant factor due to multiple passes.

- **Surface Extraction**: Identifying the surfaces involves iterating over the voxel grid and checking the neighboring voxels for each voxel belonging to a building part. The time complexity is O(V * 6), considering each voxel checks its 6 neighbors, simplifying to O(V).

- **Exporting Data**: Writing voxel and geometry data to OBJ files or a CityJSON format involves iterating over the voxels and surfaces extracted. The time complexity is O(V + S), where S is the number of surfaces identified.

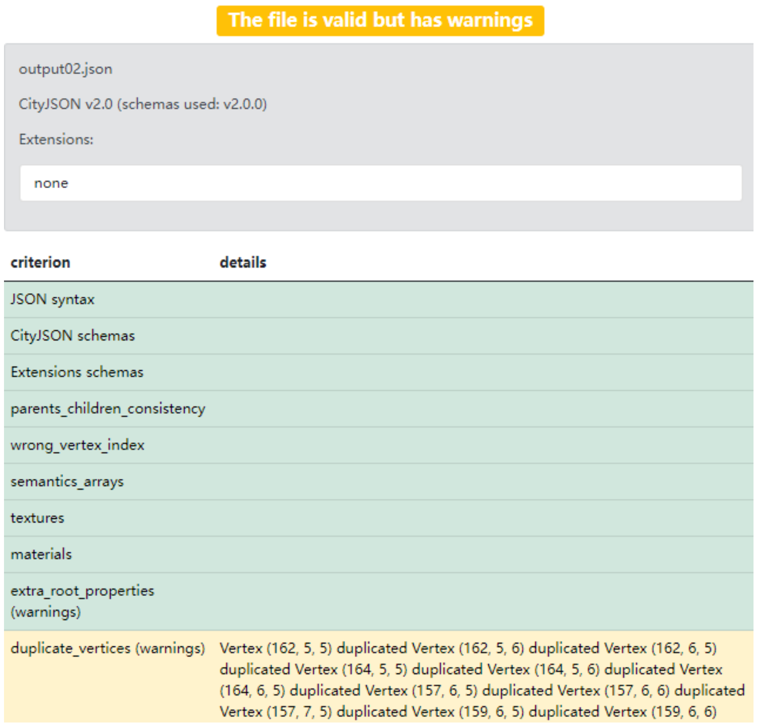
Overall Time Complexity: The overall time complexity of the program is determined by the most expensive operations, which are related to the voxel grid manipulation and triangle-voxel intersection checks. Therefore, the dominating time complexity can be approximated as O(T * V) for the intersection checks, given this could potentially involve checking each voxel against each triangle. In practice, the performance will also depend on factors like the implementation of geometric libraries (e.g., CGAL), the efficiency of data structures used (e.g., for representing the voxel grid and geometry), and hardware capabilities.

3.2 Quality of Output

- **Schema Validation**: As mentioned in the weakness part (see 2.2), our output CityJSON is valid but has warnings of duplicated vertices (see figure6).

- **Geometry Validation**: Our output is 100 $\%$ valid with the val3dity (see figure7).

- **Visualization**: We generated CityJSON with voxel size of 0.1, 0.2, and 0.3 and visualized them in Ninja (see figure 8). With the smaller voxel size, the edge



The file is valid but has warnings

output02.json

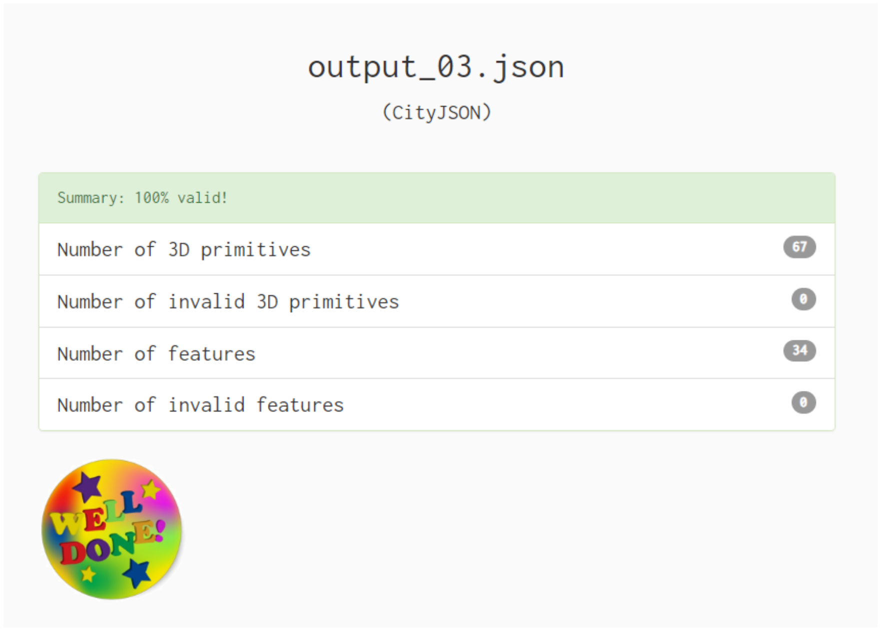
CityJSON v2.0 (schemas used: v2.0.0)

Extensions:

none

criterion	details
JSON syntax	
CityJSON schemas	
Extensions schemas	
parents_children_consistency	
wrong_vertex_index	
semantics_arrays	
textures	
materials	
extra_root_properties (warnings)	
duplicate_vertices (warnings)	Vertex (162, 5, 5) duplicated Vertex (162, 5, 6) duplicated Vertex (162, 6, 5) duplicated Vertex (164, 5, 5) duplicated Vertex (164, 5, 6) duplicated Vertex (164, 6, 5) duplicated Vertex (157, 6, 5) duplicated Vertex (157, 6, 6) duplicated Vertex (157, 7, 5) duplicated Vertex (159, 6, 5) duplicated Vertex (159, 6, 6)

Figure 6: Schema validation result



output_03.json

(CityJSON)

Summary: 100% valid!

Number of 3D primitives	67
Number of invalid 3D primitives	0
Number of features	34
Number of invalid features	0

Figure 7: Geometry validation result

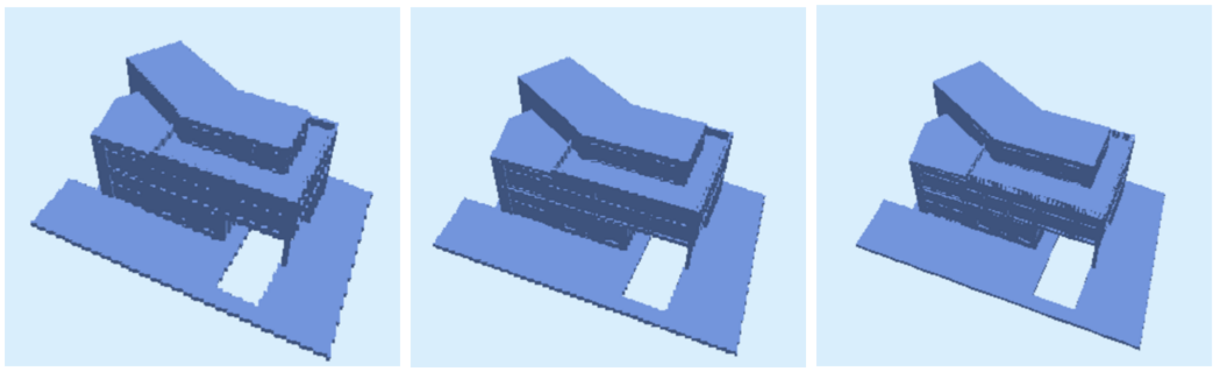


Figure 8: Visualization result

4 Workload Allocation

Haohua Gan: Focus on steps 5 and 7, and help to fix errors in steps 1 to 4.
Xiaoluo Gong: Mainly focus on step 1 to step 4, discussion in step 6 and 7 and write the report. Because of the failure in using GMP library even under the help of Ken and Dimitris, Xiaouo Gong can do little work after step 4.
Zhuoyue Wang: Mainly focus on step 6 to extract the surface and help with Xiaoluo in step 4.

References

[1] CGAL Optimal Bounding Box. Cgal 5.6.1 - optimal bounding box. https://doc.cgal.org/latest/Optimal_bounding_box/group__PkgOptimalBoundingBox__Oriented__bounding__box.html#gac1917f59df722d338d44a63d3a8aa14a, 2024. Accessed: 2024-04-08.

[2] The CGAL Project. Cgal 5.4 - 2d and 3d linear geometry kernel: Intersection functions. https://doc.cgal.org/latest/Kernel_23/group__intersection__linear__grp.html, 2023. Accessed: 2024-04-08.