Delft University of Technology

3D Modelling
GEO1004

Assignment 03

PROCESSING A BIM MODEL USING CSG

Authors:

Eleni Theodoridou (5626544),
Guilherme Spinoza Andreo (5383994),
Fabian Visser (5433916)

April 4th, 2022

Table of Contents

Link to projects github (https://github.com/guispinoza/geo1004.hw03)

Our method

In this report we will elaborate our implementation of the algorithm for processing a BIM model from the IFC format to CityJSON. The algorithm was developed in C++ and consists of manually preprocessing of the IFC file using IFCconvert to transform the BIM model into a simple OBJ file. Then, by creating the convex hull of the shells of the building, we convert the OBJ into multiple CGAL Polyhedra. Then our code turns these polyhedra into Nef Polyhedra and creates a “Big” nef polyhedra for the space filled by the building’s floors, walls, roofs and shells. Then we extract the geometries of the building’s exterior surface and individual rooms which are written to a CityJSON file. The validity of the final output was analysed by using CityJSON validator and further discussion on the results and the encountered issues are mentioned as well.

The code parses the obj file (KIThouse.obj) and converts it into multiple CGAL Polyhedra by generating a CGAL Convex Hull from the vertices and faces of each shell. Then our code transforms these Polyhedra into Nef Polyhedra, and creates a “Big” nef polyhedra for the space filled by the building’s floors, walls, roofs and shells. Then we extract the geometries of the building’s exterior surfaces and individual rooms which are written to a CityJSON file.

Implementation details

A - IFC preprocessing (IFC to OBJ)

The initial testing and development of the OBJ file was done using the IfcOpenHouse and then applied to our selected file, the KIT: AC11-FZK-Haus-IFC.

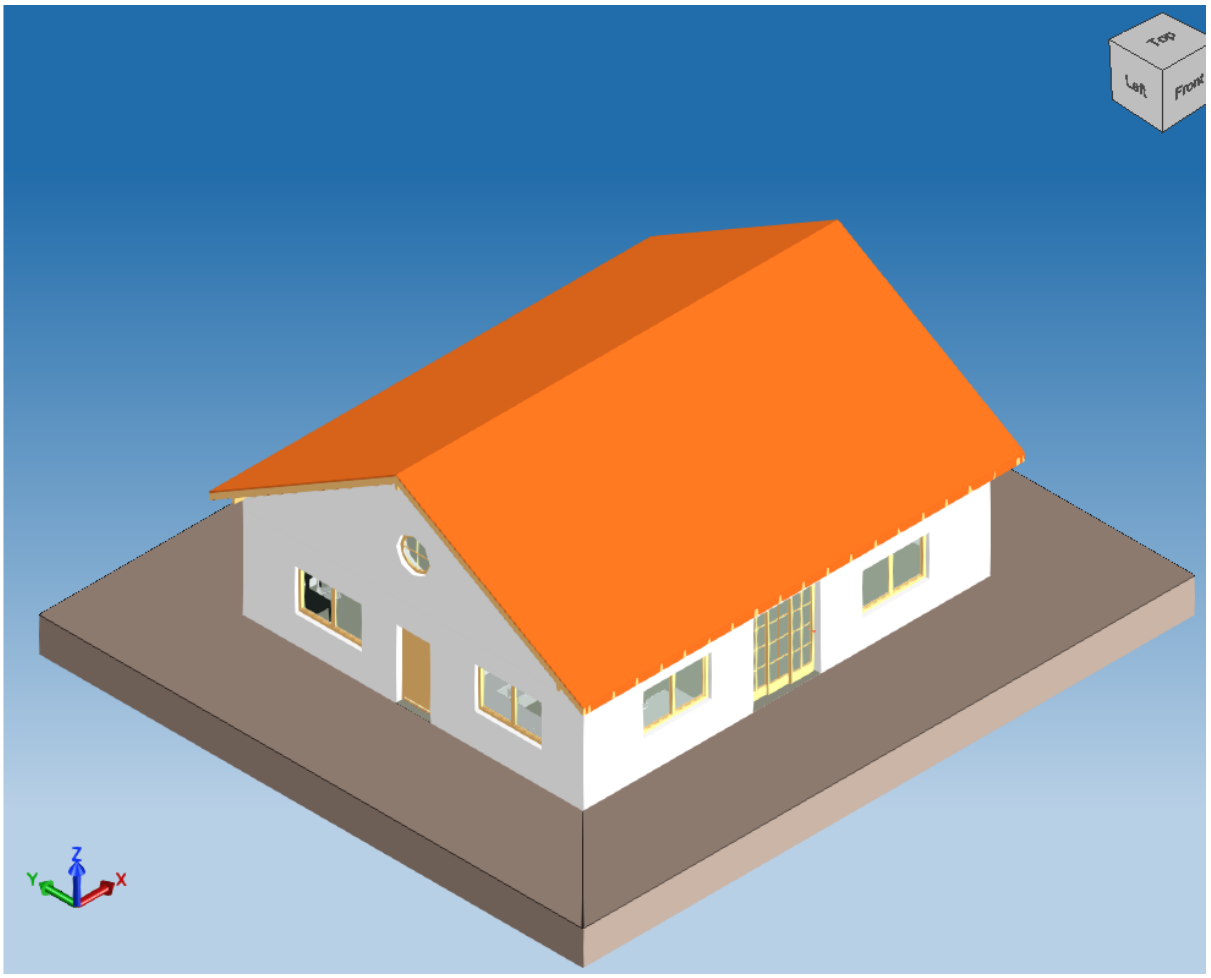

Figure 1: KIT: AC11-FZK-Haus-IFC

The preprocessing of the was done using IFCconvert, an application from Open Shell which converts the IFC file structure to an OBJ file. This was done with the following commands:

```
IfcConvert.exe 231110AC11-FZK-Haus-IFC.ifc KIThouse.obj --weld-vertices
--orient-shells --include+=entities IfcWallStandardCase IfcSlab
--validate
```

Using the *--weld-vertices* command allowed us to limit the OBJ conversion to contain only unique xyz-triplets, which should result in a manifold mesh. The *--orient-shells* command was used to guarantee a consistent orientation for the surface normals for the faces of the duplex structure.
Initially the conversion was done using the entire geometrical model, however, because the elements of the original IFC contained self-intersecting surfaces, the *--include+=entites* was required to extract only the objects that composed the exterior shell of the IFC model. Since

the initial goal was to generate a convex hull, all unnecessary elements (windows, doors, spaces, coverings, stairs and beams) to avoid conflicting geometries.

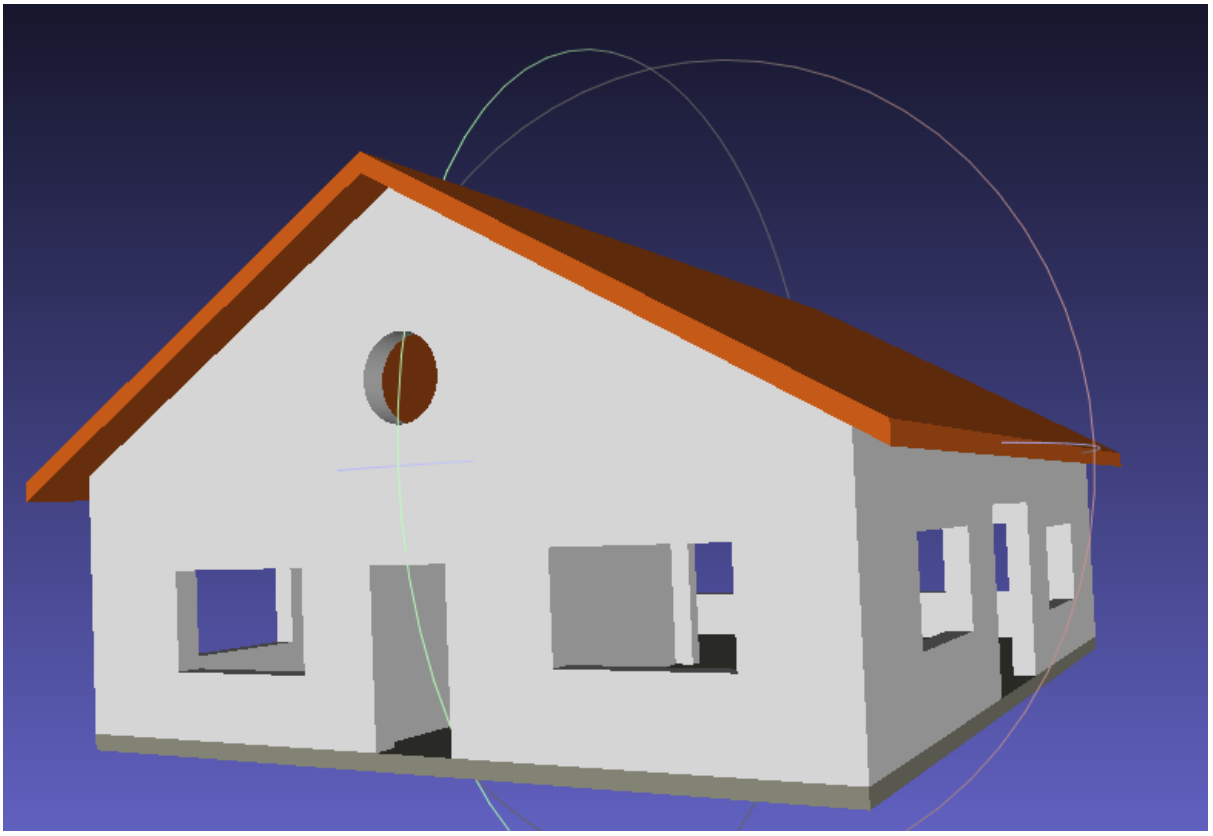

Figure 2: Post processed model (in .obj)

B - From OBJ to Nef polyhedra & Big nef polyhedron

Reading the .obj file

We start by reading the .obj file. The structure of the OBJ file is very consistent, which is first defined by a polygon group *"g",* that represents each object for our case. All the objects are stored in a vector. Next the shells of each object are read and can be distinguished according to their different *"usemtl"* ids. The shells are added to a vector of each object. The faces are found as they are defined by *"f"* and pushed in a vector of the shell of each object. Finally, the vertices are all found as they are defined with *"v"* and they are stored in a vector. They do not need to be checked for duplicates as the *"welded"* version of the obj file was used.

Creating the Nef Polyhedra

For every shell, one CGAL polyhedron is defined and the shell's vertices are stored in a vector *"verts"*. The *"verts"* are used for the "`CGAL::convex_hull_3`" and the polyhedron is used to hold the convex hull. Using the convex hull, a closed polyhedron is effectively created. Then a CGAL nef polyhedron is created from each closed polyhedron and is pushed in the vector of the nef polyhedra, named *"polyhedra"*. The decision to use the convex hull instead of the polyhedron builder was due to the fact that it was easier to simplify the geometry of the model, where the windows, doors and complex objects would generate non-manifolds, even when welding the vertices.

Big Nef Polyhedron:
For this step the individual nef polyhedra that were created for every shell of each object are merged together to create the big nef polyhedron, by using the Boolean union operation. This operation is necessary to combine all the walls, leaving us with one house polyhedron that encloses each room.

Pros & Cons of our method:
The aforementioned method with the construction of the convex hull, provides quick results without having to be aware of the specific errors that occur and cause unclosed polyhedrons, as well as allowing us to create separated interior volumes which is useful when defining the BuildingRooms. On the other hand, according to our tests implemented using the ERDC:Duplex_Apartment_Model, the results are different than expected. In the surface where the footing of the house extrudes from the floor, holes are created and the convex hull method does not consider them, therefore overlapping triangles are created over these holes. Therefore, the method is not universal and could cause problems in specific types of more complicated face structures.

C - Extracting the geometries

Now that we have our big nef polyhedron, we use a Shell Explorer to visit the facets of each shell. Using CGAL_forall_volumes, we can travel through the volumes of our big nef polyhedron. It contains 3 types of volumes; the exterior volume, the interior room volume, as well as the in-between space of the polyhedron. These each hold a collection of shells which we can parse using CGAL_forall_shells. The exterior and interior volumes both contain one shell, while the in-between volume contains multiple shells, from the outside shell and each shell enclosing the room. We noticed that the exterior volume is always first in the list, and so create a Building CityObject from that and then we apply the Shell Explorer. For the next volumes we check that they contain only one shell, and create a BuildingRoom City Object and apply the Shell Explorer.

The Shell Explorer contains the following variables; a boolean to check if we are in the first (and therefore outer) shell, a vector to store the global vertices, a vector to store the faces' indices, and a vector to store the semantics of the outer shell.

For each shell we visit its facets. We travel through each shalfedge of each facet, taking its source vertex. The vertex is searched for in the vector vertices and added if not there, and we find the index of this vertex. We finally store these indices in a vector and store that into our vector for faces' indices.

For the outer shell, we also want to define the semantics, whether a facet is part of the roof, a wall, or the floor. Using CGAL, we convert the facet into a plane, get its normal's direction, and using the z value, describe a semantic; if z is positive, the facet is part of the roof, if it is negative it is part of the floor, if it is 0 it is part of the wall. We chose these values due to the building having pretty simple geometry and therefore not requiring any detailed calculations for these semantics.

After storing these values into the vectors, we can call these vectors and write them to CityJSON.

Pros & Cons of our method:
This method works well, as we are able to access only the shells we need, and cycling through their facets and the vertices of each facet is not very complicated. One problem is finding the index of the vertices. We use std::find, which travels through the whole vector and therefore has a linear time complexity, which can likely be improved. When calculating the semantics, we assume that horizontal is in between [-1e9,1e9]. For another example, a slanted wall would therefore not be calculated as a WallSurface. A more robust calculation can be made, but as the objective of this assignment is to calculate it for our chosen house, we left it as is.

D - Writing to CityJSON

The output for the final CityJSON file was done using the shell_explorer function, since this was necessary to define and enrich the shells, where the first shell encountered was attributed as the parent Building node of the BuildingRooms. Then by iterating through each interior shell, we were able to output them into separate BuildingRooms, children of the Building CityObject. The geometry type that we applied was the MultiSurface, since it can avoid the overlapping of different building parts. Included in the Building CityObject are the semantics. An extra empty space between the roof and the floors exists, which we remove by skipping over the BuildingRoom created by this space (BuildingRoom 4).

Pros & Cons of our method:
The method we applied to write to produce the CityJSON works as intended, where we successfully implemented the required encoding schema for Building and BuildingRoom, however, because we “hard-coded” the traversal of the inner shells to stop at BuildingRoom 4, our code cannot be used for any other KIT IFC models. A pro decision was to use MultiSurfaces, since we only need our CityObjects to represent the shells. Defining them as solids is also a possibility, but requires including an empty space inside of the outer solid for the BuildingRooms. One problem with our CityJSON file is that there are duplicate vertices. We are not sure where these come from, but these should be removed for a more consistent file.

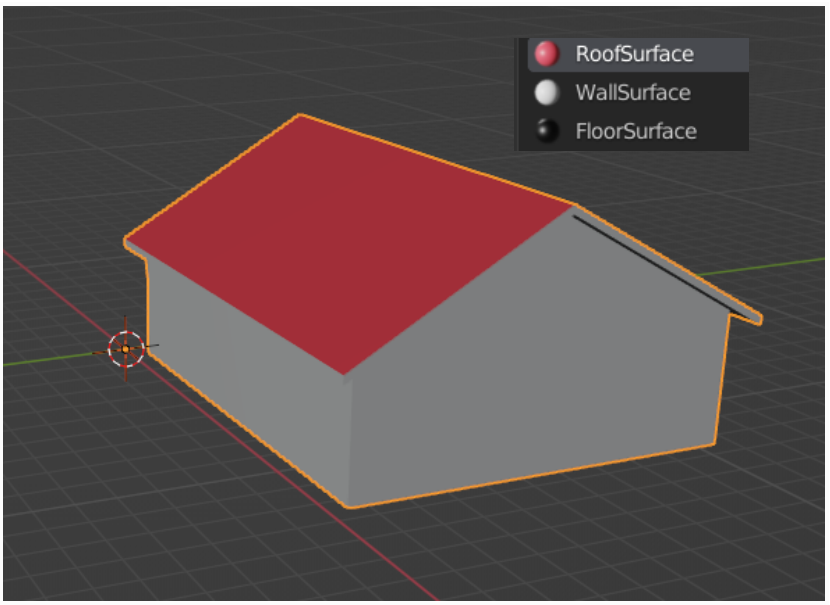

Figure 3: Visual output of the city json output of the semantics of the surfaces

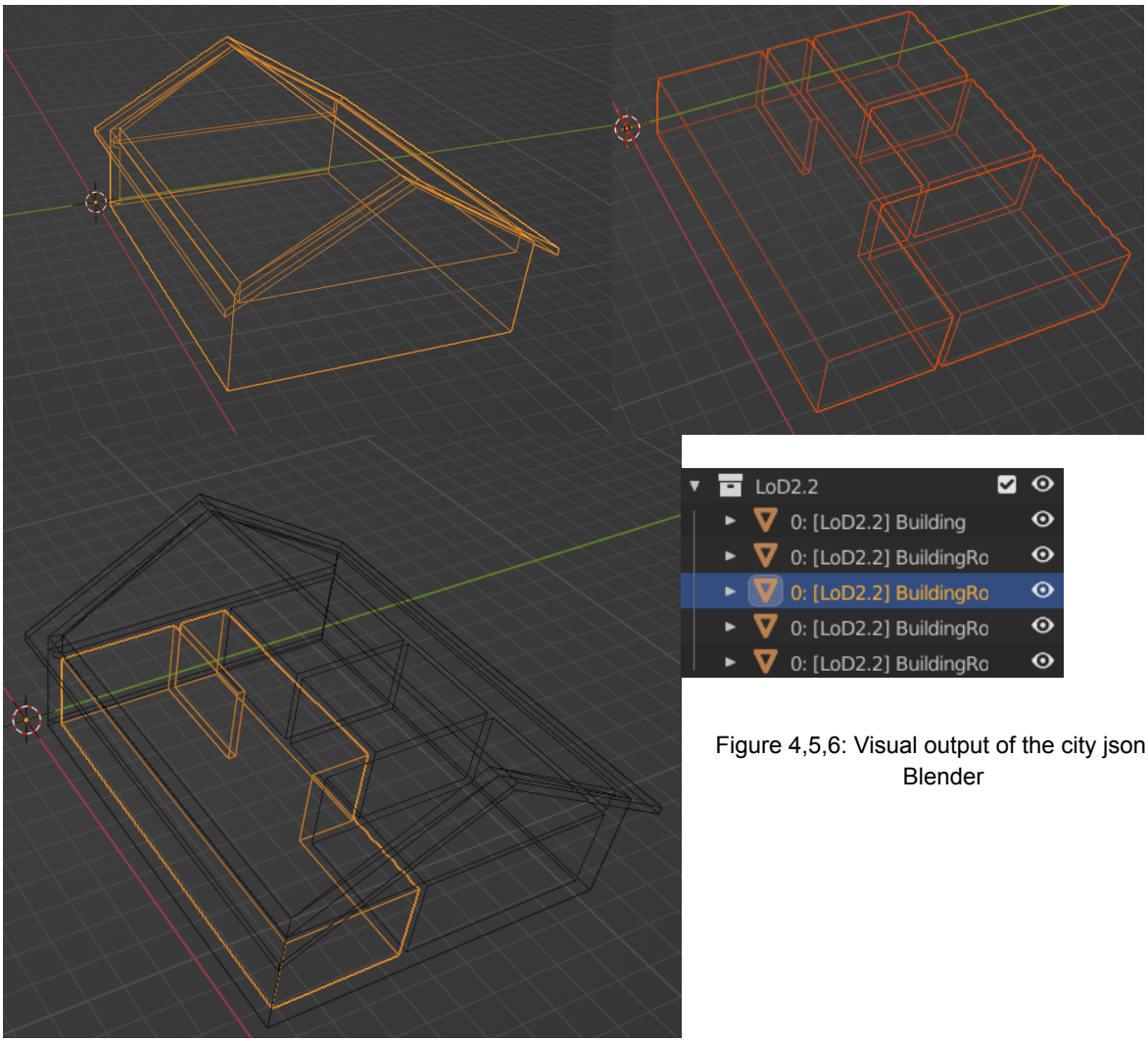

Figure 4,5,6: Visual output of the city json in Blender

Workload distribution

Most of the tasks were divided equally amongst the group members and each part required help and input from all group members.