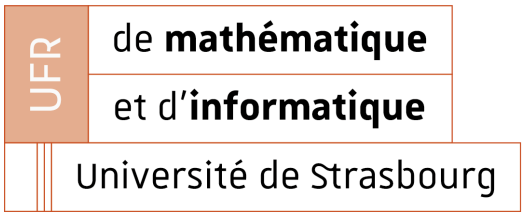


ExaMA WP1 - Vegetation

Giulio Carpi Lapi, Pierre-Antoine Senger

supervised by

Pierre Alliez and Vincent Chabannes

Date: June 3, 2024

Contents

1 Introduction

This project is part of the HiDALGO2[1] initiative, which "aims to explore synergies between modeling, data acquisition, simulation, data analysis and visualisation along with achieving better scalability on current and future HPC and AI infrastructures to deliver highly-scalable solutions that can effectively utilise pre-exascale systems."[2]

Specifically focusing on the Urban Building Model[3] Use Case (UBM) which is "developping the Urban Building pilot application to improve building energy efficiency and indoor air quality"[3], this project aims to integrate vegetation, particularly trees, into 3D models of urban environments.

The project was conducted within Cemosis[4] (Center for Modeling and Simulation in Strasbourg), which is hosted by IRMA[5] (Institute for Advanced Mathematical Research) at Strasbourg University. We operated as students under the supervision of Pierre Alliez[6], senior researcher and team leader at Inria[7] (National Institute for Research in Digital Science and Technology) Sophia Antipolis, and Vincent Chabannes[8], a research engineer at IRMA.

1.1 Context

Urban areas are complex ecosystems influenced by various factors, with vegetation, especially trees, playing a crucial role in shaping microclimates, reducing energy consumption, and enhancing overall livability[9].

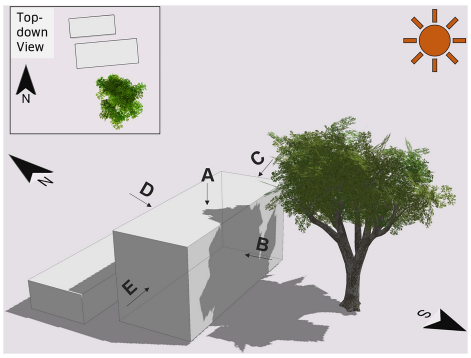


Figure 1: Tree providing shade to a building [10]

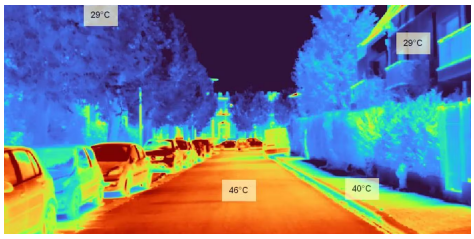


Figure 2: Thermal image of a street depicting heat distribution [11]

This project aims to integrate trees into 3D geometric models of urban environments to improve the accuracy and realism of thermal and energy simulations.

By leveraging data from `OpenStreetMap`[12], a collaborative free geographic database, we will use `CGAL`[13], an open source software library of computational geometry algorithms, to generate 3D tree models and integrate them into terrain meshes.

Our primary focus will be on Strasbourg, France. More specifically, we were provided with an `.stl`[14] file containing a 3D model of the Strasbourg city center:

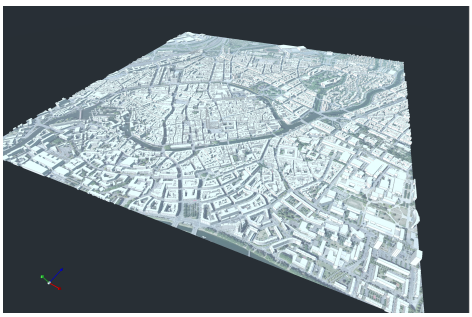


Figure 3: Strasbourg 3D model (1)

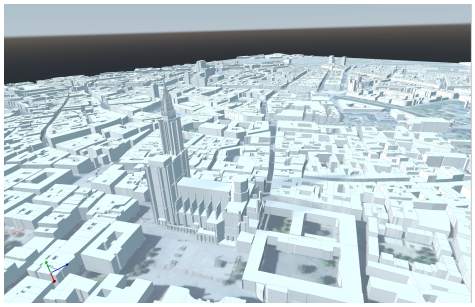


Figure 4: Strasbourg 3D model (2)

However, the software is designed to be easily adaptable to any area. Here's an example of a 3D model of Manhattan, NYC:

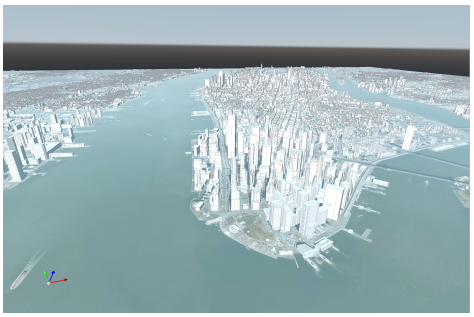


Figure 5: Manhattan 3D model (1)

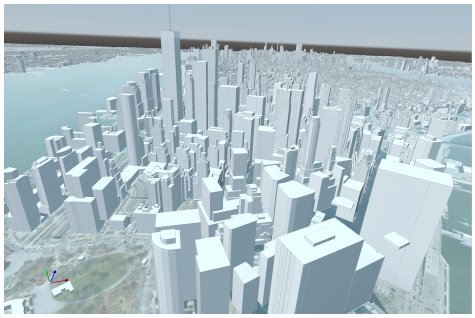


Figure 6: Manhattan 3D model (2)

1.2 Main objectives

- Extracting tree data from `OpenStreetMap`.
- Generating 3D tree models using `CGAL`.
- Integrating tree models into the terrain mesh we already have.
- Optimizing computational efficiency.
- Delivering versions `V0`, `V1`, and `V2` by specified deadlines.

1.3 Software and libraries

To source our data, we'll utilize the `Overpass API`[15] a read-only API to query data from `OpenStreetMap`, alongside `cURL`[16], a URL transfer library. For geometric modeling, we will utilize the master `CGAL` library, available on GitHub[17], known for its efficiency and reliability in geometric computation.

1.4 GitHub repository

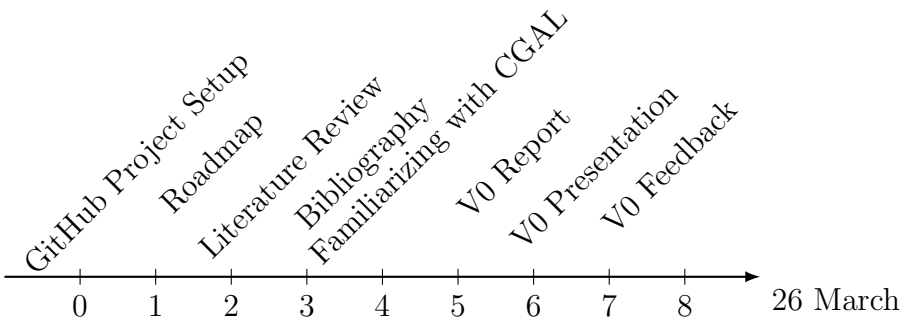
We created a GitHub repository to manage the project and facilitate collaboration. The repository contains the project's code, documentation, and resources. It will be updated regularly to reflect the progress and changes made during the project's development.

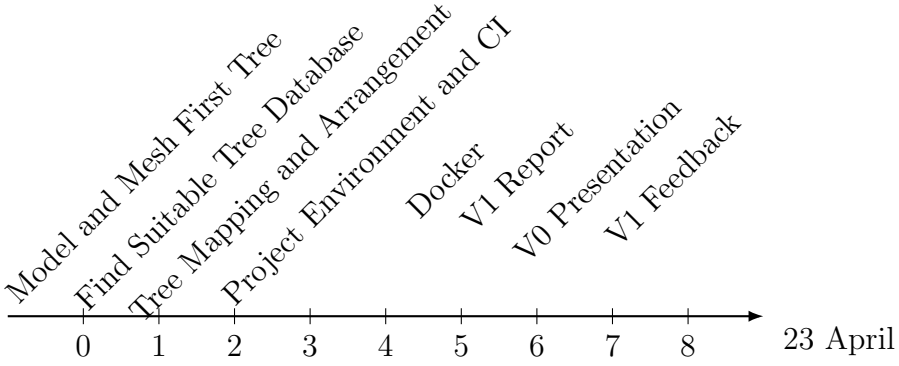
1.5 Roadmap

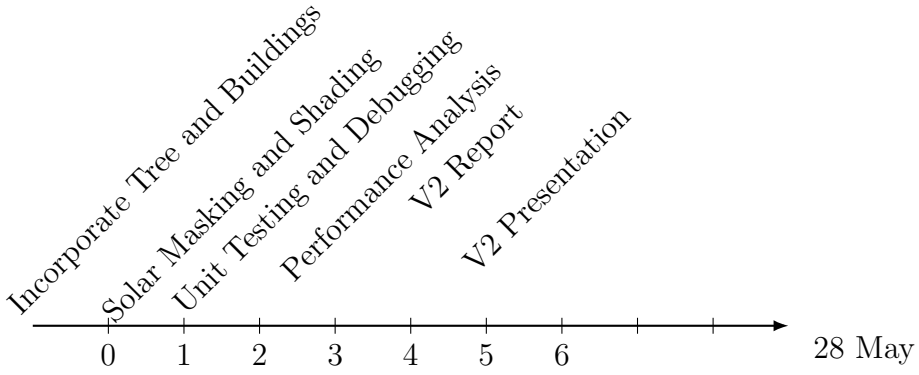
We created a roadmap on GitHub Projects to ensure we meet the deadlines for the deliverables. The deadlines are as follows:

- `V0` - 26 March
- `V1` - 23 April
- `V2` - 28 May

Here is a brief overview of the main issues:







We have adopted the Agile methodology to enhance flexibility and responsiveness throughout the project.

2 Methodology

2.1 Data acquisition

We will use the `Overpass API` and `cURL` to query `OpenStreetMap` for all the available tree data within the specified bounding box:

```
curl_easy_setopt(curl, CURLOPT_URL,
                 "http://overpass-api.de/api/interpreter");

// Set the Overpass query with the bounding box
std::string query =
    "[out:json]; (node(" + bbox + ")[\"natural\"=\"tree\"];);
    out;";

    std::cout << "Query: " << query << std::endl;
```

The `http://overpass-api.de/api/interpreter` endpoint interprets and executes the `Overpass QL`[18] queries, retrieving specific parts of the `OpenStreetMap` data. In this case, the query fetches all nodes within the given bounding box tagged as `natural=tree`.

A *config.json* file is available for the user to specify the area of interest and other parameters:

```
{
    "bbox": "48.5750,7.7394,48.5919,7.7621",
    "origin": "48.583055227464364, 7.748664426560083",
    "altitude": 0,
    "LOD": 3,
    "default_height_range": "3, 6",
    "default_genus": "Platanus",
    "input_building_mesh": "mesh_lod1.stl",
    "merge_buildings_trees": true,
    "output_name": "grande_ile"
}
```

Where :

- `bbox`: is the bounding box for the query in the format:

(SW latitude, SW longitude, NE latitude, NE longitude)

- `origin`: is the origin of the 3D space in latitude and longitude used to convert the GPS coordinates to Cartesian coordinates. It must be the same for the terrain mesh and the tree meshes.
- `altitude`: is the z coordinate of the tree base.
- `LOD`: is the level of details of the meshes (0, 1, 2 or 3)
- `default_height_range`: is a range used to randomly assign a height to trees that do not have one.
- `default_genus`: is the genus assigned to trees that lack a predefined genus.
- `input_building_mesh`: is the name of the input file representing the terrain mesh.
- `merge_buildings_trees`: is a boolean indicating whether the buildings and trees should be merged into a single mesh or not.
- `output_name`: is the base name of the output file representing the unions of the tree meshes.

The data will be stored in a *.json* file.
Here is an example of the query result for one tree:

```
{
    "type": "node",
    "id": 10162018740,
    "lat": 48.5850910,
    "lon": 7.7502624,
    "tags": {
        "circumference": "1.47655",
        "diameter_crown": "5",
        "genus": "Platanus",
        "height": "6",
        "leaf_cycle": "deciduous",
        "leaf_type": "broadleaved",
        "natural": "tree",
        "ref": "16401",
        "source": "data.strasbourg.eu - patrimoine_arbore",
```

```
        "source:date": "2022-01-02",
        "species": "Platanus acerifolia x",
        "species:wikidata": "Q24853030"
    }
}
```

Sometimes, multiple `tags` are missing, as shown here:

```
{
    "type": "node",
    "id": 4439566691,
    "lat": 48.5839128,
    "lon": 7.7487125,
    "tags": {
      "natural": "tree"
    }
}
```

We will primarily use the `position` of the tree (latitude and longitude), its `height`, and the `genus` (since this data is more abundant than the species) to generate the 3D tree models.

2.2 Tree library

We created a library of 13 tree models in `.stl` format sourced from the publicly accessible `SketchUp`[19] 3D Warehouse.

Here is an example of a Ginkgo model that we acquired from `SketchUp`:

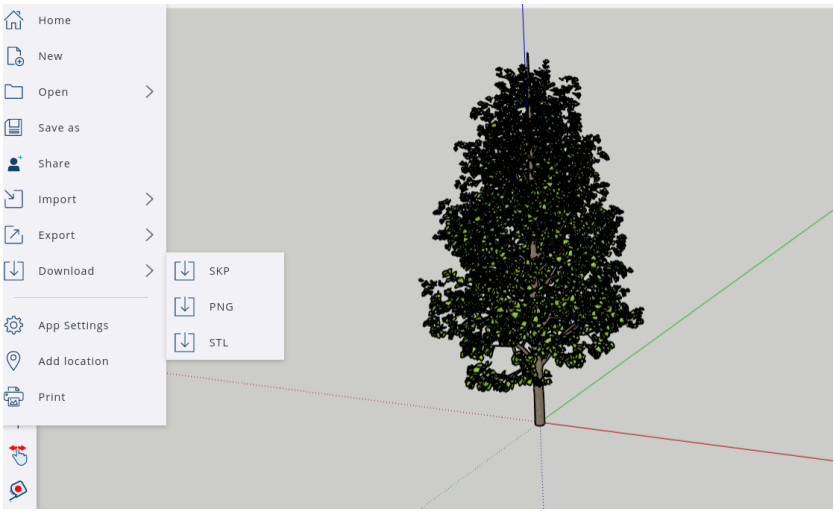


Figure 7: Mesh of a Ginkgo tree on Sketchup 3D Warehouse

Below, some of these models visualized using MeshLab[20]:

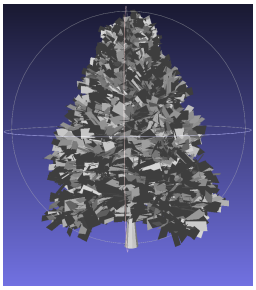


Figure 8: Abies

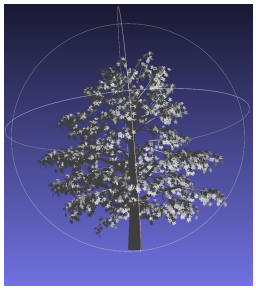


Figure 9: Acer

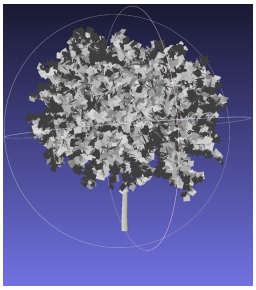


Figure 10: Aesculus

Figure 11: Catalpa

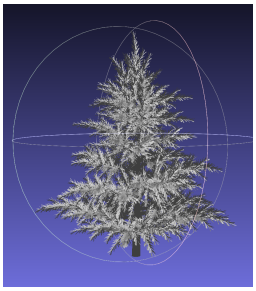


Figure 12: Cedrus

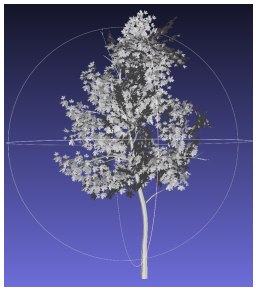


Figure 13: Liquidanbar

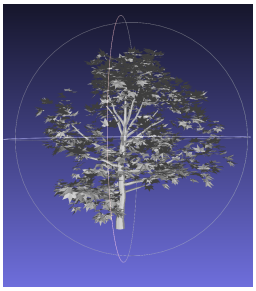


Figure 14: Platanus

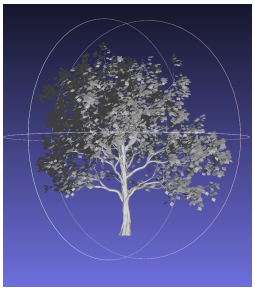


Figure 15: Quercus

To ensure the program can handle trees not included in our library, we categorized an additional 28 genera by matching them to the most similar models among the 13 trees we have.

```
{
    "known_genus": ["Abies",
                    "Acer",
                    "Aesculus",
                     ... ],
    "cedrus_like": [ "Chaemacyparis",
                    "Cupressus",
                     ... ],
    "acer_like": ["Fadus",
                "Metasequoia",
                "Sequoiadendron",
                ... ],
    "liquidambar_like": ["Liriodendron",
                         "Pyrus",
                         "Alnus",
                         ... ],
    "quercus_like": ["Corylus",
                    "Carya",
                    "Fagus",
                    ... ]
}
```

- `known_genus` is a list of the genus for which we have a mesh model.
- `cedrus_like` is a list of the genus for which the cedrus mesh model will be used.
- etc.

2.3 Dataset

The primary dataset we are working with is a 3D model of the Strasbourg city center, provided in the `.stl` file format. This file contains the geometric information of the terrain and buildings in the specified area, which serves as the base for our project.

The `.stl` (stereolithography) format is widely used for 3D printing and computer-aided design (CAD). It represents the surface geometry of a 3D object without any color, texture, or other attributes. The file comprises a collection of triangular facets, each defined by its vertices and normal vector.

Here are some key details about the `.stl` file we are working with:

- **File Name:** mesh_lod1.stl
- **Region Covered:** Strasbourg city center
- **File Size:** 42,7 MB
- **Number of Vertices:** Approximately 120,959
- **Number of Facets:** Approximately 273,178

The 3D model includes various urban features such as buildings, streets, and other infrastructure. This detailed representation is crucial for accurately integrating tree models and conducting subsequent thermal and energy simulations.

Below are visual representations of the 3D model from different angles:

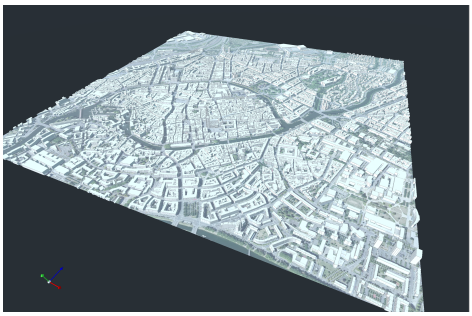


Figure 16: Strasbourg 3D model (1)

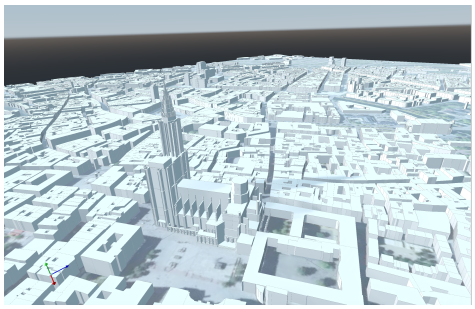


Figure 17: Strasbourg 3D model (2)

This dataset is instrumental in allowing us to accurately position tree models within the urban landscape, ensuring that our simulations reflect realistic interactions between vegetation and urban structures.

2.4 Alpha Wrapping

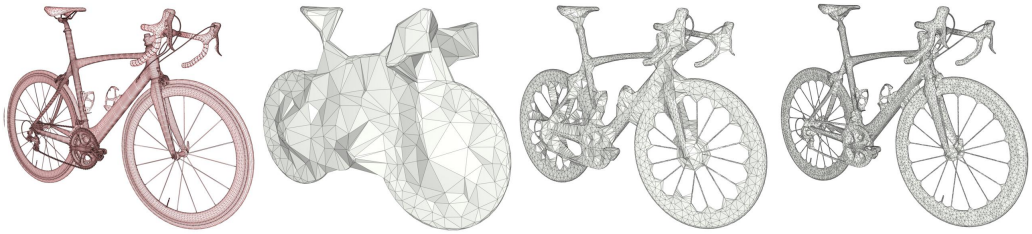


Figure 18: Different LOD of the Alpha Wrapping of a bike

2.4.1 Approach

To enclose a 3D model within a volume, various methods balance runtime and approximation quality. Simple methods, like bounding boxes, have large errors. Convex hulls improve quality but remain crude, especially for complex models. Alpha shapes, a special case of the Delaunay triangulation, offer piecewise-linear approximations but struggle with complex input data.

Inspired by alpha shapes, it uses shrink-wrapping, constructing a 3D Delaunay triangulation and iteratively removing eligible boundary tetrahedra. This process refines the triangulation, avoiding inner structures and unnecessary computations. This method supports flexible input formats (triangle soups, polygon soups, point clouds) and allows trading tightness for mesh complexity.

2.4.2 Algorithm Initialization

The algorithm starts by inserting the vertices of a bounding box into a 3D Delaunay triangulation. In CGAL's 3D Delaunay triangulation, boundary facets are adjacent to infinite cells tagged as outside, while finite tetrahedra are tagged inside.

2.4.3 Shrink-wrapping

The algorithm traverses from outside to inside cells, using a priority queue of Delaunay triangle facets (gates). A gate is alpha-traversable if its circumradius exceeds a user-defined alpha. The priority queue, initialized with convex hull gates, is sorted by decreasing circumradius. Traversal uses dual Voronoi edges.

When moving from an outside cell c_o to an inside cell c_i through an alpha-traversable facet f:

1. Check for intersections between f's dual Voronoi edge and the offset surface. Insert the first intersection point as a Steiner point into the triangulation.
2. If no intersection but c_i intersects the input, project c_i's circumcenter onto the offset surface and insert it as a Steiner point.

Newly alpha-traversable gates are added to the queue. If neither criterion is met, c_i is tagged outside, and its gates are pushed to the queue. The process terminates when the queue empties, producing a triangle surface mesh from the Delaunay triangulation.

The figure below depicts the steps of the algorithm in 2D:

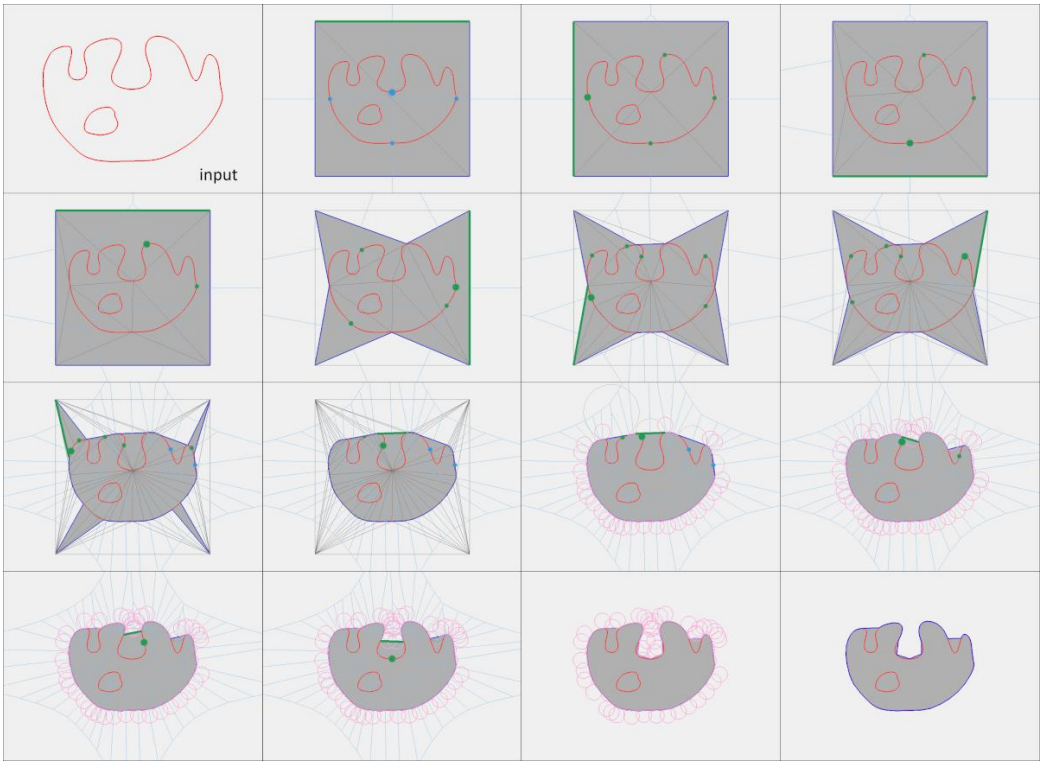


Figure 19: Steps of the shrink-wrapping algorithm in 2D. The algorithm initializes by inserting the bounding box corners into a Delaunay triangulation, tagging finite triangles inside. The current gate (green edge) is alpha-traversable. Adjacent triangles are tagged outside if they do not intersect the input. If they do, new Steiner points are inserted, and traversal resumes. The output edges (dark blue) separate inside from outside triangles.

2.4.4 Termination and Guarantees

The algorithm guarantees termination and produces a 2-manifold triangulated surface mesh that encloses the input data. By wrapping from outside to inside and refining the triangulation as needed, it ensures no intersecting cells are flagged inside. Steiner points inserted during refinement break necessary cells, reducing circumradii and ensuring completion.

2.5 Tree model generation

Using the CGAL 3D Alpha Wrapping algorithm, we will generate reference tree meshes for each level of detail (LOD) from 0 to 3. This pre-processing ensures that the meshes are readily available in memory, eliminating the need to wrap each tree model individually during program execution.

A *wrap.cpp* file will be provided to generate the reference meshes. It can be used as follows:

```
./build/wrap tree_ref/raw_tree/Ginkgo.stl 50 600
```

Where 50 is the `alpha` value and 600 is the `offset` value.

Additionally, to wrap all the trees in a directory, the *wrap_all.sh* script can be used:

```
#!/bin/bash

# Directory containing the raw STL files
INPUT_DIR="tree_ref/raw_tree"

# The alpha value to use for wrapping
ALPHA=100

# Loop through all STL files in the input directory
for input_file in $INPUT_DIR/*.stl; do
    ./build/wrap "$input_file" "$ALPHA"
done
```

The reference meshes will be used to generate the tree models obtained from the `Overpass` query. Each tree model will be generated by scaling and translating the reference mesh to match the tree's height and position in 3D space. This will be accomplished using the `CGAL Affine Transformation`[21], which has linear complexity in the number of vertices of the mesh.

The alpha parameter for the CGAL wrapper was set as follows for each LOD:

- LOD 0: 0.1
- LOD 1: 20
- LOD 2: 50
- LOD 3: 100

Here's a table showing the number of faces per tree type and LOD:

Tree	LOD 0	LOD 1	LOD 2	LOD 3
Abies	334	920	5648	35592
Acer	326	1516	10622	33606
Aesculus	274	1068	6084	28782
Catalpa	404	946	4508	19940
Cedrus	248	974	6702	27564
Ginkgo	332	1128	7488	38258
Gleditsia	228	1032	7506	28866
Liquidambar	212	858	5210	25146
Magnolia	230	892	7820	41150
Platanus	286	1150	7148	30670
Quercus	274	1008	8748	40426
Taxus	370	798	3644	15826
Tilia	328	1102	8104	46112

Table 1: Number of faces per tree type and LOD

The `offset` parameter for the CGAL wrapper was set as 600 for each LOD.

Here are the results for a Ginkgo tree model at each LOD:

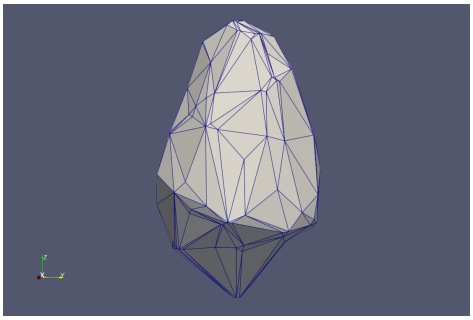


Figure 20: Ginkgo lod0

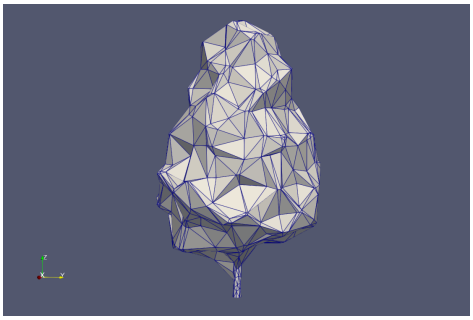


Figure 21: Ginkgo lod1

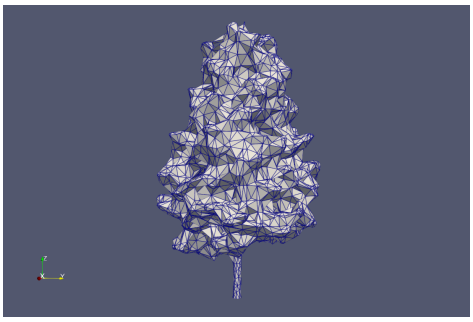


Figure 22: Ginkgo lod2

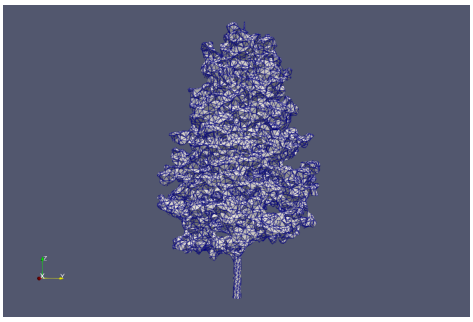


Figure 23: Ginkgo lod3

The scenario where insufficient data is available for a tree must also be addressed. Initially, we considered using a k-nearest neighbors algorithm[22] to determine the tree's metadata (species, height, leaf density, etc.) based on surrounding trees. However, this approach is impractical because data is often missing for large areas, such as parks. Instead, we will assign a random height selected from a normal distribution with a mean specified by `default_height_range` and a genus specified by `default_genus` in the *config.json* file.

2.6 Mercator projection

Generated tree models will be integrated into terrain meshes to create comprehensive 3D urban models. To ensure precise integration into the terrain

mesh (especially for large area), the tree models coordinates (latitude, longitude) will be converted to Cartesian coordinates (x, y) using a Mercator projection[23].

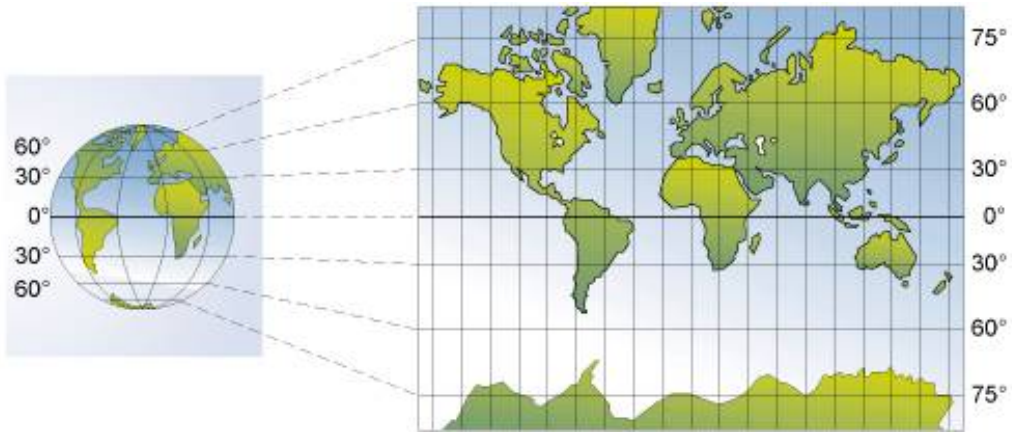


Figure 24: Mercator's projection[24]

This is the most common way to represent the Earth's surface on a plane and has the advantage of being conformal, meaning that it preserves angles locally (hence its usage in sailings).
In a Mercator projection, parallels and meridians are represented by straight orthogonal lines, with the equator being the horizontal line placed at the center of the map. The other parallels must necessarily be stretched (east-west stretching). This stretching is accompanied by a corresponding north-south stretching, so that the north-south scale is equal to the east-west scale everywhere.
Mathematically, the Mercator projection is defined as follows: if a point on the sphere has a latitude ϕ and longitude λ (with λ_0 placed at the center of the map), then its projection on the Mercator map will have coordinates

$$\begin{cases} x = \lambda - \lambda_0 \\ y = \ln(\tan(\frac{\pi}{4} + \frac{\phi}{2})) \end{cases} \tag{1}$$

To achieve this while taking into account that the Earth is not perfect sphere we will assume the Earth is a geodesic defined as `WGS84`[25] and use the `WGS84toCartesian`[26] open source header-only library to convert the coordinates.

Then the union of all the tree meshed will need to be computed to create a single mesh and avoid collision between the trees.
This will be achieved using the `corefine and compute`[27] function from the CGAL library. Another approach could be to use the convex hull of the tree meshes, using the intersection of the convex hulls to compute the union of the tree meshes.

2.7 Metrics

The complexity of the algorithms is a key metric to consider.
On each execution of the program, basics metrics will be exported to a text file in the `output` directory.
These results will be analyzed more thoroughly in the section 4.

Example of the result's metrics for *grande_ile_LOD1.txt*:

```
Area: 561545 meters
Total number of trees: 409
Number of tree which had no height: 67
Number of tree which had no genus: 27
Number of vertices: 241791
Number of faces: 482686
Time to mesh: 155.965 seconds, (2.59942 minutes)
```

3 Implementation

3.1 Config class

This class is designed to store the program's configuration. It will be used to store the `bounding box`, the `level of details`, the `output name`, the `default genus`, the `default height`, the `origin`, the `input building mesh`, the `altitude` and the `merge` boolean.

```
class Config {
    private:
        std::string M_bbox;
        int M_LOD;
        std::string M_output_name;
        std::string M_default_genus;
        std::string M_default_height;
        std::string M_origin;
        std::string M_input_building_mesh;
        double M_altitude;
        bool M_merge;

    public:
        Config(std::string const &filename);

        // Ommiting getters and setters

        std::vector<double> bbox_coords() const;
    };
```

3.2 Query function

The function `perform_query` will be used to query the `Overpass API` and save the result in a `.json` file. It will take the bounding box as a parameter. The function `get_query_result` will be used to get the result of the query using the header only library `nlohmann::json`.

```
void perform_query(std::string bbox);
nlohmann::json get_query_result();
```

3.3 Class tree

The `Tree` class will be used to store the tree's metadata and mesh.
The function `computeXY` will be used to convert the latitude and longitude to Cartesian coordinates using the `WGS84toCartesian` library.
The function `wrap` will be used to model the tree by scaling and moving the reference mesh to the correct position in the 3D space.
The function `load_genus` will be used to load tree genus categories from the `trees.json` file.
The function `createTreeFromJson` will be used to create a tree object from the `.json` data acquired from the `Overpass API`.
In order to be able to sort trees, the `operator<` function will be overloaded.

```
using K = CGAL::Exact_predicates_inexact_constructions_kernel
    ;
using Point_3 = K::Point_3;
using Mesh = CGAL::Surface_mesh<Point_3>;

class Tree {
    private:
    long M_id;
    double M_lat, M_lon, M_x, M_y;
    double M_height, M_altitude;
    double M_circumference, M_diameter_crown;
    std::string M_genus, M_species, M_season;
    std::vector<std::string> M_known_genus, M_cedrus_like,
    M_acer_like,
        M_liquidambar_like, M_quercus_like;
    Mesh M_wrap;
    std::vector<Point_3> M_points;
    std::vector<std::array<int, 3>> M_faces;

    public:
    // Ommiting getters and setters

    void computeXY(double ref_lat, double ref_lon);
    void wrap(int lod);
    void load_genus(const std::string &filename);
};

Tree createTreeFromJson(const nlohmann::json &treeJson);
std::ostream &operator<<(std::ostream &os, const Tree &tree);
bool operator<(const Tree &lhs, const Tree &rhs);
```

Each tree model has a CGAL Mesh wrapper object that will contain the tree's mesh and its position in the 3D space.

Scaling and moving the trees into the correct position ended being more complex than expected.

```
// Calculate centroid of the tree
double centroid_x = 0, centroid_y = 0, centroid_z = 0;
for (const Point_3 &p : points) {
    centroid_x += p.x();
    centroid_y += p.y();
    centroid_z += p.z();
}
centroid_x /= points.size();
centroid_y /= points.size();
centroid_z /= points.size();
Point_3 centroid(centroid_x, centroid_y, centroid_z);

// Calculate bounding box from points
for (const Point_3 &p : points)
    bbox += p.bbox();

scaling_factor_double = M_height / (bbox.zmax() - bbox.zmin()
    );

K::RT scaling_factor(scaling_factor_double); // Convert to
    exact type

// Find the base of the tree (minimum z-coordinate)
double base_z = std::numeric_limits<double>::max();
for (const auto &p : points) {
    if (p.z() < base_z)
        base_z = p.z();
}

// Create affine transformations
CGAL::Aff_transformation_3<K> translate_to_base(
    CGAL::TRANSLATION, Vector_3(-centroid.x(), -centroid.y(),
    -base_z));
CGAL::Aff_transformation_3<K> scale(CGAL::SCALING,
    scaling_factor);
CGAL::Aff_transformation_3<K> translate_back(
    CGAL::TRANSLATION, Vector_3(centroid.x(), centroid.y(),
    base_z));
CGAL::Aff_transformation_3<K> translate_to_target(CGAL::
```

```
    TRANSLATION,
                                                Vector_3(
    M_x, M_y, 0));

// Apply transformations: move to base, scale, move back,
    move to target
for (auto &p : points) {
    p = translate_to_base.transform(p);   // Move to base
    p = scale.transform(p);               // Scale
    p = translate_back.transform(p);      // Move back to
    original position
    p = translate_to_target.transform(p); // Move to target
    position
}
// Clear existing mesh data
M_wrap.clear();

// Add transformed vertices to the mesh and store their
    descriptors
std::map<Point_3, Mesh::Vertex_index> vertex_map;
for (const auto &p : points) {
    auto v = M_wrap.add_vertex(p);
    // Store the vertex descriptor for the transformed vertex
    vertex_map[p] = v;
}

// Add faces to the mesh
for (const auto &face : faces) {
    // Retrieve vertex descriptors for the face vertices
    Mesh::Vertex_index v0 = vertex_map[points[face[0]]];
    Mesh::Vertex_index v1 = vertex_map[points[face[1]]];
    Mesh::Vertex_index v2 = vertex_map[points[face[2]]];

    // Add the face to the mesh
    M_wrap.add_face(v0, v1, v2);
}
```

To ensure the placement was correct we first had to move the tree to the origin of its bounding box, scale it to the correct height, move it back to its original position (because scaling it was moving the tree around), and finally move it to the correct position in the 3D space.

Note: To improve tree placement we could also compute the center of

mass of a slice at the base of the tree and use it as its origin.

3.4 Doxygen documentation

`Doxygen`[28] documentation is available for the project. Doxygen is a documentation generator that produces comprehensive reference manuals from annotated source code.

To generate the documentation, you can use the following command:

```
doxygen Doxyfile
```

The documentation will be available in the `html` directory.
You can open it with the browser of your choice, for example with Firefox:

```
firefox html/index.html
```

4 Results

4.1 Model integration

Here is an example of the tree mesh union of Place de la République in Strasbourg using generic trees and LOD1:

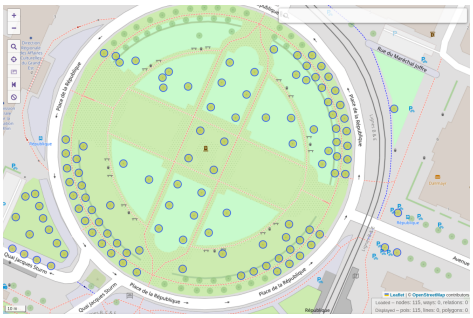


Figure 25: Overpass turbo query for Place de la République in Strasbourg

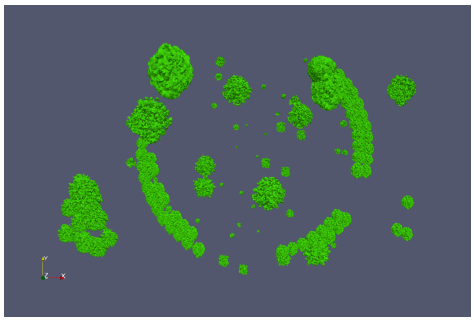


Figure 26: Mesh we generated

The idea is to integrate the tree models into the terrain mesh. Here is an example of what this integration could look like:

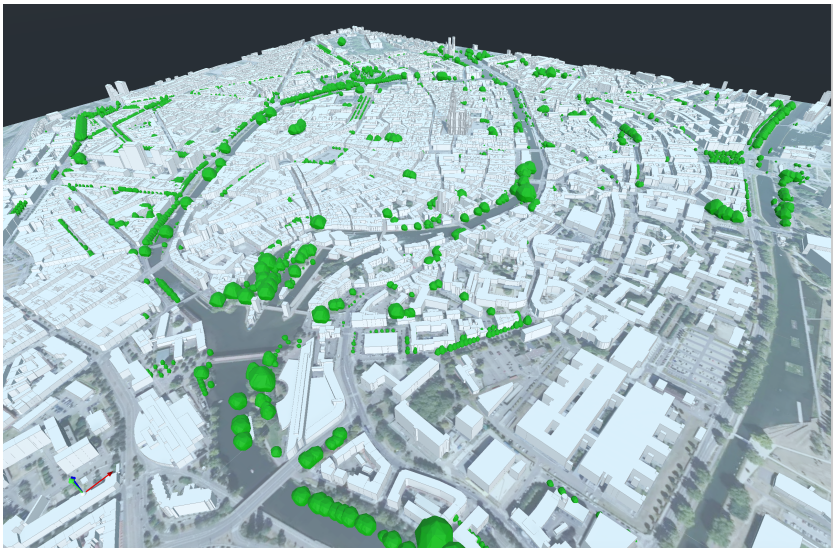


Figure 27: 3D model with trees at LOD 0, city center of Strasbourg

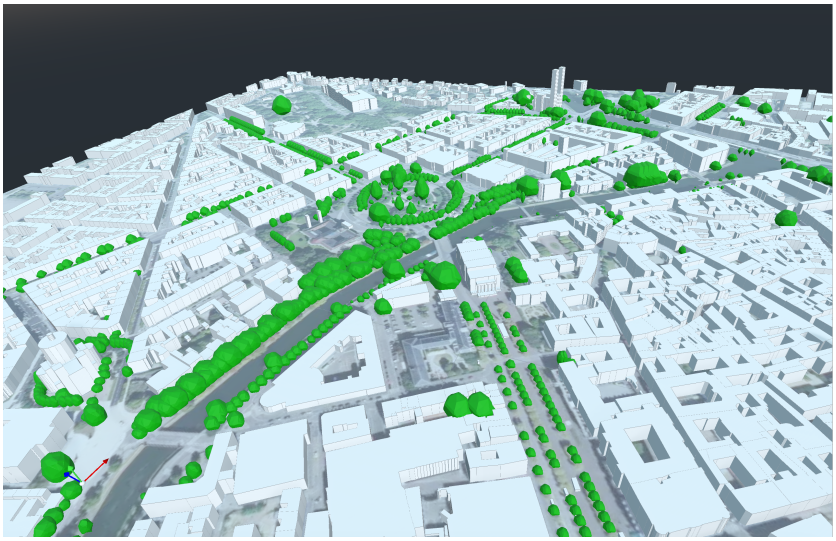


Figure 28: 3D model with trees at LOD 0, place de la République in Strasbourg

These two images consist of two distinct meshes: the tree mesh and the building mesh. For this particular model, we included only trees with a specified height in the `OpenStreetMap` database. Trees without a specified height do not appear in the model.

4.2 Complexity and performance analysis

To assess the complexity and performance of our program, we executed it on the High-Performance Computing (HPC) cluster Gaya. This cluster consists of a DELL PowerEdge R7525 head node and six DELL PowerEdge R6525 compute nodes, providing a total of 768 multi-threaded cores on the compute nodes and 96 cores on the head node. Gaya offers 150 TB of storage for data and an extremely fast 15 TB NVME scratch space. The head node is equipped with two AMD EPYC 7552 48-Core Processors running at 2.2GHz, totaling 192 virtual cores, and 1024 GB of RAM. Each compute node features two AMD EPYC 7713 64-Core Processors running at 2GHz, totaling 256 virtual cores, and 512 GB of RAM. The nodes are interconnected via Broadcom Adv. Dual 10GBASE-T Ethernet and Mellanox ConnectX-6 Dx Dual Port 100 GbE for MPI communication.

For our benchmarks, we used a single exclusive node with 256 cores and 512 GB of RAM.

We used four different bounding boxes Strasbourg city center as our test area, all centered at the same point but with varying sizes:

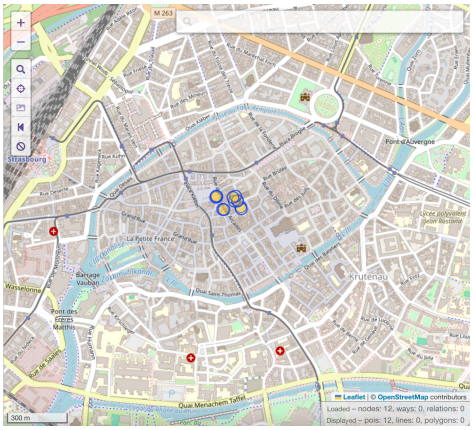


Figure 29: 153.7 m^2, 12 trees

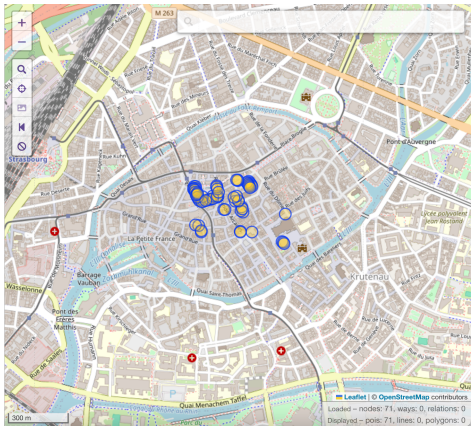


Figure 30: 384.0 m^2, 71 trees

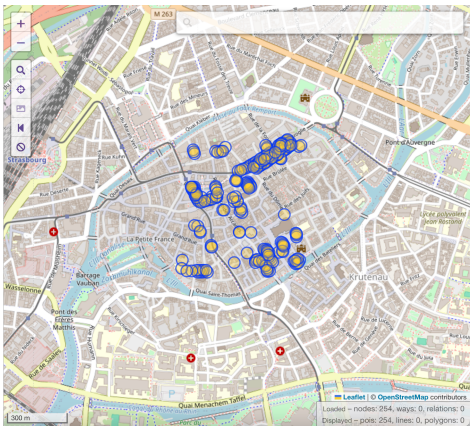


Figure 31: 626.1 m^2, 254 trees

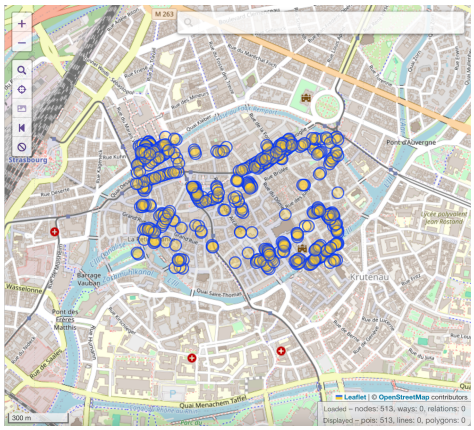


Figure 32: 808.4 m^2, 513 trees

For each bounding box, the program was run for the following Levels of Detail (LODs):

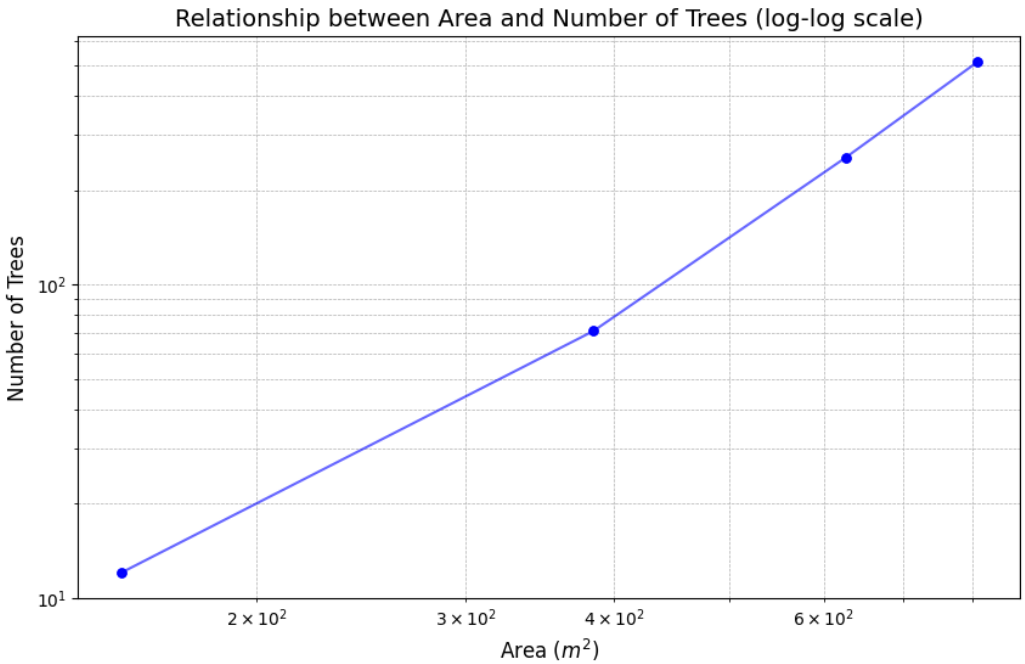
- LOD 0

- LOD 1
- LOD 2
- LOD 3

4.2.1 Area and Number of Trees

Since we're using the area as our main feature (because it is the one we control when meshing an area), we want to examine how the area (in m^2) relates to the number of trees available.

It is important to note that this relationship highly depends on the configuration of the environment. In urban settings, trees are not usually evenly distributed (e.g., avenues, parks, etc.), so we cannot always expect a linear relationship between the area and the number of trees.

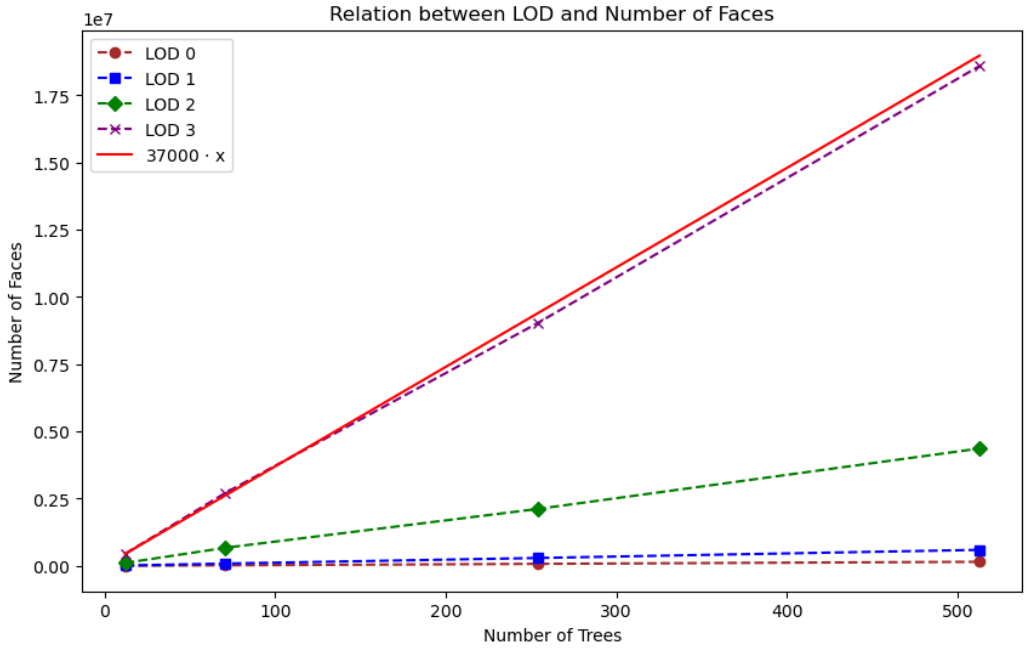


Since the plot is a line on a log-log scale, we can infer that the relationship

between the number of trees and the area (in square meters) follows a power law.

4.2.2 Impact of the level of detail (LOD)

The Level of Detail (LOD) we chose has a significant impact. Since we did not select the LODs in a linear fashion, we aim to examine how the different LODs are related to the number of faces produced in the meshes.



The relationship between the number of faces and the Level of Detail (LOD) is linear. This plot clearly illustrates our choice of LODs, with the last one being significantly more detailed.

4.2.3 Execution time

Finally, we aim to benchmark the time it takes to mesh the area for each Level of Detail (LOD).

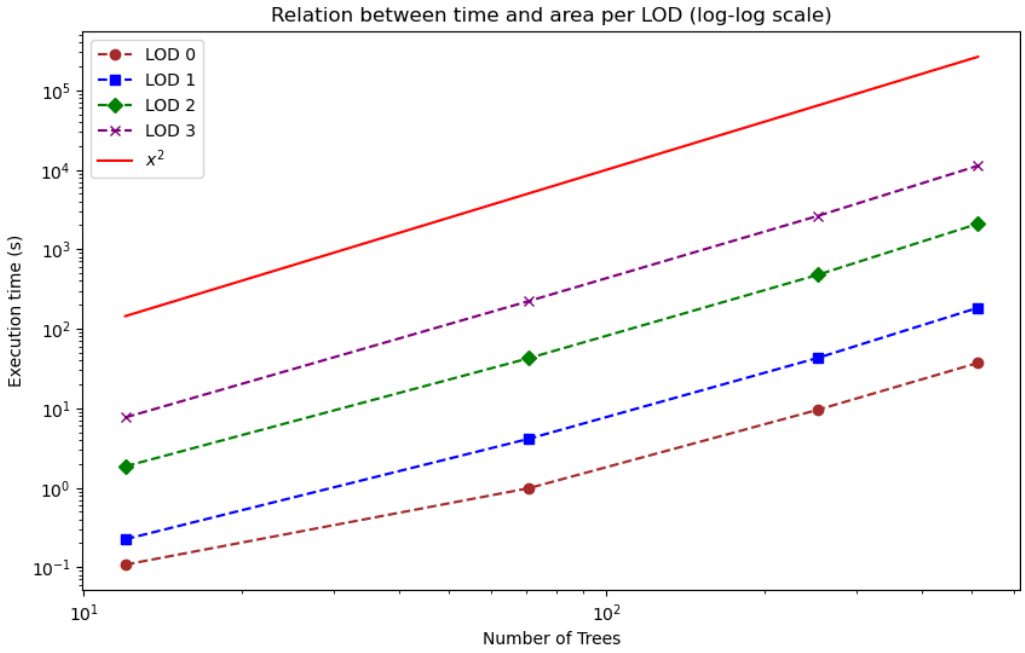


Figure 33: Execution time, union of meshes

This relationship appears to be quadratic. To confirm this with greater precision, we would need more data.

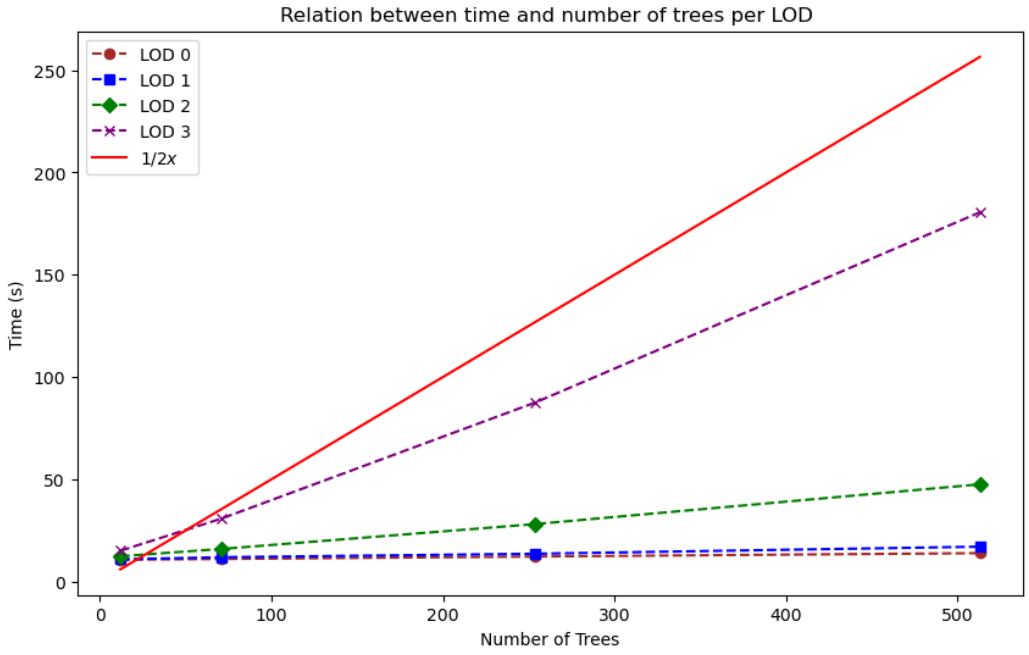


Figure 34: Execution time, union of triangle soup

We can see that making the union using triangle soup instead drastically reduces the execution time to a linear relationship.

5 Prospects

5.1 Account for seasonal changes and leaf fall

One of the developments of our project is to account for seasonal changes and leaf fall. Depending on the season, solar rays will penetrate foliage to varying degrees. To simulate this, we can group leaves into clusters of 5-10 and label them, allowing us to add or remove these clusters in the configuration file based on the season. To simplify, we can categorize the seasons into two groups: more leaves for spring and summer, and fewer leaves for fall and winter.

5.2 Shading calculations

One other development is to simulating light and shade on 3D objects, such as buildings, using the `Feel++`[29] library, which specializes in solving Partial Differential Equations (PDEs). This could help simulate ray tracing and shading effects on buildings.

6 Conclusion

The ExaMA WP1 - Vegetation project represents a significant stride towards enhancing urban modeling through the integration of vegetation, specifically trees, into 3D models of urban environments. By focusing on the Urban Building Model (UBM) use case within the HiDALGO2 initiative, we aimed to improve the accuracy and realism of thermal and energy simulations, contributing to better building energy efficiency and indoor air quality.

Through our efforts, we successfully extracted tree data from OpenStreetMap and utilized the CGAL library to generate 3D tree models. We integrated these models into existing terrain meshes, specifically focusing on the Strasbourg city center. Our methodology, involving data acquisition via the Overpass API, tree model generation using Alpha Wrapping, and precise coordinate conversion through Mercator projection, ensured a robust and scalable approach to urban vegetation modeling.

The use of different Levels of Detail (LOD) allowed for flexible and efficient modeling, catering to various computational and visual requirements. Our benchmarking on the HPC cluster Gaya confirmed the scalability and performance of our approach, highlighting the quadratic relationship between execution time and the number of trees when using union of meshes and a linear relationship when using triangle soup.

Additionally, we addressed the challenge of missing data by implementing a system to assign default values for tree height and genus, ensuring comprehensive coverage even in the absence of complete metadata. Our categorization of tree genera and the creation of a library of tree models further enhanced the versatility and applicability of our solution.

Overall, the ExaMA WP1 - Vegetation project has laid a strong foundation for future urban modeling endeavors, emphasizing the importance of integrating natural elements into urban simulations. The project's outcomes not only contribute to the specific goals of the HiDALGO2 initiative but also provide valuable insights and tools for broader applications in urban planning, environmental science, and computational geometry. As we move forward, the incorporation of seasonal changes, leaf fall, and advanced shading calculations will further refine and expand the capabilities of our

modeling approach, paving the way for more sustainable and livable urban environments.

7 References

References

[1] HiDALGO2. HiDALGO2. https://www.hidalgo2.eu, 2024.

[2] HiDALGO2. About HiDALGO2. https://www.hidalgo2.eu/about, 2024.

[3] HiDALGO2. Carlos Hidalgo. https://www.hidalgo2.eu/urban-building-model, 2024.

[4] Cemosis. Cemosis. https://www.cemosis.fr, 2024.

[5] IRMA. IRMA. https://irma.math.unistra.fr, 2024.

[6] Pierre Alliez. Pierre Alliez. https://team.inria.fr/titane/pierre-alliez, 2024.

[7] INRIA. INRIA. https://www.inria.fr, 2024.

[8] Vincent Chabannes. Vincent Chabannes. https://www.researchgate.net/profile/Vincent-Chabannes, 2024.

[9] Tania Landes. D'où vient le pouvoir rafraîchissant des arbres en ville ? https://theconversation.com/dou-vient-le-pouvoir-rafraichissant-des-arbres-en-ville-199906, 2023.

[10] Yujin Park, Jean-Michel Guldmann, Desheng Liu. Impacts of tree and building shades on the urban heat island: Combining remote sensing, 3D digital city and spatial regression approaches. https://www.sciencedirect.com/science/article/pii/S0198971521000624, 2021.

[11] P. Verchere. Thermal image of a street in the city, 2023.

[12] https://www.openstreetmap.org/help.

[13] CGAL Development Team. CGAL User and Reference Manual. https://doc.cgal.org/latest/Manual/index.html.

[14] Wikipedia. STL (file format) - Wikipedia. https://en.wikipedia.org/wiki/STL_(file_format), 2023.

[15] OpenStreetMap Contributors. Overpass API. https://wiki.openstreetmap.org/wiki/Overpass_API.

[16] curl. https://curl.se/.

[17] CGAL Development Team. CGAL. https://github.com/CGAL/cgal, 2024.

[18] Roland Olbricht. Overpass QL. https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL, 2024.

[19] Wikipedia. SketchUp. https://en.wikipedia.org/wiki/SketchUp, 2024.

[20] MeshLab Developers. MeshLab. https://www.meshlab.net/.

[21] CGAL Development Team. *CGAL 5.6.1 - 2D and 3D Linear Geometry Kernel* , 2024.

[22] Wikipedia. K-nearest neighbors algorithm. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm, 2024.

[23] Wikipedia. Mercator projection. *Wikipedia*, 2024.

[24] Bibm@th. Mercator projection, 2024.

[25] Wikipedia. World Geodetic System. https://en.wikipedia.org/wiki/World_Geodetic_System#WGS_84, 2024.

[26] Christian Berger. WGS84toCartesian. https://github.com/chrberger/WGS84toCartesian?tab=readme-ov-file, 2021.

[27] CGAL Development Team. CGAL 5.6.1 - 3D Alpha Shapes. https://doc.cgal.org/latest/Polygon_mesh_processing/group__PMP__corefinement__grp.html, 2024.

[28] Dimitri van Heesch. Doxygen. https://www.doxygen.nl/index.html, 2024.

[29] Feel++ Consortium. Feel++. https://docs.feelpp.org/home/index.html.