Processing a BIM model using CSG

3D modelling of the built environment

Dmitri Visser (4279913), Katrin Meschin (5163889), Cynthia Cai(5625483)

April, 2022

MSc Geometics, TU Delft

1 Introduction

In this assignment, we convert a building information Model to a geographical 3D city model in CityJson format. The process includes five steps:

1. Convert the IFC file to an OBJ file
2. Load each shell in the OBJ file into a CGAL format Nef polyhedron;
3. Combine the Nef polyhedra into a single big Nef polyhedron representing the space filled by the building's floors, walls, windows and so on
4. Extract the geometries representing the building's exterior surface and individual rooms
5. Write the geometries to a CityJSON file.

The tools that we used include: CLion, CGAL, JSON for Modern C++ library, IfcConvert and cjval.

The program was first developed using a relatively simple BIM model of IfcOpenHouse (Krijnen, 2012).

2 From IFC to OBJ

We used Open House BIM for the initial test. Ifc is the most common data format for building information models, but its complicated structure makes it hard to load the objects in the code. Compared to that, obj files have a much simpler structure. However, the support of obj files is mostly limited to basis geometric entities like geometric vertices, texture coordinates, vertex normals and polygonal faces. We used IfcConvert software to convert the original file into an OBJ format. The tool was also used to remove duplicate vertices, as the polyhedron builder that we used in next step only works well if the vertices are unique. The command is:

```
.\IfcConvert.exe --orient-shells --include+=entities IfcRoof IfcWall IfcFooting
↪ IfcWindow IfcDoor --weld-vertices --validate .\IfcOpenHouse_IFC4.ifc output.obj
```

3 From OBJ to Nef polyhedra

The goal of this step is to convert every shell into a Nef polyhedron. In mathematics, Nef polyhedra are the sets of polyhedra which can be obtained from a finite set of halfplanes (halfspaces) by Boolean operations of set intersection and set complement. The CGAL library provides an interface to perform geometrical manipulations on Nef polyhedra. In order to obtain Nef polyhedra, the geometric data has to be parsed from the obj files and converted by our algorithm.

3.1 Parsing .obj files

First, the obj files are read, parsed and loaded into memory in our own data structures by `read_obj()`. We keep track of the id's by storing them as a member variable of each `Vertex`, `Face`, and `Shell`

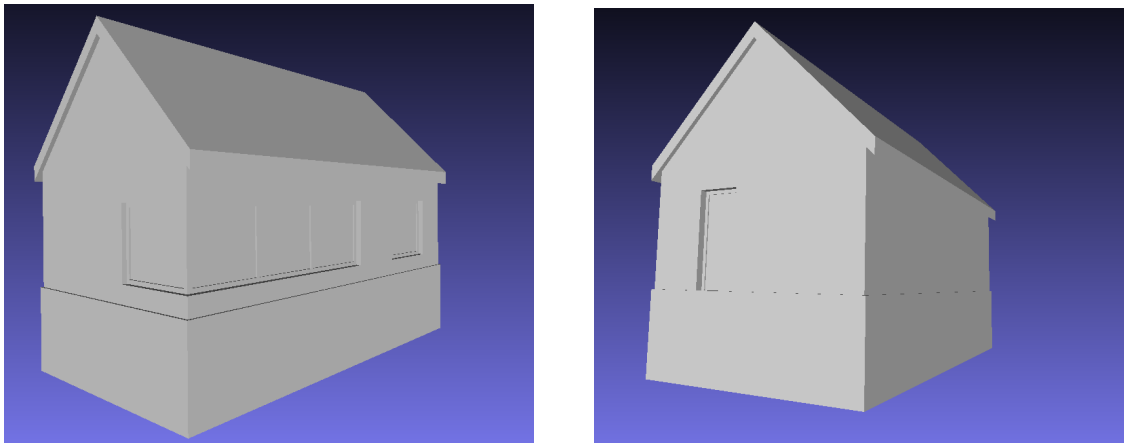

Fig. 1: Open house test dataset as polyhedrons

(groups in the obj file). However, to quickly navigate through the geometry, these should be linked together by pointers. The pointers should be set, after the entire obj file is parsed, as the list of vertices, faces, and shells grow continuously during parsing. Setting the pointers in the beginning would lead to mismatching pointers. The linking of the pointers is done in `link_pointers()`. Below follows an example for relinking pointers in faces:

```
// Link pointer in faces to vertices
for (auto &face : faces) {
    for (auto vertex_id : face.vertex_indices) {
        face.vertices.push_back(&vertices[vertex_id]);
    }
}
```

3.2 Polyhedron Builder and conversion to Nef Polyhedron

Each `Shell` was loaded into a CGAL Polyhedron_3 (see Fig 2) using the `Polyhedron_incremental_builder_3` using a custom struct. However, simply traversing all faces would add too many vertices to this struct, as the vertices can be part of multiple faces. To ensure no duplicate vertices are stored in the struct, the vertices are first stored in a `std::map`, using the original id (from the obj file) as key:

```
// Create vertices subset consisting solely of non-duplicate vertices
std::map<unsigned long, Vertex*> vert_map;
for (auto const &face : shell.faces) {
    for (auto const &vertex : face -> vertices) {
        vert_map[vertex->id] = vertex;
    }
}
```

Once the polyhedrons were obtained, they were converted into CGAL nef polyhedrons using the interface between the two. Since the class of Nef polyhedra is closed with respect to the topological operations, we examined the validation and closeness of each Nef polyhedron in the iteration process. The easiest way to process problematic Nef polyhedra is to replace them with their convex hulls. This may cause some accuracy loss, but is generally acceptable.

To visualize the extracted polyhedrons and nef polyhedrons, we wrote them in off files and imported them into meshlab for viewing.

4 The big Nef polyhedron

In this step, we merged the multiple Nef polyhedra obtained in the previous step into a single nef polyhedron. We initialized the big nef polyhedron as an empty nef polyhedron, then we iterated every single nef polyhedron to perform union operation (operator +) with the big nef polyhedron. Since it passed our closeness and validation examination, we didn't perform other operation of graphics transformation.

The Big Nef polyhedron of the Open House is a 2-manifold and has three volumes, including universe, filling, interior one, which conforms to topological constraints and reflects that it contains one exterior shell (the outer surface of the building) and a few inner shells (one for each room).

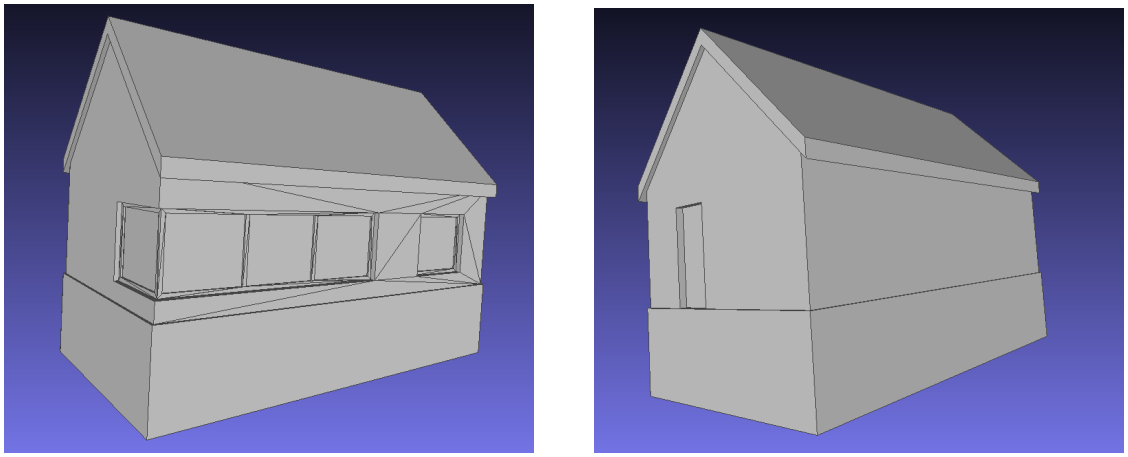

Fig. 2: Nef-Polyhedrons

5 Extracting geometries

This step is to obtain a set of simple geometries with boundaries and semantics. Starting from the single Big Nef polyhedron, we navigated through the Nef polyhedron 3 structure to extract the geometries of the different shells.

A custom struct `Shell_explorer` was written that contained the visitor pattern to go through all child entities of each shell. Again, since we are using a face-centered visitor pattern, we need to have a mechanism in place to make sure we do not extract duplicate vertices. `Shell_explorer` contains the following member variables:

```
1 // Create vertices subset consisting solely of non-duplicate vertices
2 std::unordered_map<std::string, Vertex*> vertex_map;
3 std::vector<Vertex*> vertices;
4
5 // store all the boundaries in this shell
```

```
std::vector<std::vector<std::vector<unsigned long>>> boundaries;

// store the semantics - surface type - of each face
std::vector<unsigned long> semantics;
```

This time, we store the coordinates as the key in the vertices map and only push non-duplicate vertices into a vector of vertices that delineate one face:

```
vertex_index = vertex_map[str_key]->id;
face.push_back(vertex_index);
```

We also assigned semantics to outer shells, which can be either GroundSurface, WallSurface or RoofSurface. To determine its semantics, we computed its orthogonal vector of the plane with CGAL : : orthogonal vector and obtained its orientation.

Coordinate	Orientation	Semantics value	Semantics number
(0,0,x)	horizontal	GroundSurface	0
(x,x,0)	vertical	WallSurface	1
the rest	the rest	RoofSurface	2

6 Writing CityJSON

We used Niels Lohmann's JSON for Modern C++ library to write our final file. We followed the schema of CityJSON 1.1, and the overall structure is similar to that of 3D BAG dataset. The first layer of json file contains CityObjects, metadata, transform and vertices. We stored the coordinates of vertices in vertices, and the geometries of each shell in CityObjects. The exterior surface and inner rooms are relatively recognized as BuildingPart and BuildingRoom.

Testing the output CityJSON of the OpenHouse in the ninja.cityjson.org web viewer shows decent results with the semantics of the surface types showing correctly.

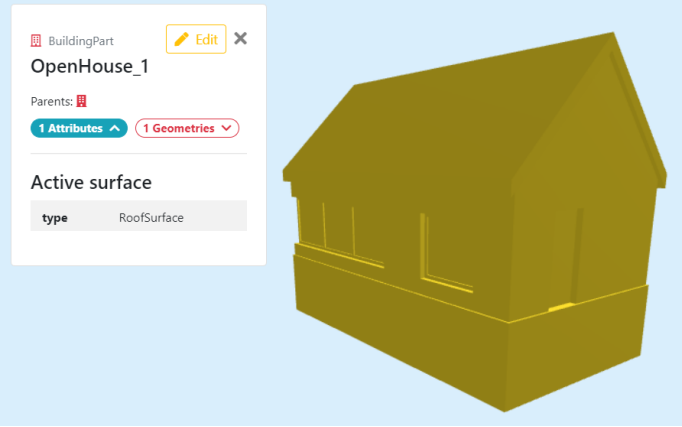

Fig. 3: The output of the CityJSON in Ninja web viewer with the semantic values of surface type showing

Looking closely at the semantic values, some errors become evident. For example, a horizontal part of the house's wall is marked as GroundSurface (Fig 5. This is because when assigning the semantics,

only the orthogonal vector of the surface was considered. This mistake could be solved if the elevation of the face or the fact that it is surrounded by wall surface would have been taken into account, which however due to time constraint of this project was not implemented.

Fig. 4: The horizontal parts of the house's wall surfaces were falsely given a semantic value as GroundSurface.

When testing the created CityJSON file with val3dity online processor, several issues were highlighted. The parent building shows error "CONSECUTIVE_POINTS_SAME" and the buildng rooms give an error "CITYOBJECT_HAS_NO_GEOMETRY". Removing those errors was not completed due to the time constraint of this project.

7 Testing the program with another BIM model

We choosed the KIT: AC11-FZK-Haus-IFC model from the Open IFC Model Repository as our test dataset. It is a simple furnished house. When converting it to obj file, we only extracted its door, window, slab(roof) and wall. However, a CGAL error about assertion violation was thrown. The explanation is 'impossible to decide which one is a visible facet (if any)'.

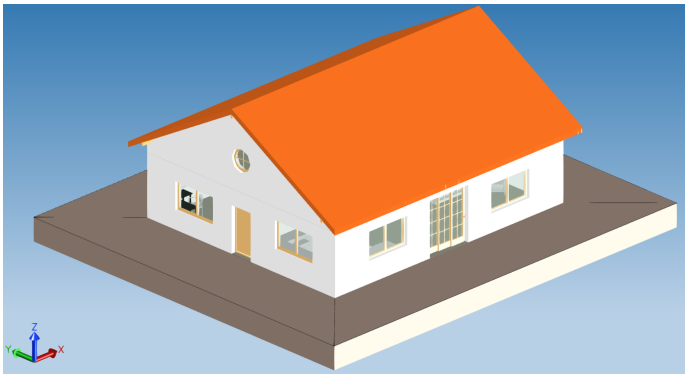

Fig. 5: the KIT: Ac11-FZK-Haus-IFC model

8 Summary of collaboration

The workload of this assignment was divided relatively equally between all three team members. However, each of us carried more weight in some of the steps as shown below:

- Input argument parse, Loading and parsing OBJ into memory, `geometry.h` skeleton, creating the polyhedra, dealing with duplicate vertices, debugging the shell explorer and the city.json output - Dmitri

- Processing the Nef polyhedra into a single big Nef polyhedron, extracting the semantics of each face in the Shell Explorer - Cynthia

- Validating the Nef polyhedra, writing the shell explorer and the geometries to cityJSON - Katrin