STAG: Smart Tools and Applications in Graphics (2021)
P. Frosini, D. Giorgi, S. Melzi, and E. Rodolà (Editors)

3D City Reconstruction from OpenStreetMap Data

S. Kaszuba[1] and F. Pellacini[1]

[1]Sapienza University of Rome, Italy

Abstract

Virtual city generation from real data is far from being straightforward for users, as it strictly depends on the application domain, amount of information available, and the adopted reconstruction techniques. Nowadays, reconstruction of virtual cities is of interests in entertainment, urban planning, emergency response and machine learning. To serve these applications, we have developed an open-source tool that can reconstruct cities at scale directly from OpenStreetMap data, that can perform full city generation in the order of hundreds of seconds.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ *Graphics systems and interfaces;*

1. Introduction

The growing interest in reconstructing large areas motivates the effort of researchers and practitioners in the generation of large cities in 3D, since several application domains take advantage of them [BSL*15]. In the entertainment industry, interactive virtual cities are becoming the norm, but they remain very expensive to produce at scale. In urban planning, 3D reconstructions provide necessary representation of both real-world data and planned town changes, but generating whole metropolis without pre-processing is still cumbersome due to scale [SOW*18]. In archaeology, city-wide reconstructions allow for better visualization, used to study past events and to present them to an audience for education and outreach. Furthermore, in emergency response, large-scaled models are employed to train emergency, policy and military personnel, so to also include planning evacuation routes for various catastrophes [LZ08].

In this paper, we focus on modeling cities from OpenStreetMap (OSM) data, which is the most widely available and complete open source dataset of cities' information. To be more precise, we seek to generate a 3D model of an urban area that includes buildings, streets, landscape elements, such as waterways, vegetation and farmlands. Such elements are described through their 2D geometry footprint, and sometimes by 3D information concerning building and roof heights. Rarely details about the terrain elevation [Wik21a] are available. However, particularities accessible from OpenStreetMap that are associated to buildings, such as the height, number of levels, color and roof shape are strictly related to the city to be reproduced. For instance, most places in Germany are provided with many details concerning the buildings, streets and natural areas, whereas various municipalities of Rome, but also the city of Tokyo, present missing data. This lack of information can be associated to either a wide area or particular city elements. Our interest is to provide an open source library, available on GitHub (https://github.com/sarettak/3DCityReconstructionFromOSM), that can produce reconstructions of urban areas directly from raw OpenStreetMap data and can scale to generate entire cities at once. However, minor preprocessing or data augmentation are considered only when essential information about a specific city element is not available. Such introduction is required to have a stable and robust system able to also reproduce all those elements characterized by lack of information.

Our application takes as input raw OSM data [Kon19] extracted directly through Overpass Turbo [Wik21d], which is a data mining tool for OpenStreetMap. Before developing our system, we tested the city reconstruction methods available in Blender and Houdini that take the same input data, but we found them neither robust nor scalable. We noticed that Blender is not suitable to render a large portion of an area, for limitations associated to its OpenStreetMap module [Wik21b] in reproducing buildings that lack of information, which is the case for the vast majority of edifices in most cities (Figure 1a). The procedural building generator of Houdini [Est19] allows the user to create buildings with different appearance, by changing values of the height, number of floors and windows on each facade, but it requires the person to design each single construction separately in order to reach the goal of a wide 3D city model (Figure 1b). CityEngine is a 3D modeling software application that also exploits a procedural modeling approach to generate urban scenarios. It uses a set of hierarchical rules to create a virtual representation of a specific city, so users not familiar with rule files are required to invest substantial time in learning them. For this reason, we developed a simple C++ application that can generate a

DOI: 10.2312/stag.20211474

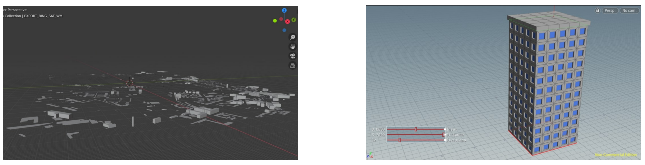

(a) *Urban area surrounding the Colosseum (Rome) in Blender.* **(b)** *Procedural building generator in Houdini.*

Figure 1: *City reconstruction in Blender and Houdini.*

3D model of the whole city in just hundreds of seconds, including serialization. Compared to the open source tools available to practitioners today, our system is significantly more scalable and robust on all cities we have tested.

In the reminder of the paper, we describe our application in details as follows. Section 2 contextualizes our study within the current state-of-the-art in 3D city reconstruction focusing on real data information. Section 3 illustrates the selected tool for information extraction and presents the content of the adopted file format. Section 4 reports the implemented features, specifically city elements added to the scene and discusses two different types of materials. Section 5 provides the performances along with the final representations of the reconstructed cities. In conclusion, Section 6 summarizes the key points of this paper, discusses its limitations and future research directions.

2. Related Works

Several studies have been conducted over the years with the aim of creating detailed city elements, such as buildings and streets, by employing different techniques and adopting various approaches. In this paper, we review the more relevant methods to whole-city generation and refer the reader to [LG06, WYD*14] for the synthesis of single buildings. The authors of [NGDA*16] develop a system that generates edifices starting from user sketches through machine learning techniques, that precisely recreate user's input. The underlying system is based on procedural grammars, that are often used for building synthesis [MWH*06, Kel21]. Moreover, further relevant works concerning reconstruction of single city elements are [CEW*08] and [VKW*12], focusing respectively on street modeling and parcels generation. To this aim, the first article addresses the problem of reproducing large street networks through the introduction of a modeling framework, where the user has the possibility to modify an existing street network or create one from scratch. Instead, the second paper presents a method for the interactive procedural generation of parcels, through a partition of the city blocks by considering user-specified subdivision attributes and style parameters.
In the last few years, procedural building generation [Kel07] through Neural Networks [BLS17] and Generative Adversarial Networks (GANs) [KGS*18] has been adopted to reproduce entire cities by changing the dimension and building style in order to obtain a more natural effect. Recently, authors of [BK20] propose a procedure for 3D city models generation from existing aerial photogrammetric datasets in order to capture city parts and terrain information. In developing this framework, CityGML and LoD2 have been adopted for reconstructing existing towns located in Sahinbey Municipality, in Turkey. Indeed, this study mainly focuses on object diversity and level of details in the recreated city models, simulating also a future representation of the interested area, but as the authors mention, such reconstruction is associated to only one Municipality. On the opposite, we would like to test scalability with our application, by analysing information of areas around the world differing in the city dimensions. Furthermore, another interesting work concerning automatic 3D building reconstruction starting from multi-view aerial images with a deep learning-based approach is presented in [YJLW21]. However, an important limitation emerges from this study. In particular, building segmentation becomes hard when constructions are located very close to each other, which is a common situation encountered in reconstructing cities. Moreover, we develop a system which is strictly based on city elements footprints and information provided in the GeoJSON files, allowing us to precisely recreate the desired place.

3. OpenStreetMap Data

The reconstruction of cities environments starts with the real world data available and with the choice of the tool to extract it.

3.1. Data Extraction

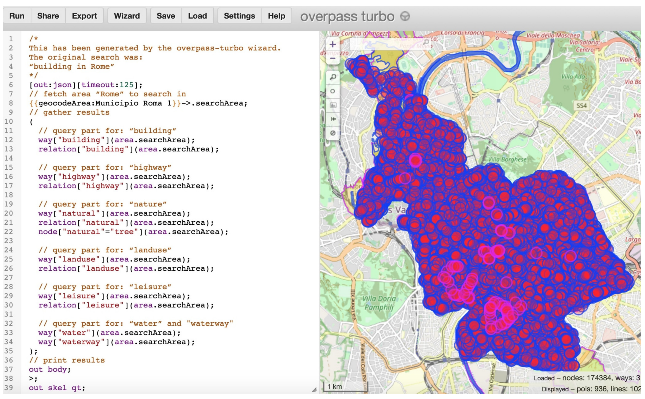
We evaluated a large variety of tools accessible for this step, differing in how they handle the spatial extent of the query, what are the returned information extracted and in what format they are available. For *OpenStreetMap*, we considered *BBBike Extract*, *Geofabrik* download server, and *Planet OSM*. The main issue we found is that these tools are targeted at reconstructing small places or only handle the proprietary *osm* format. For this reason, we focus on the reconstruction with direct queries to the OpenStreetMap database.

Overpass Turbo, one of the main APIs available for the main OpenStreetMap database, plays a key role in the development of our system. Its main feature relies on the ability of executing queries in a query language, with the aim of customizing the subset of information to extract, so to obtain a precise model of the desired city. Furthermore, Overpass Turbo supports people who have just started working with the query language with the *wizard*, whose task consists in converting simple, human readable search terms in Overpass queries, without additional user efforts.

3.2. Data Model

OpenStreetMap's data model is comprised of three main entities: nodes, ways and relations. A *node* defines a point on the Earth's surface identified by a pair of coordinates, representing respectively its latitude and longitude (e.g. a park bench or a tree). A *way* is an ordered list of nodes, constituting a river or a road, but can be also employed to delineate the area boundaries, mainly for buildings and forests. A *relation* is an ordered list that encodes the relationship between two or more data elements of nodes, ways, or other relations (e.g. a building with an internal garden).
OpenStreetMap employs tags to associate additional data for each of its entities. A *tag* is a key-value pair (a list of all the features available in OpenStreetMap is provided in [Wik21c]). The key,

which is unique, is employed for a topic, category or type of feature specification, while a more precise information is provided by the value. For instance, *highway = residential* defines the highway as a road, whose main function is to give access to people's homes. Hence, the map highlighting all the data requested through the query is displayed on the right side of the interface, such as in Figure 2 and the information can be downloaded in one of the available formats, including GeoJSON.


Run
Share
Export
Wizard
Save
Load
Settings
Help
overpass turbo
1 km
Loaded – nodes: 174384, ways: 3
Displayed – pois: 936, lines: 102

```
/*
This has been generated by the overpass-turbo wizard.
The original search was:
“building in Rome”
*/
[out:json][timeout:125];
// fetch area “Rome” to search in
{{geocodeArea:Municipio Roma 1}}->.searchArea;
// gather results
(
  // query part for: “building”
  way["building"](area.searchArea);
  relation["building"](area.searchArea);

  // query part for: “highway”
  way["highway"](area.searchArea);
  relation["highway"](area.searchArea);

  // query part for: “nature”
  way["natural"](area.searchArea);
  relation["natural"](area.searchArea);
  node["natural"="tree"](area.searchArea);

  // query part for: “landuse”
  way["landuse"](area.searchArea);
  relation["landuse"](area.searchArea);

  // query part for: “leisure”
  way["leisure"](area.searchArea);
  relation["leisure"](area.searchArea);

  // query part for: “water” and "waterway"
  way["water"](area.searchArea);
  way["waterway"](area.searchArea);
);
// print results
out body;
>;
out skel qt;
```

Figure 2: *The query for data extraction of the First Municipality of Rome (Italy) is presented on the left, while on the right, the associated visual representation is displayed in Overpass Turbo.*

3.3. GeoJSON

In our system, information concerning the city to be recreated in 3D is extracted from *GeoJSON* files. The choice of adopting such document format, rather than the proprietary *osm* one, is related to the clarity in the content organization and simplicity in its understanding, so to allow a wide range of users to take advantage of our application. GeoJSON is a geospatial data format, based on JSON extension and designed for representing simple geographical features, their properties and spatial attributes. In particular, a *Geometry*, identifying a region of space, a *Feature*, specifying a bounded entity or a *FeatureCollection* can be represented by a GeoJSON object. A FeatureCollection is a list of Features, each of which is composed of a set of additional properties and geometry information, as presented in Figure 3. Indeed, properties are essential when working with 3D city reconstruction, giving precise information about the type of the city element (if representing a building, a highway or a natural area), the color, the shape, the number of floors and the height (assigned to a building or a roof). Concerning the geometry, the following types are supported: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. Consequently, depending on the geometry type, a set of coordinates, identifying the positions of the vertices of the Feature shape, is also provided. Specifically, the longitude (X axis) and the latitude (Y axis) are required to assign a position to each vertex of the GeoJSON object. Rarely, a third coordinate, representing the elevation (Z axis), is available.

```
"features": [
  {
    "type": "Feature",
    "properties": {
      "@id": "way/23071179",
      "addr:city": "Dresden",
      "addr:housenumber": "14",
      "addr:postcode": "01067",
      "addr:street": "Dr.-Külz-Ring",
      "building": "roof",
      "building:levels": "1",
      "height": "2.9",
      "roof:shape": "flat"
    },
    "geometry": {
      "type": "Polygon",
      "coordinates": [
        [
          [
            13.7372807,
            51.0472358
          ],
```

Figure 3: *Properties information and geometry structure of a building in Dresda (Germany).*

4. City Generation

This Section is devoted to the explanation of the programming choices made in our city reconstruction application.

4.1. Base Libraries

We base our system on the *Yocto/GL* library [PNC19]. Yocto/GL is a software library for Computer Graphics written in C++, that provides support for several operations of basic math, geometry, path tracing, image and file management. Compared to other open source graphics libraries, such as PBRT [PJH16], CGAL [The21] and Libigl [JP17], Yocto/GL scales very well to large complexity environments due to its data-driven minimalistic design that improves both computation times and memory consumption for large scenes. All the aforementioned features induced us to develop our application for 3D city reconstruction relying on Yocto/GL. As additional dependencies, we used *Eartcut* [Mop20] for triangulation and JSON [Loh21] for parsing tags from GeoJSON files.

4.2. Coordinate Processing

Coordinates in GeoJSON are expressed with respect to a latitude-longitude parametrization. We instead want to map these coordinates to a 3D reference frame, centered in the middle of the city bounding box to avoid loosing too much precision. The transformation from the GeoJSON coordinates to the Cartesian one is computed as in [Wik21e]:

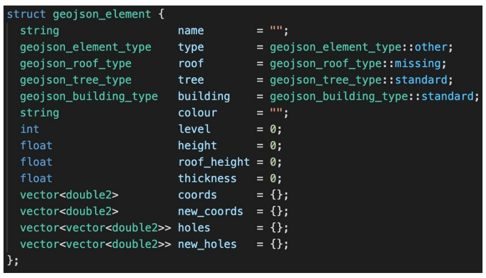
$$x = \left\lfloor \frac{256}{2\pi} 2^{zoom}(\lambda + \pi) \right\rfloor \; pixels$$

$$y = \left\lfloor \frac{256}{2\pi} 2^{zoom} \left(\pi - ln\left[tan\left(\frac{\pi}{4} + \frac{\varphi}{2}\right)\right]\right) \right\rfloor \; pixels$$

Initially, an instance of the city structure, that will contain all the required information extracted from the GeoJSON files, is initialized with initial values as shown in Figure 4. Then, each element is processed independently based on its type.

4.3. Buildings

The geometry of buildings is either a Polygon or MultiPolygon, with the only difference in the extra level of depth of the coordinates in the last case and tags that specify additional information.



```
struct geojson_element {
  string                      name        = "";
  geojson_element_type        type        = geojson_element_type::other;
  geojson_roof_type           roof        = geojson_roof_type::missing;
  geojson_tree_type           tree        = geojson_tree_type::standard;
  geojson_building_type       building    = geojson_building_type::standard;
  string                      colour      = "";
  int                         level       = 0;
  float                       height      = 0;
  float                       roof_height = 0;
  float                       thickness   = 0;
  vector<double2>             coords      = {};
  vector<double2>             new_coords  = {};
  vector<vector<double2>>     holes       = {};
  vector<vector<double2>>     new_holes   = {};
};
```

Figure 4: *Example of an element structure with initial values.*

Key	Value	Notes
"building"	"apartments", "hotel", "tower", "residential"	This key describes the typology of the building.
"building:levels"	*integer value*	This key specifies the number of levels in a building.
"building:height", "height"	*integer or float value*	This key describes the height of a building.
"building:colour"	"white", "yellow", "light yellow", "brown", "light brown" "light orange"	This key provides information about the color of a building.
"historic"	"monument", "building", "tomb", "bunker", "church"	This key identifies various historical places.
"roof:shape"	"flat", "onion", "pyramidal "gabled"	This key describes the shape of the roof.
"roof:height"	*integer or float value*	This key describes the height of a roof.
"tourism"	"attraction"	This key is used to map places of specific interest to tourists.

Table 1: *Most relevant building features analysed by our system.*

Various controls are performed in order to update all the necessary data, such as the shape and height of the roof, along with details concerning if the building is a historical or touristic place, the colour, the edifice height or number of levels, if any of these information is provided in the GeoJSON files. Table 1 illustrates the relevant building features available from OpenStreetMap treated by our system, with a brief explanation of what they represent. When some specifications are not available, they are computed using other attributes associated to the element in analysis, and if even these are absent, default values are assigned. For instance, if the number of levels of a building is missing from the properties list, but the height is given, we can easily determine the overall number of floors, starting from the standard height of a floor in real buildings, which is 3.2 meters in most locations. In the worst case, when neither information about the level nor the height is provided, but the Feature in consideration is a building of the following types: 'apartments', 'residential' or 'hotel', the number of floors is set to 3 as an average of the possible heights for such city elements. We have noticed that when default values are considered in the reconstruction (due to missing data), in the final scene, most buildings present the same height. Usually the lack of information is associated to a particular area, as demonstrated in Figure 6.
However, it could happen that a building has many details, while its neighbors are missing some essential information. Since each city element is analysed separately from the others, data propagation (e.g. details available for a building, such as the height, the color and number of floors are directly assigned to the nearest constructions that are missing them) is currently not supported by our system.
The building section is a sequence of one or more polygons, each of which is a list of coordinates. The first polygon represents the exterior walls, while all the others identify holes, i.e. interior walls, so there is the need to separate the positions of the external ring from the list of holes. To support all possible combinations, we triangulate the upper part of each building using the *Earcut* library. The edifice is then extruded by growing down its sides from the outer and inner polygons. Then, we assign colors to face depending on building tags and add texture for more realism if details are needed. To this aim, a specific control for generating the building sides is introduced. Indeed, the idea of a texture refinement lead us to generate a quad mesh rather than a triangular one, so to analyse the indices of two contiguous positions in order to find the correspondent vertices on the ground, as shown in Figure 5. Furthermore, a distinction of the surface in the upper part of the building (flat rooftop) from the rest is introduced, so to obtain two different shapes to whom we assign a specific colour and consider the upper surface as the base for extruding the roof. Additionally, a unique model would not be suitable to assign a specific material (either texture or colour) to the building and a different one to the roof.

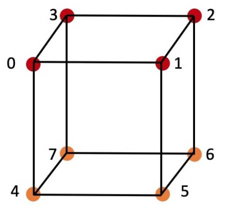


Figure 5: *Side generation of a building from 4 initial positions (in red).*

Roof information, and in particular roof shape, is rarely available in GeoJSON files and when present, usually its value indicates a flat rooftop. The other most common roof type is a gabled roof, that we support too. The remaining roof types are rarely adopted and mostly used non-consistently, so we ignore these tags and employ the gabled roof for all non-flat types. The chosen strategy consists in finding the 2D coordinates of the barycenter defined by the vertices of the rooftop, so to identify the highest position of the roof. Earcut library is responsible for triangulation, while the procedure

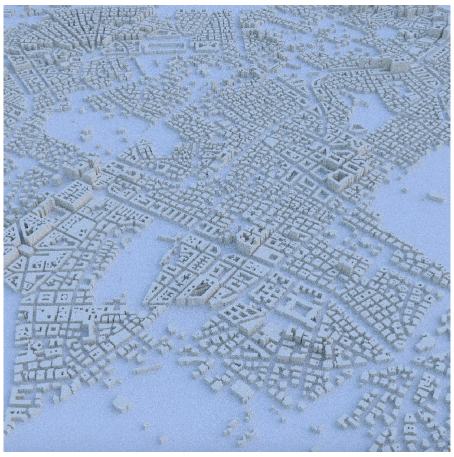
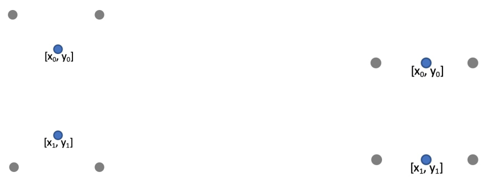


Figure 6: *Visualization of 3D buildings (without materials) of the Second Municipality of Rome (Italy).*

Key	Value	Notes
"highway"	"footway", "pedestrian", "steps", "path", "living_street", "trunk", "sidewalk", "primary", "secondary", "motorway", "tertiary"	This key is fundamental to describe roads and footpaths.

Table 2: *Most relevant street features analysed by our system.*



(a) *First type of polygonal street generation.* **(b)** *Second type of polygonal street generation.*

Figure 7: *Two different types of street representation.*

for the generation of roof sides is similar to the one presented previously for the buildings, with the only difference, in this case, in the presence of the triangular shapes. Furthermore, the total height is defined as the sum of the building and the roof heights.

Our building representation is simplistic and not detailed when viewed closely. However, we add particularities using textures, but do not include further geometry. We chose this reconstruction over the use of complex procedural buildings since it is scalable to large cities and more importantly, faithful to the OpenStreetMap data. Noticeably, we do not use instances, as typical in large environments rendering, but we generate a different shape for each building in the city.

4.4. Streets

Usually, streets in GeoJSON are represented as either 'LineString' or 'MultiLineString', both used for roads and bridges. Rarely a polygonal shape is adopted to such city elements, mainly to identify squares and large pedestrian areas. Regarding the information extracted, we distinguish two different types of streets: 'highway' and 'pedestrian'. The most frequent values associated to street features, analysed by our system, are presented in Table 2. Roads specified as polygons are simply triangulated, while streets represented as sequence of line segments, must be transformed into a polygonal surface. For this reason, the analysis of street points is performed by working on two contiguous positions at the same time (representing the current and next location). Indeed, starting from the thickness, the current and next coordinates, the most suitable surface is generated, as shown in Figure 7, in which points in blue identify the information available in the GeoJSON Feature, while the grey ones are obtained by considering a specific value for the thickness that is determined based on the street value: 0.00005 for 'pedestrian' type and 0.0001 for 'highway' specification. Therefore, the combination of separate squared surfaces positioned one next to the other generates a consistent shape of the street network, such as in Figure 8.

4.5. Natural Areas

This category encompasses the following natural elements: parks, forests, arable lands, rivers, lakes and beaches. The main features considered in our application are reported in Table 3. In general, the geometry representation of these city objects can vary, but they are mainly of type 'Polygon' or 'MultiPolygon'. Rarely some of them (e.g. rivers) are represented as 'LineString' or 'MultiLineString', so following the same analysis of the streets. In this last case, the thickness of the rivers, represented by either 'water' or 'waterways' value, is 1.0. Moreover, a precise distinction also concerns the green areas, in which we identify zones with a lot of vegetation as 'forest', while gardens and parks as 'grass', as presented in Figure 9.

Trees are the simplest city elements represented as 'Point' geometry type, consisting of a unique position. No vegetation models are present in the raw data, exception for the tree type that may be

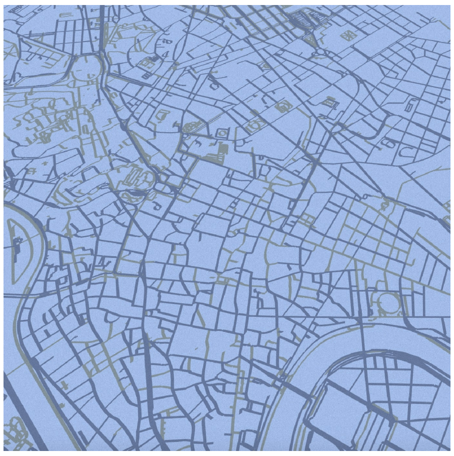


Figure 8: *Streets of the First Municipality of Rome (Italy).*

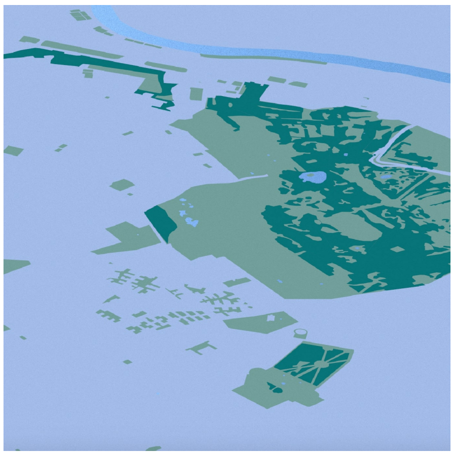


Figure 9: *Green areas of the Second Municipality of Rome (Italy).*

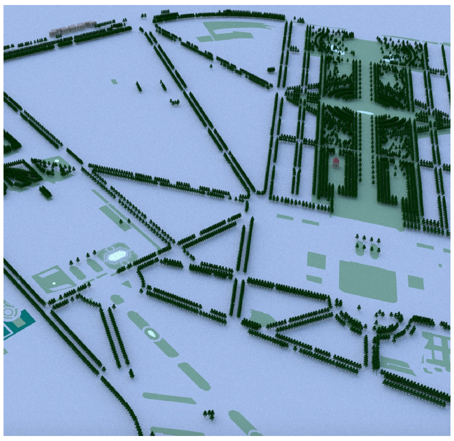
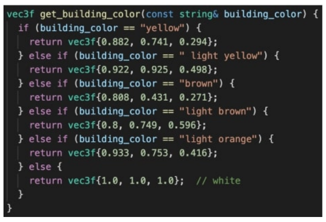


Figure 10: *Trees in the 7^{th} Arrondissement of Paris (France).*

Key	Value	Notes
"genus"	"Quercus", "Cupressus", "Pinus"	This key provides the scientific name of a tree.
"landuse"	"grass", "village_green", "farmyard", "meadow", "forest", "farmland", "orchard"	This key describes the purpose of a land area.
"leisure"	"park", "garden", "playground", "pitch", "recreation_ground", "dog_park"	This key identifies places where people go in their spare time.
"natural"	"scrub", "tree", "grassland", "water", "wood"	This key describes natural and physical land features, including the ones modified by humans.
"type"	"waterway", "pine", "palm", "cypress"	This key provides information about the feature typology in analysis.
"waterway"	"canal", "river"	This key specifies the type of the waterway.

Table 3: *Most relevant natural features analysed by our system.*



```
vec3f get_building_color(const string& building_color) {
  if (building_color == "yellow") {
    return vec3f{0.882, 0.741, 0.294};
  } else if (building_color == " light yellow") {
    return vec3f{0.922, 0.925, 0.498};
  } else if (building_color == "brown") {
    return vec3f{0.808, 0.431, 0.271};
  } else if (building_color == "light brown") {
    return vec3f{0.8, 0.749, 0.596};
  } else if (building_color == "light orange") {
    return vec3f{0.933, 0.753, 0.416};
  } else {
    return vec3f{1.0, 1.0, 1.0};  // white
  }
}
```

Figure 11: *Material mapping function from GeoJSON tags to 3D models.*

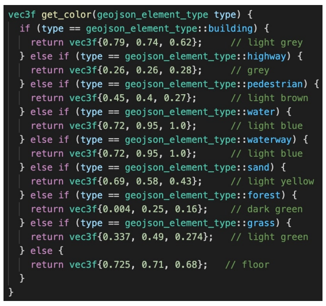
available. To this aim, we use instanced 3D meshes as proxy for trees, downloaded from Free 3D. Thus, the height of these city elements is adapted through the usage of the graphic software Blender by considering their type. The pine is the highest one, followed by the oak that is slightly smaller. The palm and the cypress have a similar value, while the standard tree is taller than the last two tree types. The standard model is adopted in all cases a tree exists in a GeoJSON file, but it is not of the aforementioned types. These models are converted in *ply* format and loaded in our system by exploiting Yocto/GL functions. An example of a scene with trees information is shown in Figure 10.

4.6. Materials

We assign materials to city objects based on their element type and the associated tag data. Indeed, we use either solid color or textures.

Colors and material types (i.e. matte, glossy, transmissive) are mainly employed to distinguish the type of streets, the green areas, but also to obtain a more natural result for the water, and to indicate historical buildings. For this reason, we define a material mapping function from GeoJSON tags to 3D scene elements that can be customized with ease. Such function converts the most common colors employed for historical buildings into their corresponding RGB values, as shown in Figure 11. Furthermore, function in Figure 12 assigns a specific color to each city object by relying on the element type in analysis. Additionally, users can customize the final appearance (in terms of materials) of the reconstructed area, by introducing in the system their own textures and by changing the colors assigned to different city elements. Indeed, from this point of view, our application is easy to use, developed with the aim of facilitating its improvement by the introduction of updates and changes. Our goal is to provide a simple, robust and customizable system for users to generate visualizations, since material information is mostly not present in the raw data.

Textures are used to obtain a more pleasant look of the reconstructed 3D city. Indeed, facades textures are introduced in the application to add particularities to buildings, such as windows and balconies, by analysing the number of floors. Texture coordinates (u,v) are generated accordingly to map facade repetitions. Hence, depending on the building levels, (u,v) coordinates scale consequently to adapt the texture to the height of the building facade.



```
vec3f get_color(geojson_element_type type) {
  if (type == geojson_element_type::building) {
    return vec3f{0.79, 0.74, 0.62};      // light grey
  } else if (type == geojson_element_type::highway) {
    return vec3f{0.26, 0.26, 0.28};      // grey
  } else if (type == geojson_element_type::pedestrian) {
    return vec3f{0.45, 0.4, 0.27};       // light brown
  } else if (type == geojson_element_type::water) {
    return vec3f{0.72, 0.95, 1.0};       // light blue
  } else if (type == geojson_element_type::waterway) {
    return vec3f{0.72, 0.95, 1.0};       // light blue
  } else if (type == geojson_element_type::sand) {
    return vec3f{0.69, 0.58, 0.43};      // light yellow
  } else if (type == geojson_element_type::forest) {
    return vec3f{0.004, 0.25, 0.16};     // dark green
  } else if (type == geojson_element_type::grass) {
    return vec3f{0.337, 0.49, 0.274};    // light green
  } else {
    return vec3f{0.725, 0.71, 0.68};    // floor
  }
}
```

Figure 12: *Function mapping the specific color to the element in analysis.*

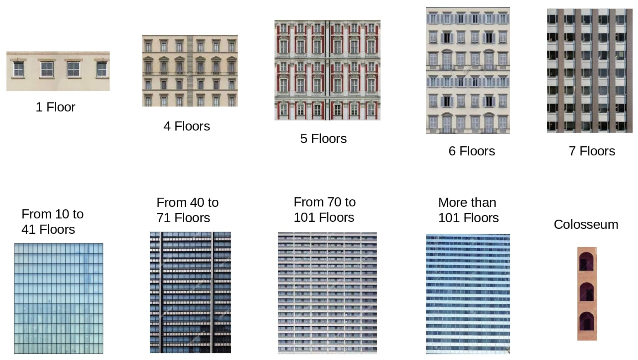


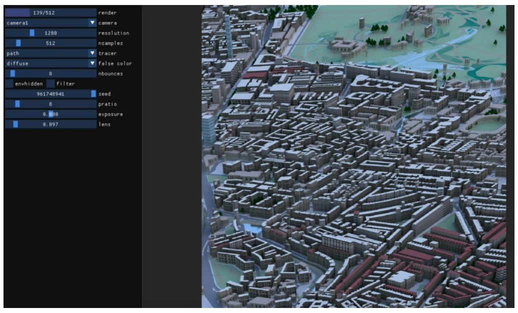
Figure 13: *Textures for building facades adopted for all the reconstructed cities.*

Figure 13 shows examples of the textures used in our system. Finally, we render our scene with path tracing and environmental illumination. Skies are either procedurally generated or use a captured HDRI map. Example of environmental maps are shown in Figure 14.

4.7. Graphical User Interface

A Graphical User Interface (GUI) is provided in our system for 3D city reconstruction, allowing users to interact directly with the scene. People can change either the perspective of the city, displayed on the right of the GUI, or values associated to the parameters (positioned on the left of the GUI) through sliders. Indeed, such changes are related to the resolution, the exposure (representing the amount of light reaching the camera) and the number of samples (the noise appearing when rendering the scene). However, compared to Yocto/GL GUI, we add the *lens* parameter, so to easily manage the quantity of data to display and to observe, from a closer point of view, the features of the reconstructed city elements. The GUI of our application is illustrated in Figure 15.

Figure 14: *Respectively texture of the morning, sunset and night.*



```
139/512        render
camera1        camera
1280           resolution
512            nsamples
path           tracer
diffuse        false color
8              nbounces
envhidden  filter
961748941      seed
8              pratio
0.000          exposure
0.097          lens
```

Figure 15: *GUI with the render of London.*

Model	GeoJSON	Size	Notes
Amsterdam	36.3 MB	12 km^2	
London	55.2 MB	24 km^2	London and Westminster
Paris	80.0 MB	23 km^2	10 Arrondissement
Rome	48.2 MB	40 km^2	I and II Municipalities
Tokyo	35.1 MB	17 km^2	

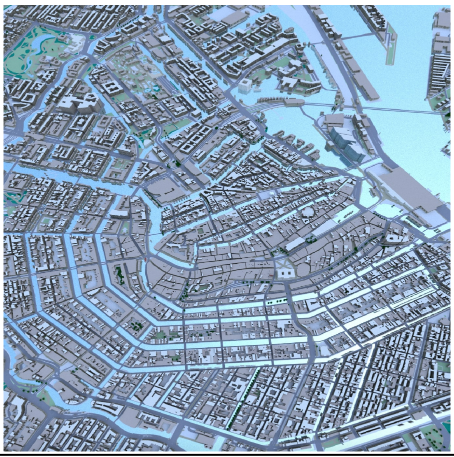
Table 4: *Cities GeoJSON data.*

5. Results

Starting from the available OpenStreetMap data, we demonstrate the efficiency of our system by analyzing the reconstruction performance of the following cities: Amsterdam, London, Paris, Rome and Tokyo. Indeed, Amsterdam is known for its canals, London for the alternation of ancient and modern buildings, while parks are emerging in Paris. Rome is characterised by its historical buildings and Tokyo is identified by the skyscrapers. The aforementioned cities are selected for their different structures, with the goal to compare the final results with the amount of information provided in the starting GeoJSON files. Hence, the dimension of such representative content has a key role in the performance analysis, considering that the time to generate each city element strictly depends on the amount of data to reconstruct. The dimensions of the selected urban areas are summarized in Table 4. For each of the aforementioned places, we discuss the scene of the final 3D city reconstruction. Below each resulting image, we provide also a table including all the identified city elements along with information concerning the generation time (in seconds), number of triangles, quads and instances.

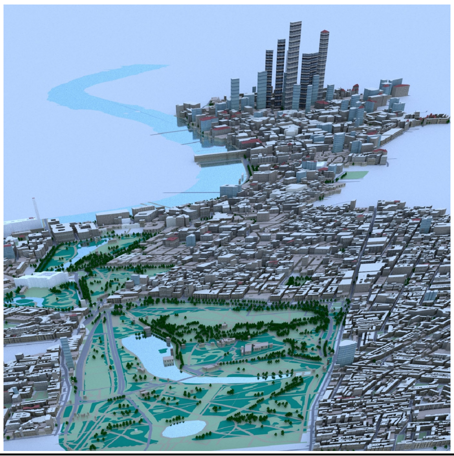
5.1. Cities

The scene of *Amsterdam* requires 1 minute and 27 seconds to be generated and saved, 36 seconds to be loaded and 1 second to be displayed. The resulting image in Figure 16 is identified by the characteristic canals of Amsterdam, accentuated by the sun reflecting in the waterways. For this render, a daily environmental texture is preferred, allowing the viewer to immediately focus on the water properties, that give more natural effect to the final scene.



	Time	Triangles	Quads	Elements
Buildings	0.19	170306	240305	23333
Green Areas	0.006	8747	0	1015
Highway	0.04	31674	0	15834
Pedestrian	0.02	19757	0	8222
Water/Waterway	0.007	11052	0	1562

Figure 16: *Reconstruction of Amsterdam.*



	Time	Triangles	Quads	Elements
Buildings	0.34	179991	271445	30474
Green Areas	0.006	15965	0	1104
Highway	0.08	60105	0	29937
Pedestrian	0.06	53337	0	22737
Water/Waterway	0.004	11564	0	153

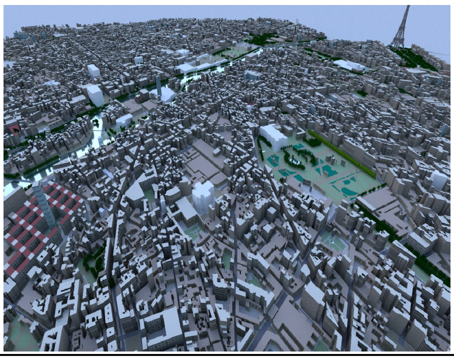
Figure 17: *Reconstruction of London.*

London is the second city reconstructed. In this case, the scene is generated and saved in 2 minutes and 33 seconds, 1 minute and 10 seconds is needed to load it, while 1 second for visualization. Figure 17 illustrates a representation with both skyscrapers, characterizing the modern part of the city, and low rise buildings in the foreground. The reconstructed parks and waterways are emerging, including all trees provided in the GeoJSON files.

The third city presented is *Paris*, requiring 2 minutes to be created and saved, 1 minute and 39 seconds to be loaded and 1 second to be displayed. The generated 3D scene presents an incredible amount of details, clearly deductible by the presence of many tree models. Indeed, Paris is one of the cities with the highest number of information provided in the GeoJSON files, as noticeable in Figure 18. Looking at this image, the viewer notices the reflecting buildings in the Seine river that flows obliquely through the scene and the Eiffel Tower in the background, that characterizes Paris.

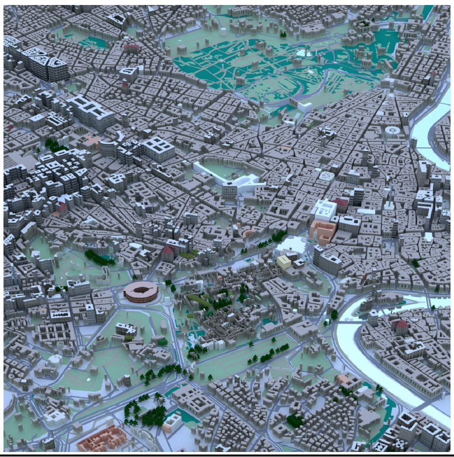
The render of the two municipalities of *Rome* requires 2 minutes and 24 seconds to be generated and saved, 1 minute and 11 seconds to be loaded, while 1 second to be presented. In the foreground of Figure 19, the Colosseum is emerging along with some historical buildings (monochrome) representing churches. Differently from other cities, the building heights are rarely provided in the GeoJSON files of Rome. For this reason, a standard height is adopted and many constructions appear to have the same elevation. However, different types of trees are distinguishable in the scene, mainly in the area surrounding the Colosseum, as pines, cypresses, palms and standard trees.

The last city presented is *Tokyo* that takes 2 minutes to be created, 25 seconds are necessary to load it and 1 second to display this area in 3D. The skyscrapers are the first city elements that capture the viewer attention. Indeed, they are positioned in the center of the rendered scene, characterized by different textures based on their number of floors that varies from 10 to more than 101. Particularly relevant in Figure 20 is the missing information of buildings in the lower part of the picture, where only streets and some green areas, that create the contours of the absent elements, are emerging.



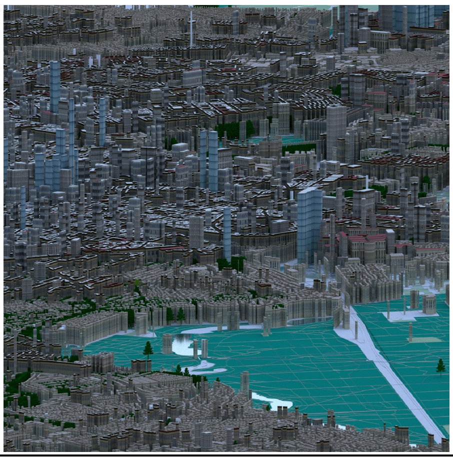
	Time	Triangles	Quads	Elements
Buildings	0.34	378726	485845	35695
Green Areas	0.01	30822	0	2400
Highway	0.057	43488	0	21393
Pedestrian	0.05	62890	0	18173
Water/Waterway	0.001	3446	0	252

Figure 18: *Reconstruction of Paris.*



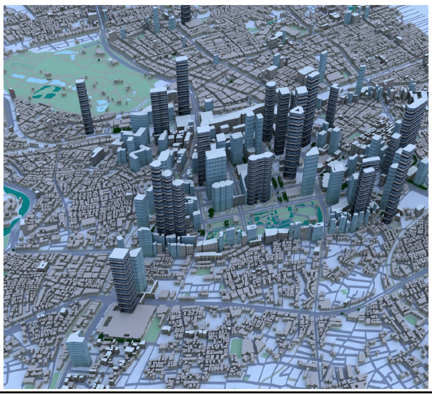
	Time	Triangles	Quads	Elements
Buildings	0.21	193215	266357	24347
Green Areas	0.01	34074	0	2784
Highway	0.11	78622	0	39263
Pedestrian	0.07	60321	0	25818
Water/Waterway	0.002	4678	0	202

Figure 19: *Reconstruction of Rome.*



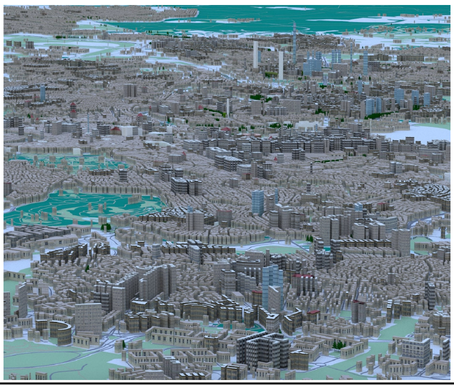
	Time	Triangles	Quads	Elements
Buildings	3.12	2122567	3156025	344489
Green Areas	0.17	298709	0	28287
Highway	0.94	597654	0	298218
Pedestrian	1.02	650984	0	315412
Water/Waterway	0.04	56330	0	9974

Figure 21: *Reconstruction of Berlin.*



	Time	Triangles	Quads	Elements
Buildings	0.32	103912	205800	33963
Green Areas	0.003	7248	0	545
Highway	0.058	42721	0	21353
Pedestrian	0.04	29800	0	13893
Water/Waterway	0.005	4409	0	1642

Figure 20: *Reconstruction of Tokyo.*



	Time	Triangles	Quads	Elements
Buildings	0.99	640253	1016889	125491
Green Areas	0.08	191374	0	12667
Highway	0.77	529212	0	264349
Pedestrian	0.25	186661	0	86931
Water/Waterway	0.005	14042	0	405

Figure 22: *Reconstruction of all available municipalities of Rome.*

5.2. Performance

Looking at the tables, it is possible to notice that the highest generation time is associated to the buildings. In particular, Paris (0.34 seconds) and London (0.34 seconds) are the cities requiring more time to generate edifices, followed by Tokyo (0.32 seconds), Rome (0.21 seconds) and finally by Amsterdam (0.19 seconds). However, the presented time performances are coherent with respect to the other city elements, because additional seconds are required for the generation of building sides. Moreover, we tested for scalability by reconstructing all the available municipalities of Berlin (11 in total) and Rome (11, including also the 1st and 2nd municipality presented previously). Hence, by analysing all the reconstructed cities, Berlin (illustrated in Figure 21) is the widest area generated, with the dimension for the provided data equal to 700 MB, followed by all municipalities of Rome (represented in Figure 22), with a total dimension of 205.9 MB.

6. Conclusion

Summarizing, we presented an application for urban area reconstruction that can handle processing of large city data in just hundreds of seconds. To this aim, we started with the available OpenStreetMap data and generated a 3D model suitable for scene visualization and rendering in a path tracer. Moreover, the efficiency and potential of the program is demonstrated by rendering cities differing in features, level of details provided in the GeoJSON files and dimension of the area to reproduce. Hence, compared to other open source tools, we are significantly faster and more robust at our target scale of whole-city reconstruction.

Looking at the future, we would like to introduce a terrain system for converting elevation data [Mas19, CMsRP12], more accurate roofs as in Straight Skeleton algorithm [EHP21], and we would also include embedded buildings found in the OpenStreetMap database. Furthermore, another interesting improvements concerns the introduction of small particularities, such as street lamps, traffic lights and street signs, relying on the information available on OpenStreetMap. Moreover, the GUI can be also optimized to allow users to customize the individual elements of the reconstructed city, such as materials (both textures and colors), number of floors and the height of the buildings with lack of information.

7. Acknowledgements

This work is partially supported by MIUR under grant *Dipartimenti di Eccellenza*.

References

[BK20] Buyukdemircioglu M., Kocaman S.: Reconstruction and efficient visualization of heterogeneous 3d city models. *Remote Sensing 12* (2020). 2

[BLS17] Biljecki F., Ledoux H., Stoter J.: Generating 3d city models without elevation data. *Computers Environment and Urban Systems 64* (07 2017), 1–18. 2

[BSL*15] Biljecki F., Stoter J., Ledoux H., Zlatanova S., Çöltekin A.: Applications of 3d city models: State of the art review. *ISPRS International Journal of Geo-Information 4*, 4 (2015), 2842–2889. 1

[CEW*08] Chen G., Esch G., Wonka P., Müller P., Zhang E.: Interactive procedural street modeling. *ACM Trans. Graph. 27* (08 2008). 2

[CMsRP12] Che Mat PhD R., shariff r., Rodzi A., Pradhan B.: *3D Terrain Visualization for GIS: A Comparison of Different Techniques*. 01 2012, pp. 265–277. 10

[EHP21] Eder G., Held M., Palfrader P.: Implementing straight skeletons with exact arithmetic: Challenges and experiences. *Computational Geometry 96* (2021), 101760. 10

[Est19] Estela M.: Building Generator, 2019. URL: https://www.sidefx.com/tutorials/building-generator/. 1

[JP17] Jacobson A., Panozzo D.: Libigl: Prototyping geometry processing research in c++. In *SIGGRAPH Asia 2017 Courses* (2017), Association for Computing Machinery. 3

[Kel07] Kelly G.: Citygen : An interactive system for procedural city generation. 2

[Kel21] Kelly T.: *CityEngine: An Introduction to Rule-Based Modeling*. Springer Singapore, 2021. 2

[KGS*18] Kelly T., Guerrero P., Steed A., Wonka P., Mitra N.: Frankengan: Guided detail synthesis for building mass models using style-synchonized gans. *ACM Transactions on Graphics 37* (12 2018). 2

[Kon19] Konieczny, M.: What is OpenStreetMap?, 2019. URL: https://mapsaregreat.com/What-is-OpenStreetMap.html. 1

[LG06] Larive M., Gaildrat V.: Wall grammar for building generation. pp. 429–437. 2

[Loh21] Lohmann N.: JSON for Modern C++, 2021. 3

[LZ08] Lee J., Zlatanova S.: *A 3D data model and topological analyses for emergency response in urban areas*. 01 2008, pp. pp.143–168. 1

[Mas19] Masri R.: Terrain model and gridding method variations for making topographic maps. *Journal of Physics: Conference Series 1280* (11 2019), 022029. 10

[Mop20] MopBox: Earcut, 2020. URL: https://github.com/mapbox/earcut.hpp. 3

[MWH*06] Müller P., Wonka P., Haegler S., Ulmer A., Van Gool L.: Procedural modeling of buildings. *ACM Trans. Graph. 25* (07 2006), 614–623. 2

[NGDA*16] Nishida G., Garcia-Dorado I., Aliaga D. G., Benes B., Bousseau A.: Interactive sketching of urban procedural models. *ACM Trans. Graph. 35*, 4 (2016). 2

[PJH16] Pharr M., Jakob W., Humphreys G.: *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., 2016. 3

[PNC19] Pellacini F., Nazzaro G., Carra E.: Yocto/GL: A Data-Oriented Library For Physically-Based Graphics. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2019), The Eurographics Association. 3

[SOW*18] Saran S., Oberai K., Wate P., Konde A., Dutta A., Kumar K., Kumar A. S. S.: Utilities of virtual 3d city models based on citygml: Various use cases. *Journal of the Indian Society of Remote Sensing 46* (2018), 957–972. 1

[The21] The CGAL Project: *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021. 3

[VKW*12] Vanegas C., Kelly T., Weber B., Halatsch J., Aliaga D., Müller P.: Procedural generation of parcels in urban modeling. *Computer Graphics Forum 31* (05 2012), 681–690. 2

[Wik21a] Wikipedia: 3D city model, 2021. URL: https://en.wikipedia.org/wiki/3D_city_model. 1

[Wik21b] Wikipedia: Blender, 2021. URL: https://wiki.openstreetmap.org/wiki/Blender. 1

[Wik21c] Wikipedia: Map features, 2021. URL: https://wiki.openstreetmap.org/wiki/Map_features. 2

[Wik21d] Wikipedia: Overpass turbo - openstreetmap wiki, 2021. URL: https://wiki.openstreetmap.org/wiki/Overpass_turbo. 1

[Wik21e] Wikipedia: Web Mercator projection, 2021. URL: https://en.wikipedia.org/wiki/Web_Mercator_projection. 3

[WYD*14] Wu F., Yan D.-M., Dong W., Zhang X., Wonka P.: Inverse procedural modeling of facade layouts. *ACM Trans. Graph. 33*, 4 (2014). 2

[YJLW21] Yu D., Ji S., Liu J., Wei S.: Automatic 3d building reconstruction from multi-view aerial images with deep learning. *ISPRS Journal of Photogrammetry and Remote Sensing 171* (2021), 155–170. 2