Delft University of Technology

GEO 1004 3D modelling for the built environment(2021/22 Q3)

Assignment 03

Assignment group members:

SURNAME: Xia
NAME: Yitong
STUDENT ID: 5445825
STUDY PROGRAMME: Geomatics for the built environment

SURNAME: Zhang
NAME: Fengyan
STUDENT ID: 5462150
STUDY PROGRAMME: Geomatics for the built environment

FRAUD AND PLAGIARISM AWARENESS

We confirm herewith that we have read and we are aware of the TU Delft rules (and consequences) regarding fraud and plagiarism, as written in: Fraud & Plagiarism.

1. Introduction

The goal of this assignment is to convert a BIM model(in IFC format) to a CityJSON file. Main steps of our converting methodology are derived from the suggested guidance, based on which a few modifications are developed to help us to proceed. Adjustments have been done mainly in the process of obtaining the big nef polyhedron, towards which two different approaches are implemented. As for acquiring the geometries and writing to city json file an agreement(will be expressed in the chapter 5) has been reached. The workflow is illustrated in the figure below(see Fig. 1), and the details will be explained in the corresponding chapters.

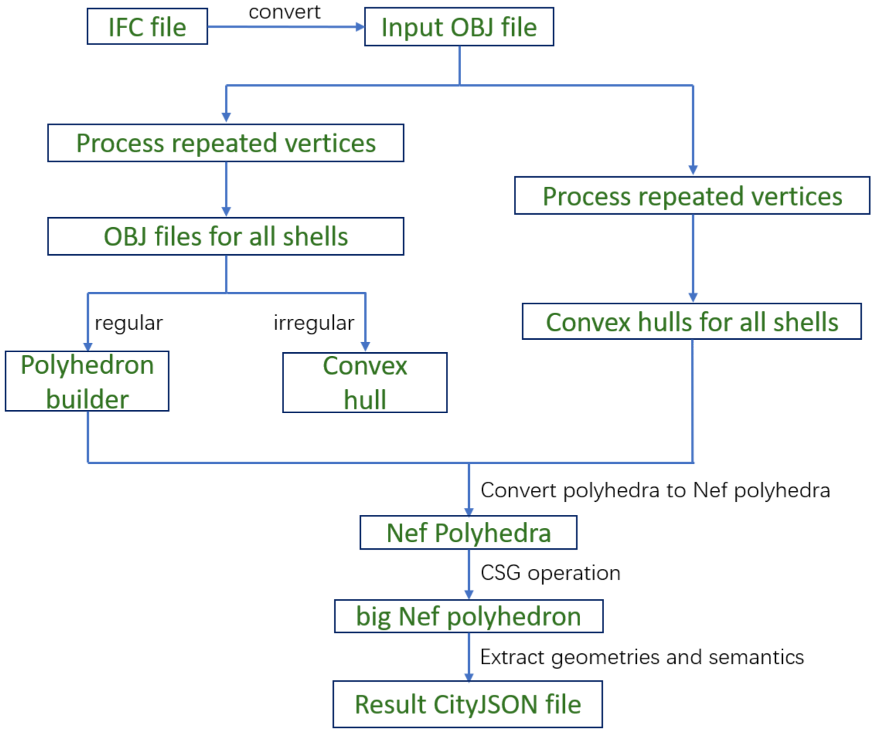


Figure 1 : Workflow

2. Convert IFC File

We choose one KIT model as our target file and use IfcConvert for converting. *IfcSlab*, *IfcDoor*, *IfcWindow* and *IfcWall* are selected as the major converting components. The command is as follows:

```
IfcConvert --include+=entities IfcSlab IfcDoor IfcWindow IfcWall --weld-vertices --orient-shells --validate KIT.ifc KIT.obj
```

(*Special thanks go to Leon (5605822), he helped us to derive this command*)

It should be noted that `--weld-vertices` command is used for roughly reducing the repeated vertices in the converted OBJ file, however there are still quite a few redundancies which need to be processed in our code. `--orient-shells` command is mainly for ensuring the correct orientation of surfaces, though this needs to be re-checked (can be done through MeshLab). And `--validate` command could help for verifying the correctness and completeness of the converted file. The result of this step is the OBJ file named *KIT.obj* – used as the input file of our program.

3. Obtain Polyhedra

Since there are many repeated vertices existing in the input OBJ file, removing duplicates is necessary, for the polyhedron builder requires non-repeated vertices. After cleaning the file, shells need to be filtered out and corresponding polyhedra need to be constructed. It should be noted that each polyhedron represents one shell.

Two different approaches are carried out to obtain the polyhedra. They will be explained respectively as follows.

3.1 Build Convex Hulls Directly

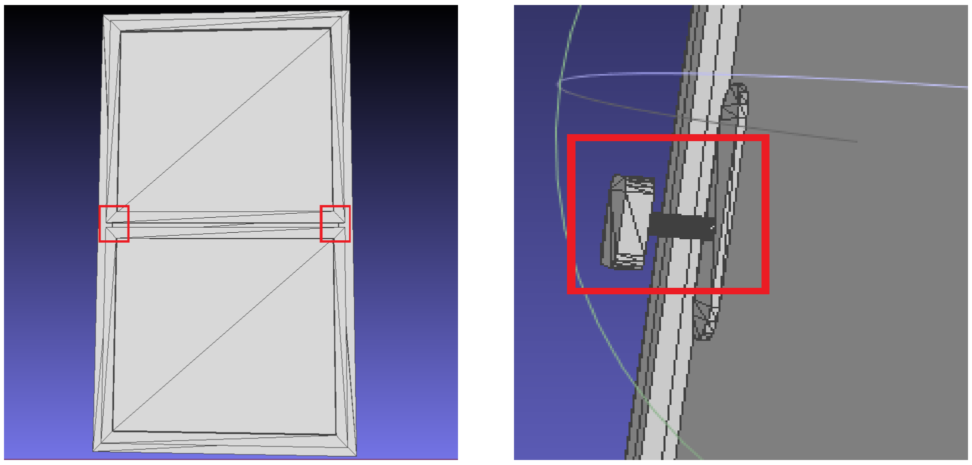
In order to read the file more easily and reduce the intermediate steps, a function to read the whole OBJ file at once is designed. The function processes in shells, storing points and faces in their corresponding containers. Whenever finishing reading a shell (i.e. the word currently reading in is "s"), the duplicate points are removed in the stored points and facets, then polyhedra can be built by constructing the convex hull of points, and finally the container is emptied and ready for the next shell's points storage.
The advantage of this approach is that the file can be very easy to read, no need to split and store the shells in OBJ format, and at the same time to get the needed polyhedra. The disadvantage

also lies in the inability to automatically distinguish shells with regular (relatively simple) geometries and irregular (relatively complex) geometries, therefore polyhedra can only be built by creating convex hulls for all of the shells. This method has been tested, yet the full procedure is still under construction. The following approach is chosen as our main method.

3.2 Build Polyhedra and Convex Hulls Respectively

Since the polyhedron builder can be picky, after the duplicate processing, faces should be given a guarantee of having consistent orientations as well. In order to check the consistency, the input OBJ file, which stands for the whole building, is firstly decomposed to multiple shells. These shells are stored and exported as intermediate OBJ files. With such behavior all of the shells being operated can be visualized in MeshLab and if any errors exist, they can be easily identified. However, this method is merely suitable for a relatively simple building – for checking each shell's shape and status can be time consuming.

After having the decomposed OBJ files, shells with regular (relatively simple) geometries can be passed to polyhedron builder for constructing the needed polyhedra, as for shells having irregular (relatively complex) geometries, corresponding convex hulls are selected for eliminating the errors which will be yielded if these shells are passed to the polyhedron builder. In practice, these irregular shells are mainly doors and windows, see figures below:



(a) Window(connecting parts) *(b) Door (door knob)*

Figure 2 : Complex structures in some shells

Through this approach polyhedra can be constructed from all of the shells, and meanwhile geometric features of shells can be preserved as much as possible.

4. Obtain the Big Nef Polyhedron

The constructed polyhedron can be easily converted to a nef polyhedron as long as it is closed. After having the entire nef polyhedron set, CSG(Constructive solid geometry) operations can be performed to merge the nef polyhedra into a big nef polyhedron with multiple shells. In our case, this is done by CGAL Boolean point set operations, and the `union operator +` is mainly used for joining the nef polyhedra together.

A discovery is found that if the convex hulls representing all of the shells are used, the obtained big nef polyhedron is non-simple, indicating it is a non-2-manifold. However, if the convex hulls and polyhedra built through polyhedron builder are both used, the result big nef polyhedron is simple, which means it is a 2-manifold. One possible guess is that the convex hulls and polyhedra obtained in approach 2 are built from the decomposed OBJ files, and in the process of creating files there may be a loss of precision in point coordinates, to which CGAL can be highly sensitive, therefore trivial changes in coordinates could cause different results returned by `is_simple()` function of the big nef polyhedron. Another hypothesis is that the big nef polyhedron is not fully and strictly closed, the most likely locations are eaves and roofs and their junctions with the building. In contemplation of erasing the possible small gaps, 3D Minkowski Sum has been experimented, see the figure below:

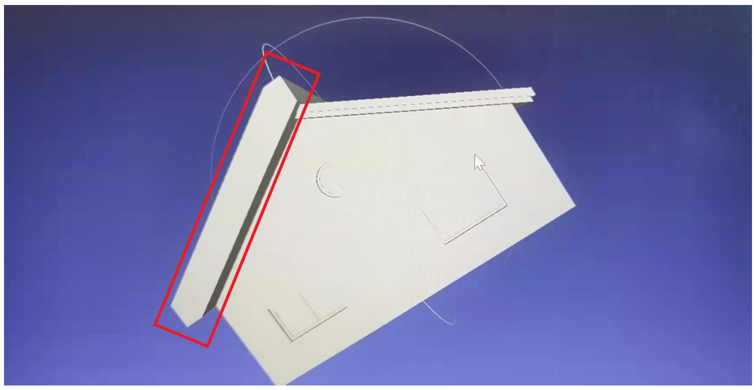


Figure 3 : Minkowski sum of one roof and a 1x1 small cube

Minkowski sum expands the boundaries of the target object and thus has the ability to fill the potentially existing small gaps. Multiple combinations have been tested, however the result still remains unchanged (the big nef polyhedron is simple).

Though further exploration may need to be performed to explain the above situation, a big nef polyhedron is still attained and it will be later verified to contain the shells needed to extract geometries.

5. Extract Geometries

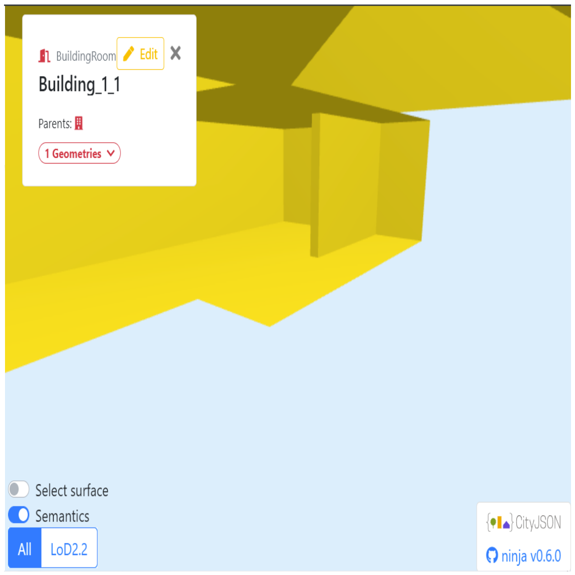
The main goal of this step is to obtain the vertices of each surface and extract the corresponding semantics.

5.1 Vertices Extraction

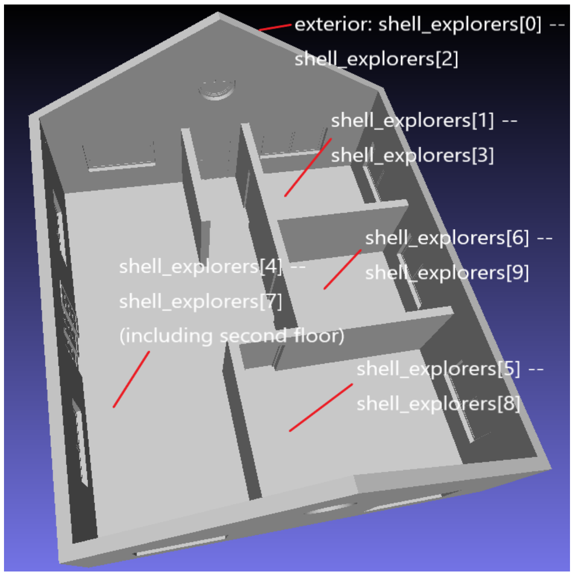
The shells in the obtained big nef polyhedron contain the information of all rooms and the building itself. The structure `Shell_explorer` provided in the guidance is used for extracting vertices of each halffacet when traversing each shell. It should be noted that in the process of attaining vertices, all of the shells in the big nef polyhedron are visited, which means there can be redundancies in the obtained vertices list, for each volume (in our case it can be roughly considered as each room) can have multiple shells, specifically through our test each room corresponds to two shells.

5.2 Room Identification

Due to the ambiguity of shells, the rooms they actually represent are hard to identify. Although through the `volume->marker()` the status of volumes can be printed and the volume standing for the whole building can be filtered out, this attribute is not applicable to the shells. In order to clearly recognize which shells stand for which rooms, a decision has been made to firstly write each shell to a CityJSON file and visualize it using ninja. In such manner the connections between rooms and shells can be easily and precisely established, however this approach is only viable when the number of shells is not too large, please see the figure below:



(a) Visualize one shell through ninja



(b) Connections between rooms and shells

Figure 4 : Establish the connections between rooms and shells

As is shown in Fig. 4(a), a single shell has been written to the CityJSON file and thus visualized in ninja, the corresponding room can be identified in Fig. 4(b), which is marked with shell_explorers[4] - - shell_explorers[7], that being said, in our shells list, the 4th shell and the 7th shell stand for the highlighted room in Fig. 4(a). This approach goes to all of the rooms in our building, and the connections are illustrated in Fig. 4(b).

5.3 Semantics Assignment

For the exterior surfaces of the building, three main different semantics – *GroundSurface*, *WallSurface* and *RoofSurface* need to be assigned. This process can be done through multiple approaches, for instance, using the code we wrote for hw02 to calculate the normals of surfaces, based on which the semantics can be determined, or using the CGAL orthogonal vector (*special thanks go to Yuduan (5625483), she kindly shared this useful link to us*) to perform akin operations. Another thought is to use the locations of surfaces, a feasible way is to use the vertex coordinates, which can be easily denoted in MeshLab, to estimate the locations. These thoughts are experimented but not adopted in the end, the reasons will be explained as follows respectively.

5.3.1 Using Normals

Using normals is the most universal and automatic method, though in our case, some modifications have to be done to fully recycle the code written for hw02, and the more important thing is that there are small and thin triangles existing in surfaces, because of which calculating normals can be faulty and inefficient.

5.3.2 Using Coordinates to Estimate Locations

Although coordinates of a certain surface can be easily obtained in MeshLab (using Get Info button to select one specific surface and coordinates of incident vertices will pop up), it can be cumbersome to develop a relatively complete system to classify the different face locations. For instance, the locations of connecting vertices between surfaces need to be taken into account, and for each surface, it may be necessary to limit the *xyz* coordinates of all incident vertices at the same time for assigning semantics.

Due to the limitations above, another achievable but not completely automatic method has been implemented. For our building, the total number of exterior surfaces (which is 27 in our case) is small enough for us to assign each surface a specific semantic (i.e. *RoofSurface*) and then write this surface to the result file and visualize it using ninja or CityJSON Loader plugin in QGIS. According to the visualization, the location (*Ground*, *Wall*, *Roof* or others) of the colored surface can be recognized and thus the semantics of this surface can be assigned. Through this way each surface can get a unique semantic and if this is done with carefulness, semantic ambiguity or omission of small surfaces can be avoided. The major disadvantage of this method is that it is only designed for one specific building, and if too many exterior surfaces are given, this method can become undoable.

To sum up, obtaining vertices and semantics of surfaces is mainly done with our code and manual help. The code and manual work are integrated as a whole after this step is completed, that being said, the program can execute the whole process automatically in the end. However it should be warned that the program is suitable and robust for our input OBJ file only.

6. Result & Analysis

Multiple websites and tools are used for verifying and visualizing our result, which would be described as follows, meanwhile some analysis is also given.

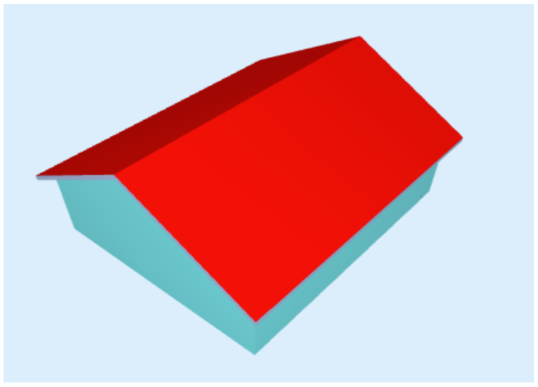
6.1 Result Verification

Our json file consists of one *Building*, one *BuildingPart* indicating the exterior surfaces, and 4 *BuildingRooms* representing all of the rooms. Inspired by the hw02*, Solid* is selected as the type of geometry. And cjval is used for verifying the schema, according to which the file is 100% valid, the criteria is honored.

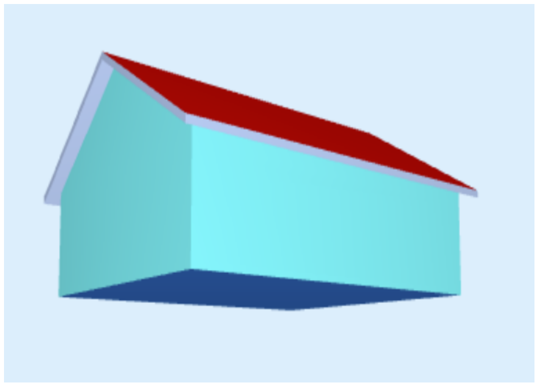
6.2 Result Visualization

ninja is used for the visualization of the result file, please refer to the following figures:

6.2.1 Overview



(a) RoofSurface and WallSurface



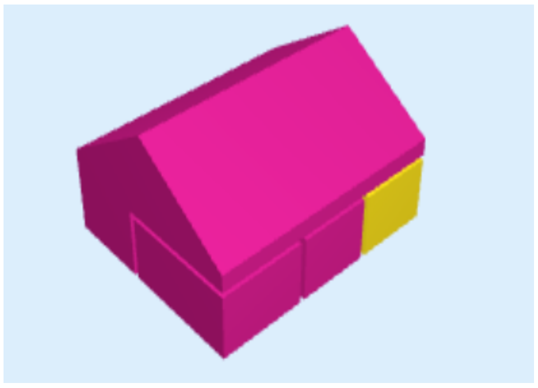
(b) RoofSurface, WallSurface and GroundSurface

Figure 5 : Visualization of result json file

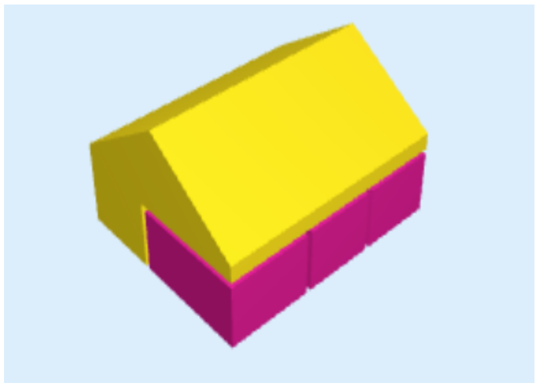
In the figure above, red indicates the type *RoofSurface*, wathet blue represents for the type *WallSurface*, and dark blue stands for the type *GroundSurface*. Since the doors and windows are expressed as the corresponding convex hulls, the shapes of them can not be clearly seen in the result file, yet the other originally geometric characteristics of the building are preserved. For instance, the two roofs keep their original shapes (solids) instead of being represented with plane surfaces. And only the upper surfaces(see Fig. 5(b), colored in red) are assigned the semantic *RoofSurface*, the flanks and lower ends on both sides have no specified semantics, which are colored in gray.

6.2.2 Internal Structures

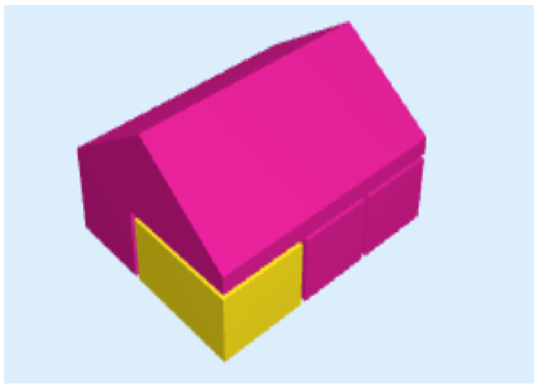
Since selecting/deselecting certain surfaces can be difficult either in ninja or CityJSON Loader plugin in QGIS. Another version of the result json file which excludes the exterior surfaces has been derived, the internal structures of the building can thus be obviously observed, please refer to the figures below, the selected room is highlighted in yellow:



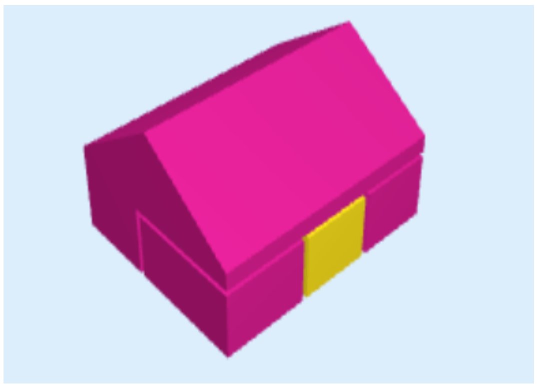
(a) Room 1



(b) Room2 (including the second floor)



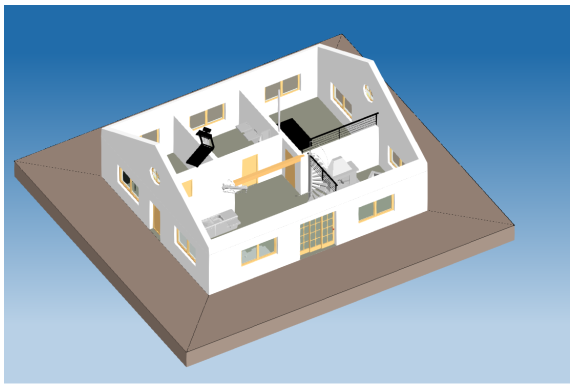
(c) Room 3

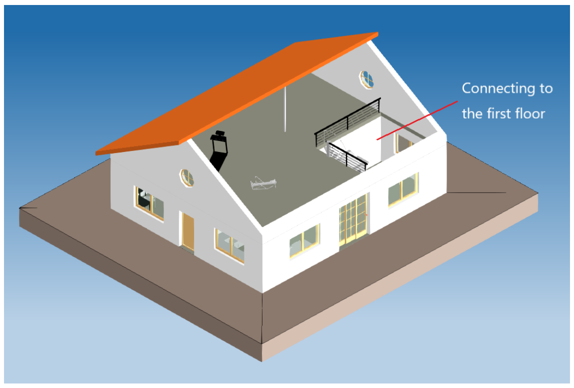


(d) Room 4

Figure 6 : Internal structures

Comparing the result with the original IFC file (see Fig. 7), a conclusion can be drawn that the converting process is basically successful and the distribution of rooms follows the original locations in the building. It should be noted that, the second floor is connected to a room on the first floor (see Fig. 7(b)), and this is why the shape of Room 2 looks a bit strange.


(a) First floor



(b) Second floor

Figure 7 : Internal structures (Open IFC Viewer)

6.3 Improvements

6.3.1 Geometric Primitives

val3dity is used for verifying the geometric correctness of primitives in the result json file. Two errors have been thrown as follows:

CONSECUTIVE_POINTS_SAME	Points in a ring should not be repeated. This error is for the common error where 2 consecutive points are at the same location.
RING_SELF_INTERSECTION	The self-intersection can be at the location of an explicit point, or not. This case includes rings that are (partly) collapsed to a line.

Some experiments have been carried out, i.e. printing out the coordinates of error vertices according to the error report from val3dity, as is shown below, each line stands for the *xyz* coordinates of one error vertex, yet their coordinates don't seem to be exactly the same. Another attempt is to identify the self-intersection ring, based on our process the self-intersection case is most likely due to a very thin triangle surface.

```
errors - consecutive points:
(4.995, 0, 0)
(-1.60882e-22, 9.99999, 3.2)
(-2.01859e-21, 1.34465e-37, 3.2)
(12, 5, 6.08675)
```

The time is not enough for us to fully eliminate these two errors and the proper method needs to be further studied.

6.3.2 Windows & Doors

Due to the usage of convex hulls, the windows and doors can not be identified in the result json file, which may lead to the difficulties in some certain applications. For instance, distinguishing the front or back *WallSurface* is not viable in our result file, and it is impossible to judge the orientation of the doors and windows.

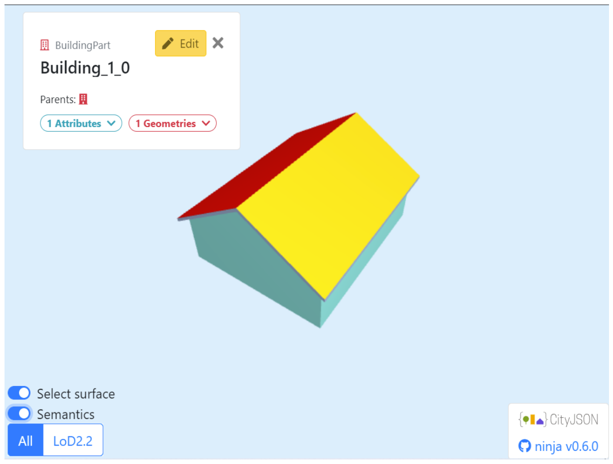
Doors and windows can be important attributes in some scenarios. A possible solution is that when writing to a json file, use the convex hulls of doors and windows as the inner shells of the geometry of exterior surfaces, in this way they should be able to be observed and captured during the visualization.

6.3.3 Attributes Enhancement

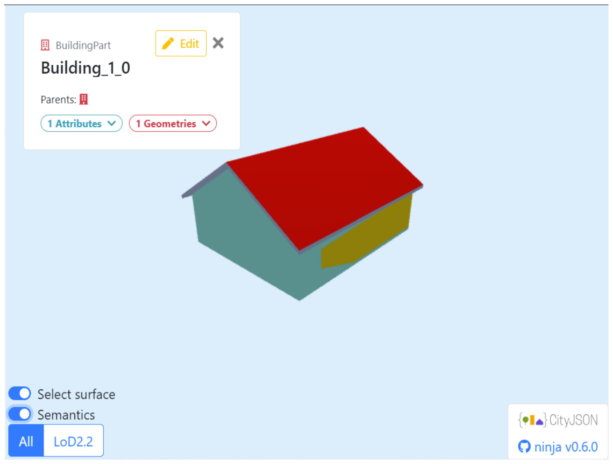
Principally the result json file can be used as the input of hw02, a few attributes can be calculated, i.e. the volumes of the building and rooms, the area and orientation of *RoofSurfaces* and the number of floors of this building. These can be achieved if more time is given.

6.3.4 Surface Selection

In our building, some faces are divided into several irregular parts, the reason why this could happen can not be clearly defined. It may come from the originally irregular geometric primitives in the input OBJ file, or due to the construction of the convex hulls, please refer to the figures below (the selection is highlighted in yellow):



(a) Selection of RoofSurface



(b) Selection of WallSurface

Figure 8 : Surface selection in ninja

The roof as a whole can be selected in Fig. 8(a), whereas in Fig. 8(b) only a part of the whole wall is selected. In real applications this may lead to the inconvenience from the perspective of users. A feasible way to remove this could be merging the small parts belonging to one semantic surface.

6.3.5 Code Optimization

In the process of removing repeated vertices, the container unordered_map could have been used for accelerating and better organizing our code. Since our input OBJ file is relatively small, the performance impact is negligible, yet when it comes to a large file, unordered_map can help us significantly reduce the time complexity.

7. Reference

(1) GeoBIM:
https://3d.bk.tudelft.nl/projects/geobim/#open-source-software
(2) Automatic generation of CityGML LoD3 building models from IFC models:
http://resolver.tudelft.nl/uuid:31380219-f8e8-4c66-a2dc-548c3680bb8d

8. Workload

We discuss together and carry out the two different processing methods(for obtaining the nef polyhedra), the report is written together and the workload is evenly divided.

9. Appendix

Our git repository for this assignment:
Build polyhedra using approach 1: https://github.com/YitongXia/BIM_processing.git
Build polyhedra using approach 2: https://github.com/SEUZFY/BIMConvertToGeo

10. Thanks

Special thanks for Leon (5605822), he helped us to derive the Ifc command.
Special thanks for Yuduan (5625483), she kindly shared this useful link to us: CGAL orthogonal vector.
Special thanks for Siebren (4880412), he kindly shared the useful link, cgal-discuss: iteration over facet of volumes to us, inspired by which our method of extracting vertices can be derived.

Special thanks for Ken Arroyo Ohori (k.ohori@tudelft.nl). His kindly given guidance helps us to develop our code and during the assignment, our questions are always answered quickly and accurately, without which this assignment would not have been possible to be delivered.