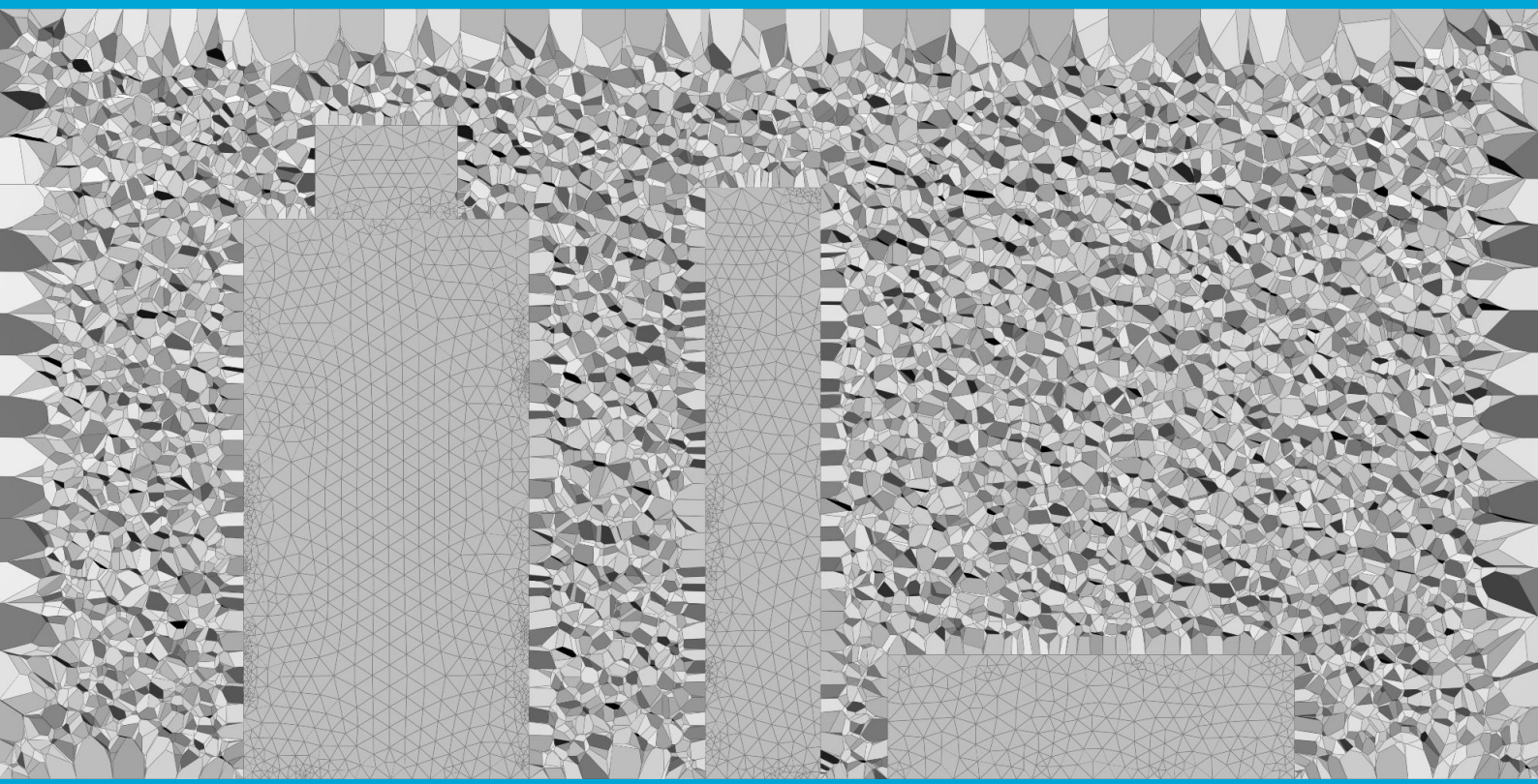
Geomatics Master's thesis

Voronoi mesh generation
tailored for urban flow simulations

Ákos S. Sárkány
2025



TU Delft

MSc thesis in Geomatics

VORONOI MESH GENERATION TAILORED FOR URBAN FLOW SIMULATIONS

Ákos S. Sárkány

December 2025

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

"One does not ask whether Euclidean geometry is true, but whether it is convenient."

— Henri Poincaré

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors: Dr. Clara García-Sánchez
Dr. Hugo Ledoux
Co-reader: Dr. Akshay Patil

Abstract

Accurate urban Computational Fluid Dynamics (CFD) simulations require volumetric meshes whose boundary conforms to complex city geometry while maintaining cell quality and enabling adaptive mesh sizing. This thesis develops a Voronoi-based polyhedral meshing methodology tailored to the output of City4CFD, where the simulation domain is provided as a triangulated Piecewise Linear Complex (PLC) whose faces are grouped into semantic boundary patch types. The central goal is to generate a boundary-conforming polyhedral mesh *without* clipping Voronoi cells against the boundary, and to preserve patch information so that CFD boundary conditions can be transferred consistently to the mesh.

The method constructs a set of boundary (surface) Voronoi sites by intersecting triplets of spheres centered at the vertices of a refined surface triangulation. Building on the sphere-based sampling conditions of VoroCrust, sphere radii are initialized and iteratively adjusted to satisfy smooth-coverage, smooth-overlap, and Lipschitz-type size-transition constraints, while a shrinking step resolves "half-covered" seed configurations. The surface triangulation is refined by splitting triangles and protecting edges, then regularized with centroidal smoothing until the triangles associated with each facet intersect in two points, yielding paired sites on opposite sides of the boundary. These sites induce Voronoi facets that coincide with the input surface, producing triangular boundary faces and avoiding cell clipping. Patch-boundary preservation is enforced by identifying protected edges not only via dihedral-angle sharpness but also via changes in patch groups, ensuring that 1D interfaces between patch types are explicitly represented in the resulting Voronoi boundary.

To populate the mesh interior with Voronoi sites, the thesis evaluates several strategies (uniform random scattering, adaptive distance-based refinement, and structured lattices) to regulate cell density and promote larger cells away from geometric features. A prototype implementation in C++ using CGAL demonstrates feasibility for 2-manifold inputs and produces boundary-conforming Voronoi meshes compatible with OpenFOAM-style polyhedral representations. The approach assumes a valid 2-manifold boundary and does not repair non-manifold or overlapping input geometries.

Keywords: Voronoi meshing, polyhedral mesh generation, computational fluid dynamics, urban flow simulation, boundary conforming meshes

Contents

Acronyms

Symbols

Here are some symbols I use throughout the thesis:

$\mathcal{T}$	Input Piecewise Linear Complex
σ	A facet in $\mathcal{T}$
$\mathcal{M}$	Boundary of a volume
$\mathcal{P}$	Set of sample points
$\mathcal{B}$	Set of surface balls
b_i	Any ball in $\mathcal{B}$
$\mathcal{U}$	Union of balls $\cup\mathcal{B}$
sz	Sizing parameter
θ	Sharp edge threshold
δ	Dihedral angle between adjacent facets
$\mathscr{P}(f_i)$	The patch group of face p_i
$\mathcal{H}$	A half edge graph structure of a surface represented by $\mathcal{T}$
$\mathcal{K}$	The kD tree storing the vertex locations of $\mathcal{T}$
$\mathcal{H}_{PE}$	The set of protected edges in $\mathcal{H}$
$\mathcal{H}_{PV}$	The set of protected vertices in $\mathcal{H}$
$\mathcal{H}_{SV}$	The set of vertices having at least one incident protected edge
$\mathcal{H}_{RV}$	The inverse of $\mathcal{H}_{SV}$, the set of regular vertices

1. Introduction

Accurate urban wind-flow simulations are vital for understanding microclimate, urban heat, and pollutant dispersion. A conventional CFD workflow relies on volumetric meshes with a finite number of discrete cells on which it performs the computations. When a CFD simulation is run in an urban context, the domain of the simulation is the air volume surrounding the buildings. To make the volume finite, the air volume is generally represented by its boundary using a PLC. However, since the CFD simulation expects discrete volumetric cells, the boundary representation must first be discretized into cells. This discretization process, called mesh generation (or simply, *meshing*), is a significant challenge that has been extensively researched over the years due to its importance not only in CFD, but also when computing internal stresses in solid elements in a Finite Element Analysis (FEA). The resulting discretized interior volume (mesh) can be characterized by several aspects, including how closely its boundary follows the original geometry, the shape, and size of the mesh cells. When the mesh follows the original boundary perfectly, it is said to conform to that boundary. This is always a desirable property when generating a mesh.

The shape of the cells is one of the most important properties of a mesh, since poorly shaped or degenerate cells can negatively affect a simulation or even prevent it from running. Additionally, cell shape directly influences the computational complexity of CFD simulations (López-Pachón & Marcé-Nogué, 2025). Hexahedral cells simplify numerical computations, but restricting the mesh to this cell type makes it difficult to conform to non-orthogonal features. Tetrahedral cells, on the other hand, can always conform to a boundary represented by a PLC. However, each hexahedron can be subdivided into five tetrahedra, meaning that for the same number of vertices, the mesh will contain five times as many cells, significantly increasing simulation time. Polyhedral cells offer a compromise by balancing geometric flexibility and cell quality. The main challenge with polyhedral meshes lies in enforcing conformity to the original boundary. While it is always possible to construct a polyhedral cell that conforms to a geometric feature, for example, by clipping the cells against the boundary, this approach can degrade mesh quality by creating concave, sliver, or extremely small (degenerate) cells (Figure 1.1) (Ebeida & Mitchell, 2012). Although polyhedral cells provide great flexibility, computations on them are generally more expensive than on hexahedral cells. A way to improve the efficiency of polyhedral cells is to construct them from a Voronoi diagram (discussed in Section 2.2), which partitions space according to nearest-neighbor relations and always produces convex polyhedral cells. Furthermore, the lines connecting the centroids of the cells are always perpendicular to the face shared by the corresponding cells, which is a beneficial property in CFD (López-Pachón & Marcé-Nogué, 2025).

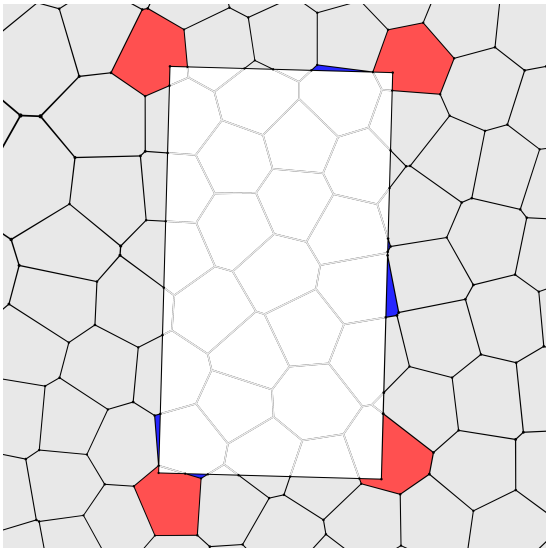


Figure 1.1.: Potential issues arising due to clipping cells against a boundary are concave (red) and sliver/irregularly sized cells (blue).

Another important factor influencing the efficiency of a CFD simulation is cell density, which is strongly correlated with cell sizing. Smaller cell sizes result in higher density, and as density increases, the total number of cells in the mesh grows accordingly. The main characteristic influencing cell size is the *local feature size* of the original boundary. This refers to geometric or topological features such as sharp edges, corners, and regions of high mean curvature that locally constrain the admissible element size. Cells that are too large relative to nearby feature sizes fail to capture the boundary geometry, making conformity to the original boundary impossible. Reducing the cell size allows smaller features to be resolved, but this comes at the cost of increasing the total number of cells. This relationship between feature size and cell size can be interpreted as a measure of mesh resolution. Similar to image resolution, which describes the relationship between the number of pixels and the detail the image captures, the resolution of a mesh determines the smallest feature that can be captured on the boundary. Increasing resolution also enables finer detail in the simulated flow during CFD, but slows down the simulation due to the increased computational load.

In urban CFD, simulations are typically performed on large and complex domains. These domains consist of 3D models of urban regions, such as city districts or entire towns, containing multiple buildings enclosed by a bounding geometry (e.g., a box). According to practices discussed by Blocken (2015), the size of the computational domain should scale with the height of the tallest building it contains. This results in a large volume that includes relatively small features, such as buildings and other geometric elements, e.g. trees or street furniture. This situation is challenging because accurately capturing small features requires appropriately small cells, but filling the entire domain with such cells leads to an excessive number of elements. A common approach is to use fine cells near small features and gradually transition to larger cells farther away. In the urban CFD context,

small cells near boundaries are essential not only for preserving geometric features but also for resolving flow behavior near surfaces. As a result, it is crucial that the mesh boundary closely conforms to the original geometry, particularly by preserving sharp features (Blocken, 2015). Since flow behavior is relatively uniform in most of the volume and most interesting phenomena occur near buildings, cell density should be high near these features and sparse elsewhere, with density regulated to capture detail and cell shape adjusted to maintain boundary conformity.

The final two important aspects discussed here are the robustness and automation of the meshing process. Robustness is especially critical when dealing with large domains. One key factor is scalability, which describes how well the process handles increasing input size or geometric complexity. More specifically, it refers to how the complexity of the meshing process grows as the size of the urban area increases, directly affecting the approach's overall scalability. Another important consideration is parallelization, meaning whether the process can efficiently scale with available hardware resources. Automation is important for integration into an end-to-end pipeline. The fewer manual configurations required during meshing, the better. Ideally, each meshing parameter should be derived directly from the input geometry.

1.1. Motivation & Research questions

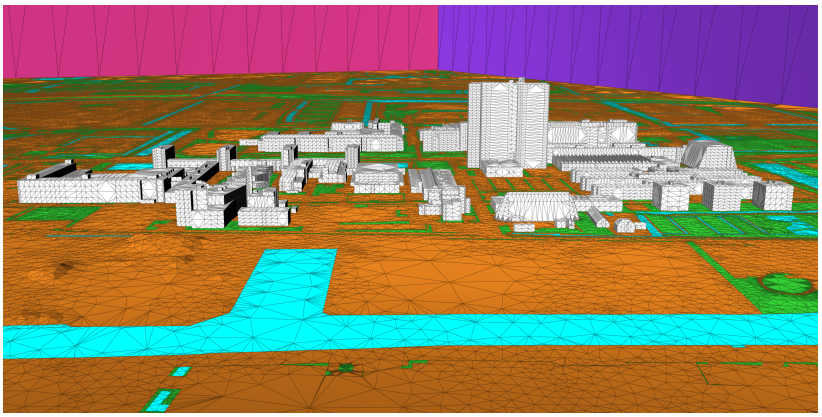


Figure 1.2.: Output of City4CFD is a triangulated PLC with patch information about boundary types illustrated with different colors. The meshing domain equals the volumes of the buildings (grey) subtracted from the bounding box, which consists of the terrain elements (brown, green, blue) and the boundary elements (purple, pink). During meshing, the 1D boundaries (edges) between surface patches must be preserved.

Today, automation and pipeline development are important concepts. In the context of urban CFD, a fully automatic pipeline would enable wind simulations for arbitrary urban regions without extensive upfront investment. This would support smarter design decisions

for new developments and help to prevent wind pollution and constrain the dispersion of hazardous airborne pollutants.

Pađen et al. (2022, 2024) introduce methods to automatically reconstruct buildings and terrain in urban areas using raw point cloud and Kadaster datasets. Their open-source software, City4CFD, reconstructs the boundary representation of the CFD domain and provides a starting point for an automated pipeline by generating a PLC that represents the domain boundary. It not only reconstructs the geometry, but also separates primitives with different boundary patch types, such as buildings, vegetation, and water, into distinct groups in the output PLC (Figure 1.2). However, City4CFD is not yet capable of directly producing a mesh suitable for CFD simulations.

In their pipeline, Pađen et al. (2022) use the snappyHexMesh utility of OpenFOAM to generate the mesh for the CFD simulation. This tool, however, often produces suboptimal meshes and requires careful manual configuration to achieve acceptable results. The iterative refinement process of snappyHexMesh is highly sensitive to its configuration, and automatic meshing of sharp or small geometric features is problematic in many cases (Pađen et al., 2022). As a result, the generated mesh may fail to conform to the original boundary, introducing inaccuracies in the simulation. Additionally, the resulting meshes often have many cells. The method produces a hybrid mesh with multiple cell types by repeatedly splitting cells and moving vertices to improve boundary approximation. Convergence is slow and does not necessarily yield a perfectly conforming mesh, especially because the maximum number of refinement steps is often reached before convergence occurs. Furthermore, cell sizing is configured manually and is not adapted to the local feature size, so small features may not be captured.

My motivation for doing this thesis is to investigate techniques to improve the existing pipeline by developing a methodology to construct a mesh from the output of City4CFD that conforms more closely to the input geometry than the mesh produced by the current snappyHexMesh-based approach. This improvement should be achieved without degrading cell quality or (significantly) increasing cell count, while also requiring less or equal manual configuration. I aim to accomplish this by generating polyhedral Voronoi cells whose boundaries conform to the input geometry upon initialization, thereby avoiding additional processing steps such as clipping. This approach is feasible because Voronoi cells can conform to lower-dimensional features (Merland et al., 2014), although the main challenge lies in determining how to place the Voronoi sites.

In this thesis, I identify and review existing meshing techniques that use 3D Voronoi cells to construct polyhedral meshes. Building on these methodologies, I propose a novel approach in which the configuration, shape, and size of the cells are tailored to urban CFD simulations. Furthermore, the proposed process uses the output of City4CFD as input, providing a pathway to further extend the reconstruction pipeline for urban CFD. More precisely, this thesis aims to answer the following research questions:

Is it possible to generate a polyhedral mesh that directly conforms to a boundary without clipping the cells while preserving boundary patch information?

1. Under what conditions do $n - dimensional$ Voronoi diagrams conform to lower-dimensional features?
2. Is it possible to derive the necessary conditions from a piecewise linear complex?
3. Is it possible to conform the Voronoi cell boundaries to the boundaries separating surface patch types? How to transfer the boundary information to the generators of the Voronoi cells?
4. Is it possible to regulate cell sizes and construct larger cells in regions without small features?
5. Is it possible to achieve an adaptive cell density where the cell sizes change gradually between neighboring cells, and increase their size towards the interior of the mesh?

1.2. Scope of the thesis

In this thesis, I identify relevant literature and state-of-the-art techniques for three-dimensional mesh generation, with particular attention to mesh quality requirements in urban CFD. Building on recent methodological advances, I derive a new approach for generating Voronoi-based meshes tailored to the output of City4CFD. By tailored, I refer to a strategy that directly consumes the `.obj` representation produced by City4CFD. This represents the entire domain boundary as a single closed manifold PLC where surface faces are grouped according to material or boundary-condition patches

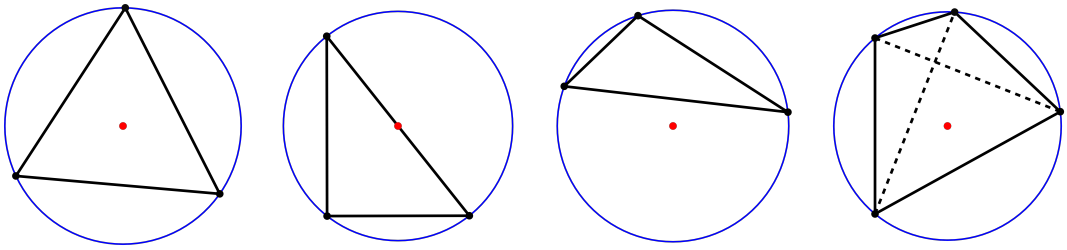
Starting from this triangulated PLC boundary, the proposed method aims to reconstruct a set of Voronoi seeds whose Voronoi diagram matches the input boundary exactly. The seeds associated with boundary-conforming Voronoi cells must also preserve the patch information of the surface from which they were recovered. In addition, the method seeks to reconstruct the complete Voronoi diagram as a discrete set of finite polyhedral cells, each described by a unique set of faces and vertices, so that the resulting mesh structure is compatible with the requirements of OpenFOAM (Subsection 2.3.1). However, actual testing with OpenFOAM is out of scope. Furthermore, outside the scope of this thesis is the correction or repair of defective input geometries. I assume that the input boundary is a valid 2-manifold; the presented methodology is not expected to perform reliably if this assumption is violated.

2. Relevant Concepts

In this section, I discuss concepts essential for understanding this thesis.

2.1. Delaunay Triangulation and Tetrahedralization

Given a 2D triangle, its circumcenter is the point equidistant from all three of its vertices, with distance d. The corresponding circumcircle is the circle centered at this circumcenter with radius d. By definition, the triangle's vertices lie on the boundary of this circle. Note that the circumcenter does not necessarily lie inside the triangle: it lies on the boundary for right-angled triangles and outside the triangle for obtuse triangles. Figure 2.1 illustrates these cases.



(a) (b) (c) (d)

Figure 2.1.: (a-c) Different positions for the circumcenter depending on the type of triangle. (d) If there are more than three vertices that are cocircular, then there is more than one unique triangulation for the points.

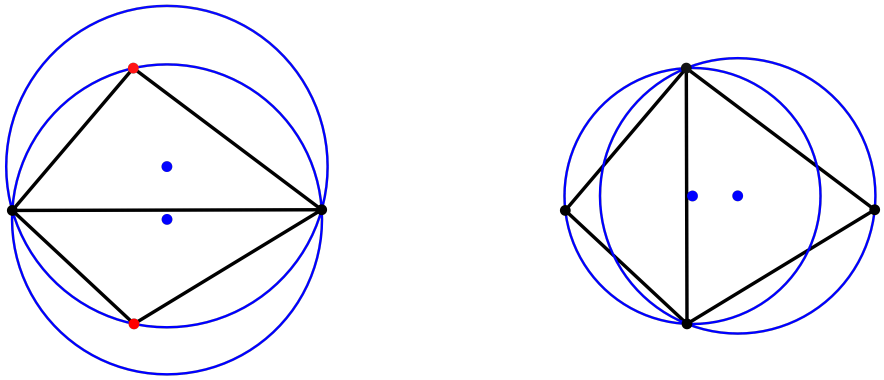
For a point set $P \subset \mathbb{R}^2$ with at least three non-collinear points, one can construct a triangulation where the vertices are the points in P, with edges connecting pairs of points, and faces are triangles. Each face has a unique circumcircle. A triangulation is called a *Delaunay triangulation* if none of these circumcircles contains any other point of P in its interior. Figure 2.2a shows a configuration where the Delaunay condition is violated, and Figure 2.2b shows how to triangulate case b correctly to ensure Delaunay conditions.

If the point set contains no duplicates, at least one Delaunay triangulation always exists. However, the Delaunay triangulation is not always unique. Degenerate configurations where four points lie on a common circle admit multiple valid triangulations. In such a case, the circumcenters of all admissible triangles coincide at the center of the shared circumcircle

(Figure 2.1d), and the diagonal of the quadrilateral may be flipped without violating the Delaunay condition.

Most implementations of planar Delaunay triangulations introduce an auxiliary *infinite vertex* representing the unbounded region outside the convex hull of P. This vertex has no geometric position; it is a topological construct that allows the triangulation to be modeled as a closed 2-manifold rather than an open planar graph. Edges and faces incident to this infinite vertex encode the convex hull structure and ensure that the data structure remains manifold. These infinite elements have no geometric meaning and exist solely for representation and algorithmic convenience.

The concept generalizes naturally to three dimensions. A *Delaunay tetrahedralization* of a point set $P \subset \mathbb{R}^3$ is a decomposition of the convex hull into tetrahedra such that the circumsphere of each tetrahedron contains no other points of P in its interior. Analogously to the 2D case, degeneracies occur when five or more points lie on a common sphere.



(a) (b)

Figure 2.2.: Subfigure (a) illustrates an example of a non-Delaunay triangulation. The red vertices violate the Delaunay conditions since they lie inside the circumcircle corresponding to another triangle. (b) showcases that the triangulation can be made Delaunay by *flipping* the edge.

2.2. Voronoi Diagram

To introduce Voronoi diagrams, we again begin with the planar case. Given a point set $P \subset \mathbb{R}^2$, the Voronoi diagram is a subdivision of the plane into vertices, edges, and cells. Each Voronoi cell c_i corresponds to a point $p_i \in P$, whose location relative to the other points determines the shape of the cell. Throughout this thesis, I use the verb *generate* to describe this process, and I refer to each generating point p_i as a *Voronoi site* or *Voronoi seed* interchangeably.

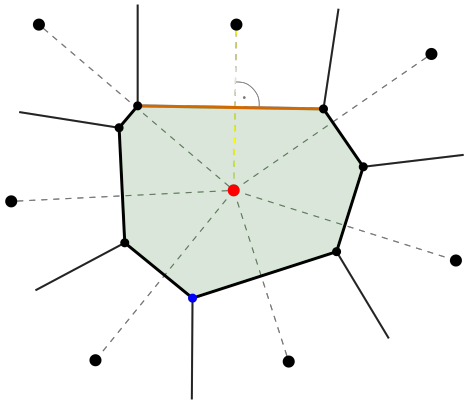


Figure 2.3.: Illustration of four different elements in a Voronoi diagram. In red: a *Voronoi site*, in blue: a *Voronoi vertex*, in green: a *Voronoi cell*, and in yellow: a *Voronoi edge*. A useful property of Voronoi diagrams is the orthogonality between cell boundaries and the lines connecting the sites of the corresponding cells (dashed lines).

A cell is defined by the set of all points $x \in \mathbb{R}^2$ which are closer to p_i than to any other sites $p_j \in P$:

$$c_i = \left\{ x \in \mathbb{R}^2 \;\middle|\; \|x - p_i\| < \|x - p_j\| \;\; \forall p_j \in P,\; j \neq i \right\}.$$

A Voronoi edge $e_{i,j}$ is the locus of points equidistant to exactly two sites p_i and p_j, and closer to these than to any other site. A Voronoi vertex is a point equidistant to at least three sites. Equivalently, it is the circumcenter of a circle passing through a subset $P_{cc} \subseteq P$ with $|P_{cc}| \geq 3$.

In many practical implementations, the Voronoi diagram is augmented with an additional *infinite vertex* to represent the unbounded region outside the convex hull of the point set P. Cells associated with sites on the convex hull extend unbounded, and the infinite vertex provides a convenient topological handle, allowing the diagram to be represented as a closed graph rather than an open planar complex. Edges and rays that extend to infinity are modeled as being incident to this infinite vertex, mirroring the convention used in Delaunay triangulations. As in the Delaunay case, this vertex has no geometric coordinates and serves solely as a conceptual construct that simplifies data structures and traversal of the diagram.

The construction generalizes to any dimension. For a point set $P \subset \mathbb{R}^n$, a Voronoi cell is the set of points whose nearest neighbor in P is a given site. A Voronoi vertex is a point equidistant to at least $n+1$ sites. Voronoi edges, faces, and higher-dimensional elements arise as sets of points equidistant to $k < n+1$ sites, where the dimension of the element

is $n - k + 1$. From now on, whenever I use the term *Voronoi edge*, I refer to the element equidistant from exactly two sites, whichever dimension it may have.

2.2.1. Duality between Delaunay Triangulation and Voronoi Diagram

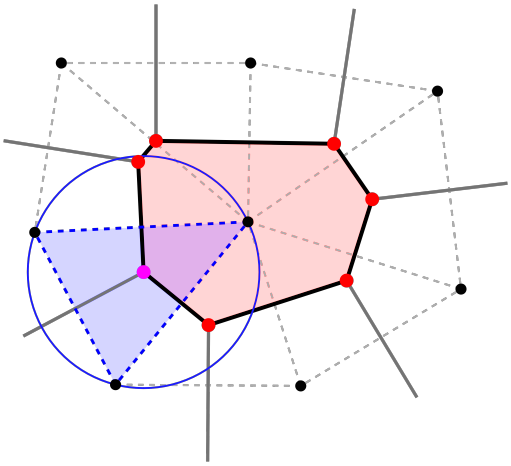


Figure 2.4.: The duality between the Delaunay triangulation and the Voronoi diagram of the same set of points. The circumcenters of the Delaunay elements form the vertices of the Voronoi diagram, enabling the direct construction of Voronoi diagrams from Delaunay triangulations or tetrahedralizations.

There is a natural geometric duality between the Delaunay triangulation and the Voronoi diagram of the same point set P. Figure 2.4 illustrates this relationship.

Dual elements. The correspondence is as follows:

- The Voronoi cell of a site p_i is dual to the Delaunay vertex p_i.
- A Voronoi edge between the cells of p_i and p_j is dual to the Delaunay edge (p_i, p_j).
- A Voronoi vertex equidistant to p_i, p_j, p_k is dual to the Delaunay triangle (p_i, p_j, p_k).
- In $\mathbb{R}^3$, a Voronoi face shared by several cells is dual to a Delaunay tetrahedron.

More generally, in $\mathbb{R}^n$, a Voronoi element equidistant to k sites corresponds to a Delaunay simplex with those k vertices.

Orthogonality. Dual elements intersect orthogonally:

- Voronoi edges are perpendicular bisectors of their dual Delaunay edges.
- In 3D, Voronoi faces are orthogonal to their dual Delaunay triangles.
- The circumcenter of a Delaunay simplex lies at its corresponding Voronoi vertex.

Topological inversion. The duality also reverses the dimension:

0D A Delaunay vertex corresponds to a Voronoi cell.

1D A Delaunay edge corresponds to a Voronoi edge.

2D A Delaunay triangle corresponds to a Voronoi vertex.

Thus, the duality is both geometric (orthogonality, circumcenters) and combinatorial (dimension reversal).

2.2.2. Power diagrams

Power diagrams generalize Voronoi diagrams to settings where each site is assigned an additive weight. Aurenhammer (1987) showed that the distance governing such diagrams is the *power distance*

$$\Pi_i(x) = \|x - p_i\|^2 - r_i^2,$$

which compares the position x to a sphere centered at p_i with radius r_i. Two spheres induce an affine boundary where their power distances are equal, and the intersection of such boundaries defines the vertices of a power diagram. Edelsbrunner and Shah (1996) demonstrated the dual relationship between power diagrams and regular (weighted) triangulations, in which weighted spheres govern local geometric structure.

Ju (2007) introduces a density function that defines a power-distance metric, thereby weighting the iterations during centroidal smoothing. Under this metric, generators are positioned to minimize a density-weighted energy, thereby attracting more vertices to high-density regions and enforcing finer triangle resolutions. The resulting centroidal Voronoi tessellation adapts cell sizes according to the prescribed density, and its dual Delaunay triangulation inherits this spatially varying refinement. In practice, this allows the triangulation to automatically increase resolution near boundaries, interfaces, or other geometrically necessary regions without manual point insertion.

2.3. Computational Fluid Dynamics

Computational Fluid Dynamics solvers compute fluid dynamics by numerically resolving the Navier–Stokes equations over a CFD domain, which consists of a finite collection of non-overlapping control volumes. Each control volume (or cell) represents a discrete portion of the fluid and stores the physical quantities required by the numerical scheme. The solver advances these quantities by evaluating fluxes across cell interfaces, thereby producing an iterative solution of the flow field.

In contrast, the meshing domain is the geometric region to be simulated, typically provided as a boundary representation (B-Rep) such as a surface mesh. Meshing is the process of converting this geometric description into the discretized volume by filling the interior of the B-Rep with a valid tessellation of volumetric cells. Thus, the mesh is not the geometry itself but the discretized computational space derived from it. The quality, shape, and connectivity of the resulting cells directly influence the accuracy, stability, and efficiency of the CFD solver (see Section 3.1).

Because simulations run on a discrete domain, boundary information must also be encoded explicitly. For this purpose, CFD frameworks partition the geometric boundary into *surface patches*, each representing a unique, non-overlapping part of the meshing domain's boundary. These patches are then assigned boundary condition types, which prescribe the behavior of the flow variables at the corresponding portions of the CFD domain. Accurate and consistent patch definitions are therefore essential for the solver to apply the intended physical constraints.

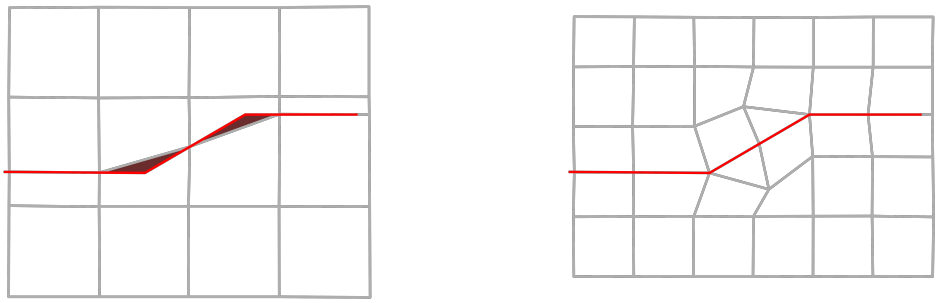
2.3.1. OpenFOAM

OpenFOAM is a widely used open-source software suite for CFD and related continuum-mechanics simulations. Like other finite-volume solvers, it computes solutions to flow problems over a discretized domain of cells. Its open-source nature and modular architecture make it suitable for research, customization, and integration with automated meshing pipelines, which motivates its use as the target format in this work.

OpenFOAM represents meshes using a polyhedral finite-volume structure, in which a list of faces defines each cell, and an ordered list of vertices characterizes each face. The mesh description includes three primary components: the points file (vertex coordinates), the faces file (face–vertex connectivity), and the owner/neighbor files (cell–face relations). Boundary information is stored separately in the boundary file, which groups faces into named patches and assigns each patch a boundary condition type (e.g. wall, inlet, outlet, symmetry, or empty). These patch types determine how the solver constrains velocity, pressure, or other transported fields at the corresponding part of the domain boundary.

For a mesh to be valid in OpenFOAM, boundary faces must conform precisely to the geometric boundary of the domain. If the mesh does not conform e.g. if surface cells do

not align with or correctly represent the intended domain boundary, faces may be assigned to incorrect patches or may fail to define the boundary at all. Figure 2.5b illustrates such a situation: although the surface geometry visually appears correct, the discrete cell faces may not coincide with the boundary surface, resulting in missing or misplaced boundary conditions. Ensuring conformity is therefore essential for both numerical correctness and physical interpretability of the simulation.



(a) (b)

Figure 2.5.: Illustrations: a case when the mesh cells conform to the domain boundary in red (a) and a case where the cell boundaries do not conform to the original boundary (b).

2.4. City4CFD

City4CFD is a digital tool for automatic geometry reconstruction, based on the methodologies discussed by Pađen et al. (2022, 2024). The purpose of the software is to reconstruct the geometry of a selected area. The resulting geometry is a PLC representation of the surfaces that constrain the domain of a CFD simulation. This includes buildings and ground surfaces within the selected region (Figure 1.2). Optionally, upon request, it also constructs lateral walls, and the top boundary can be reconstructed to make the domain finite.

Buildings are reconstructed using point clouds to derive height information and roof features, while building footprints are obtained from Kadaster datasets. Terrain height is reconstructed from point cloud data, and surface patch types are assigned using a 2D polygonal representation of surface types. The terrain triangulation is constrained to conform to the edges of the provided polygons. They use the Dutch national registry for large-scale topography (BGT) to obtain these polygons, but they may also be obtained from other sources, such as OpenStreetMap. These bounding geometries can be constructed using either rectangular or circular layouts, using automatic or manual sizing for their extents.

The current version of the software, available on GitHub at `tudelft3d/City4CFD`, outputs the reconstructed boundary in two separate parts. The first part consists of the buildings,

each represented by a watertight boundary. It is important to note that if two buildings share a common wall, the reconstruction may produce overlapping faces (Figure 2.6. The second part is the union of the terrain and the bounding geometry. In this step, the bounding walls are sewn to the ground surface to form a second watertight geometry. The bottoms of the buildings are positioned entirely below the ground surface.

The reconstructed geometry is represented as a PLC consisting exclusively of triangular faces. Each face belongs to a single boundary patch type (e.g., buildings, vegetation, ground, etc.), and edges shared by faces of different patch types therefore define the boundaries between surface patches. To preserve this information, the software either outputs each patch type as a separate geometry file or encodes the patch type as metadata, for example, as primitive groups in `.obj` files.

A feature currently under development in City4CFD allows the user to merge these two geometries. This process computes their intersection and removes all surface regions covered by the other geometry. In this way, the actual 2-manifold boundary of the CFD domain can be obtained.

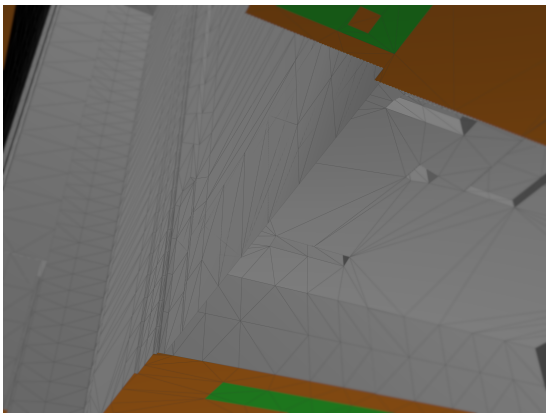


Figure 2.6.: A potentially problematic case is when the boundary has overlapping faces.

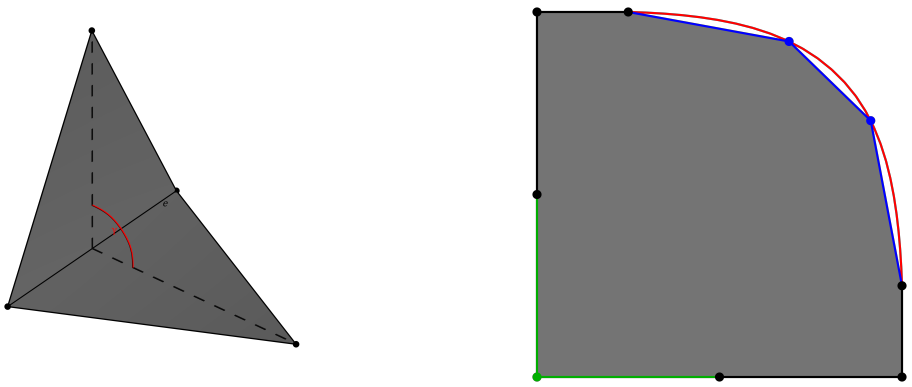
2.5. Boundary representation

It is possible to represent an abstract boundary $\mathcal{M}$ in multiple ways. One popular representation is a PLC $\mathcal{T}$, which provides a discrete approximation $\hat{\mathcal{M}}$ of $\mathcal{M}$. The boundary of a volume must be closed—i.e., without holes—and non-self-intersecting. A pragmatic way to encode such a structure is by using a half-edge graph.

A half-edge data structure is fundamentally composed of only two primitive entities: *vertices* and *half-edges*; yet it conceptually represents four: vertices, half-edges, edges, and faces. Each half-edge is an oriented edge segment that stores references to the vertex it originates from (source) and to the vertex it points to (target). A pair of opposite

half-edges connecting the same two vertices collectively represents an (undirected) edge. Similarly, a face is not stored as a separate primitive but is implicitly defined by a cyclic loop of half-edges . The half-edge may reference the half-edge pointing in the opposite direction or the incident face, depending on the implementation of the data structure. This explicit adjacency encoding makes it straightforward to traverse the boundary of a face, move across faces via opposite half-edges, and efficiently walk through the entire mesh. Although half-edge structures are commonly implemented using pointer-based references, equivalent variants based on index tables or array-based storage offer improved memory control and can be advantageous in performance-critical contexts.

2.5.1. Sharp Features



(a) The dihedral angle γ at an edge e is defined as the angle enclosed by the planes of the adjacent faces.

(b) The dihedral angle can be used to distinguish between sharp and smooth edges using a sharp angle threshold θ. $\alpha < \theta$ therefore the section in green marked as sharp, while $\beta > \theta$ thus section in blue is not. The red stroke indicates how the surface should be interpreted at smooth edges.

Figure 2.7.: Boundary representation with a piecewise linear complex.

Because $\hat{\mathcal{M}}$ is a discrete approximation of $\mathcal{M}$, the boundary represented by $\hat{\mathcal{M}}$ is never perfectly smooth. As shown in Figure 2.7b, a smooth curve on $\mathcal{M}$ becomes a sequence of planar segments forming corners on $\hat{\mathcal{M}}$. To encode the latent "curved" nature of such regions, we can distinguish flat and sharp features on $\hat{\mathcal{M}}$ using the *dihedral angles* between adjacent faces.

Given an edge e shared by two incident faces, they define a dihedral angle δ, the angle enclosed by the planes embedding the faces (Figure 2.7a). For a chosen threshold θ, edges whose dihedral angle exceeds this threshold are classified as sharp. Figures 2.7b illustrate

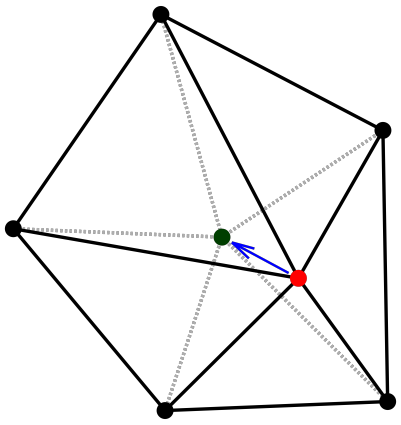


Figure 2.8.: The centroidal smoothing illustrated for a single vertex. In red, the original vertex position, and in blue, the new vertex position. The new location of a vertex is computed as the average of the positions of the vertices around it.

the distinction between sharp and flat features and how these categories influence both interpretation and visual rendering.

On a surface containing both flat and sharp edges, two points $x, y \in \mathcal{T}$ are called *co-smooth* if there exists a path between them on the surface that does not cross any sharp edge. Co-smoothness provides a convenient way to reason about smooth subsets of the boundary even in the presence of discrete approximations.

2.6. Regularization and smoothing

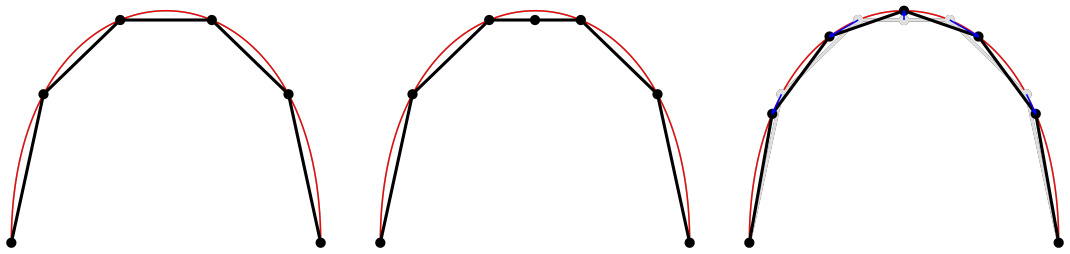
Smoothing aims to improve the regularity of a surface mesh by repositioning vertices to reduce noise, irregularity, or discretization artifacts. A classical and widely used method is Lloyd (centroidal) smoothing. In its basic form, the process replaces the position of a vertex with the centroid of its neighboring vertices (Figure 2.8). This idea is entirely dimension-agnostic: if a mesh intrinsically lives in a flat Euclidean space $\mathbb{R}^k$, then the centroid of any local neighborhood also lies in $\mathbb{R}^k$, and therefore the smoothing step is naturally compatible with the domain. Repeated application of this update tends to produce uniform vertex distributions and improved element shapes (Ju, 2007).

However, many meshes do not represent subsets of a flat Euclidean space but instead approximate a lower-dimensional manifold embedded in a higher-dimensional Euclidean space. A visualizable example is a 2-manifold surface embedded in $\mathbb{R}^3$. Even though the ambient space is Euclidean, the manifold itself is curved, and its intrinsic geometry differs from the geometry of the embedding space. Consequently, while the centroid of a vertex

neighborhood always lies in $\mathbb{R}^3$, it generally does *not* lie on the underlying surface. This is not a special case but a general phenomenon: for any embedded manifold of codimension greater than zero, Euclidean centroids drift away from the manifold unless the manifold happens to be flat in the embedding (e.g., a planar patch). Classic examples, such as spherical surfaces, illustrate this clearly: the centroid of points sampled on a sphere is located strictly inside the sphere and thus outside the manifold. Figure 2.9a demonstrates this phenomenon.

Applying Lloyd smoothing directly to such embedded manifolds, therefore, introduces systematic geometric errors. Each smoothing step pulls vertices away from the surface, resulting in shrinkage, loss of curvature, and a gradual flattening of features. Although the method itself remains mathematically valid, it does not respect the intrinsic geometry of the manifold.

To address these limitations, cotangent Laplacian smoothing is commonly employed. This method is derived from the discrete Laplace-Beltrami operator constructed from cotangent weights and yields an approximation of the mean curvature flow (Desbrun et al., 1999; Hildebrandt & Polthier, 2007). The key idea is to find the location that minimizes the cotangent-weighted Dirichlet energy, i.e., the point at which the discrete Laplacian vanishes and the vertex follows the direction of mean curvature flow (Figure 2.9c). In contrast to simple centroidal smoothing, the cotangent Laplacian operates intrinsically on the surface and therefore preserves curvature more faithfully while still improving mesh quality.



(a) The original configuration of a surface and its mean curvature.

(b) A new vertex is inserted on the linear surface, not respecting the curvature.

(c) Using the tangential Laplacian transform, the vertices are moved along the mean curvature flow vector, better approximating the curve.

Figure 2.9.: Laplacian smoothing can be applied to a vertex after centroidal smoothing to recover its position w.r.t. the mean curvature of the original surface (in red). The figures show a cross-section of a 2-manifold surface.

3. Literature review

The two most permanent fields where meshing is used are in CFD and FEA. Meshing is the discretization of a volume into finite elements. In general, there is a given boundary $\mathcal{M}$, bounding an arbitrary volume $\mathcal{O}$. $\mathcal{M}$ can be represented in several ways. A smooth boundary might be represented by NURBS or Bezier curves, which could capture curvature with technically infinite precision. Another popular method is to represent $\mathcal{M}$ with a PLC. In this thesis, I focus on the second type of representation. Unless otherwise stated, it is henceforth assumed that the boundary is given as a PLC $\mathcal{T}$. Furthermore, I use the phrases *"inside"* and *"outside"* to refer to the bounded and unbounded sides of the boundary $\mathcal{M}$, respectively.

3.1. Mesh generation

An essential characteristic of any CFD mesh is the shape of its volumetric cells, as this governs numerical accuracy, stability, and the computational cost of resolving the flow (López-Pachón & Marcé-Nogué, 2025). Among the many possibilities, three cell families dominate practical CFD workflows: hexahedral, tetrahedral, and polyhedral cells, offering distinct advantages depending on the geometry and the required local resolution.

Hexahedral cells Hexahedra are the preferred cell type in regions where high numerical accuracy and orthogonality are required (López-Pachón & Marcé-Nogué, 2025). Their low skewness and predictable alignment with the flow make them ideal for structured areas of the domain, such as background meshes or boundary-layer layers, where smooth gradients

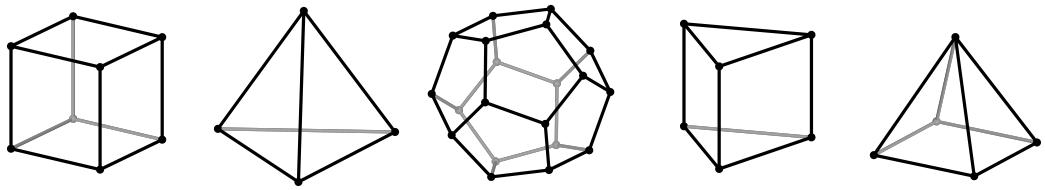


Figure 3.1.: Different cell types used for meshing.

must be resolved reliably. Their main limitation is geometric conformity: the quadrilateral faces of a hex must be planar and remain consistently connected. For this reason, hex-dominant workflows typically begin with a uniform background mesh and rely on progressive refinement, snapping, and local morphing to approximate curved boundaries while preventing face warping (López-Pachón & Marcé-Nogué, 2025).

Tetrahedral cells Tetrahedra are the most flexible elements for filling geometrically complex or organically shaped volumes. They can be generated automatically through Delaunay triangulation and naturally conform to complicated surfaces (López-Pachón & Marcé-Nogué, 2025). This makes them valuable for biological or irregular geometries, where accurate boundary representation is essential. Their drawback lies in numerical efficiency: it requires significantly more tetrahedra to fill a volume with the same vertex density as hexahedra, since each hexahedron can be split into five tetrahedra.

Polyhedral cells Polyhedral cells are generated either through direct polyhedral meshing or by converting from tetrahedral or hexahedral meshes. Because polyhedra possess many faces, they typically produce smoother gradients and better flux balances, resulting in improved numerical accuracy per degree of freedom (López-Pachón & Marcé-Nogué, 2025). They also adapt well to curved or irregular shapes, making them a powerful compromise between the efficiency of hexes and the flexibility of tets. Care must be taken to avoid generating concave polyhedra, which degrade solution quality. Still, well-formed polyhedral meshes offer excellent conformity and require fewer elements than an equivalent tet mesh (López-Pachón & Marcé-Nogué, 2025).

Other cell types Prisms and pyramids appear primarily as transition elements, for example, in boundary-layer inflation (prisms) or between hex and tet regions (pyramids). Their role is secondary but crucial for smoothly connecting different mesh zones or resolving thin near-wall layers (López-Pachón & Marcé-Nogué, 2025).

3.1.1. Refinement and Conformity

In practice, CFD meshes are built through iterative refinement steps that increase resolution only where needed while preserving cell quality and geometric conformity (Blocken, 2015). A typical workflow begins with a coarse structured background mesh (often hexahedral) onto which successive refinement regions are applied around the object, in its wake, and in areas of expected strong flow gradients. These refinements subdivide the local cell structures, producing finer hexes or transitioning into tet or polyhedral zones depending on the meshing strategy (López-Pachón & Marcé-Nogué, 2025).

To ensure that the mesh conforms accurately to complex boundaries, surface-snapping or projection procedures adjust refined cells so that their faces lie on or closely approximate the

geometry (López-Pachón & Marcé-Nogué, 2025). This process prevents gaps, warped faces, or non-planar cell interfaces. When tetrahedral or polyhedral cells are used to fill regions where hexes cannot conform adequately, additional refinement near the surface ensures that the mesh captures curvature and avoids poor-quality or overly stretched elements (López-Pachón & Marcé-Nogué, 2025).

Through this combination of targeted refinement and careful maintenance of element conformity, modern CFD meshes can, in theory, achieve the dual goals of geometric fidelity and computational efficiency, ensuring that the discretized control volumes accurately support the numerical solution of the governing equations. In practice, however, such high-fidelity results are rarely achieved, especially within a reasonable computation time.

3.1.2. 3D Voronoi tesselation as polyhedral mesh

Given a 3D Voronoi tessellation of a set of points, each finite cell is defined as a convex polyhedron (Section 2.2). A valuable property of the Voronoi structure is that each face is perpendicular to the line segment between the corresponding pair of sites, which yields an inherent orthogonality similar to that of structured hexahedral meshes (López-Pachón & Marcé-Nogué, 2025).

A Voronoi tessellation can be constructed simply by computing the Delaunay triangulation (DT) of the points and taking its dual, the Voronoi diagram (VD). The point set may be generated randomly or in a structured manner. To ensure conformity to a bounded domain, they clip the unconstrained Voronoi cells against the boundary surface (Yan et al., 2013). While conceptually straightforward, clipping introduces several practical challenges: the geometric complexity of intersecting each Voronoi cell with a non-convex domain scales poorly, and the resulting boundary cells may contain slivers, concave faces, or drastically varying sizes (Ebeida & Mitchell, 2012; Yan et al., 2013). Such artifacts degrade mesh quality and reduce numerical stability, motivating more advanced approaches that couple point placement, refinement, and boundary conformity more tightly.

3.2. Voronoi mesh generation without clipping

By the nature of Voronoi diagrams, one can place Voronoi sites so that the resulting cells conform directly to a polygonal boundary $\mathcal{M}$. As illustrated in Figure 3.2a, positioning sites on both sides of $\mathcal{M}$ ensures that all cells inside it are finite and that the cell faces adjacent to $\mathcal{M}$ coincide with it. Consequently, all infinite cells lie outside the convex hull of $\mathcal{M}$. During construction, only the interior cells need to be generated; being finite, they require no clipping, while the exterior infinite cells can be ignored entirely. The boundary itself is recovered as the shared faces between interior and exterior cells.

An earlier approach by Merland et al. (2014) discusses an iterative method to generate a Voronoi mesh that conforms to 3D structural features. They randomly scatter a user-specified number of Voronoi seeds within the meshing volume and iteratively optimize their positions by minimizing a composite objective function. The objective function consists of two terms. A CVT (centroidal Voronoi tessellation) term F_{CVT}, which improves cell quality by driving seeds toward their mass centroids and maximizing cell compactness; and a conformity term F_{Vout}, which penalizes the volume of each Voronoi cell that lies on the "wrong side" of a structural feature. The conformity term uses a local planar approximation of the feature to compute an isolated outer volume inside each cell. Minimizing this term encourages Voronoi facets to align with surfaces.

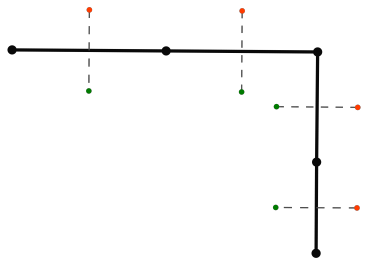
They demonstrate that this approach can produce Voronoi grids that conform well to embedded 3D structural features such as faults, horizons, and pinch-outs, and the resulting meshes show quantitatively small conformity errors. However, the method has several shortcomings. The conformity function is not guaranteed to converge. Exact conformity is not always achievable, especially at pinch-outs or in highly curved regions. This is because the compactness and conformity terms in these cases are competing with each other. The result also depends on the number of seeds: too few seeds prevent a conforming mesh from forming. Additional local post-processing is required when exact geometric matching is needed. They achieve surface-conforming behavior indirectly through the objective function rather than by explicitly constructing surface seeds. In the following subsections, I investigate an approach to derive the surface seeds directly from a refined surface PLC representation of a boundary.

3.2.1. A naive approach with mirroring

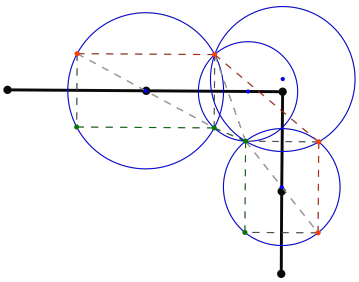
A simple idea is to randomly generate points on one side of the surface and mirror them across the surface. For flat surfaces, this works well: mirroring guarantees that each pair of seeds is equidistant from the surface, so the surface appears naturally as a set of Voronoi cell boundaries. However, as illustrated in Figure 3.2, the situation becomes problematic near corners or on curved patches, where the generated seeds may no longer form correct mirror pairs.

Consider seeds s, t and their mirror images $\hat{s}$, $\hat{t}$; then define vectors $\mathbf{v}_s = s - \hat{s}$ and $\mathbf{v}_t = t - \hat{t}$. If $\mathbf{v}_s$ and $\mathbf{v}_t$ are parallel, then the four seed points describe an isosceles trapezoid (Figure 3.2d). As discussed in Sections 2.1 and 2.2, an isosceles trapezoidal configuration of Voronoi sites defines a single surface Voronoi vertex. In such a configuration, the surface is parallel to the perpendicular bisectors of the two parallel edges of the isosceles trapezoid. Since the circumcenter is defined as the common intersection point of the perpendicular bisectors of its edges, and these bisectors are orthogonal to the surface and aligned with its normal direction, the circumcenter must lie on the surface.

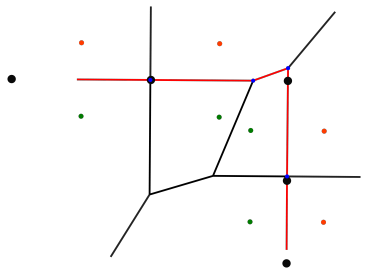
However, if the vectors are not parallel, the two parallel edges are no longer present, and the four points (more likely than not) describe a general polygon that has a unique Delaunay



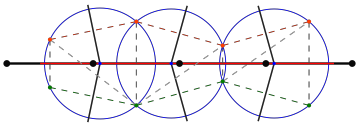
(a) The mirroring approach generates points on one side of the boundary and mirrors them to the plane of the boundary to obtain their pair. This way the boundary is equally distant from both points which is a property of Voronoi diagrams.



(b) However, issues arise when looking into the Delaunay triangulation of the points. At the corner, this configuration yields two unique circumcircles.



(c) Therefore, the resulting Voronoi diagram will have an additional circumcenter that does not lie on the boundary. In the constructed mesh, it becomes a "pinch out". The black dot shows where the vertices should be, and the blue dot shows where they actually are.



(d) The mirroring approach only works trivially on a flat surface where adjacent pairs always form isosceles trapezoids. Even in this case, if the points are not offset equally far from the boundary, the resulting Voronoi vertices will not align with the original vertices, which is undesirable.

Figure 3.2.: 2D examples for the mirroring approach and its drawbacks that can be generalized for higher dimensions. The orange and green dots are the sampled seeds using the mirroring, where orange indicates the original sample and green its mirror image. The blue circles illustrate the circumcircles of the triangles in the Delaunay triangulation of the sites. The circumcenters of the circles are the blue dots. The thicker black line denotes the original boundary. The thin black lines indicate the boundaries of the Voronoi cells. Edges of the Voronoi cells that should conform to the boundary are marked in red, while *infinite* edges are in grey.

triangulation. This means the seeds form triplets that define two unique circumcenters, which both turn to vertices in the Voronoi diagram. Thus, the surface is not reconstructed correctly. Therefore, it is not enough to ensure equivalent distances between seeds and the surface, but the seeds must be equidistant from the vertices of the surface as well.

3.2.2. Boundary conforming Voronoi mesh

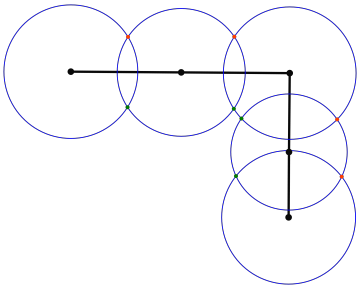
In their paper, Abdelkader et al. (2018) discuss a different approach of mirroring to mesh volumes bounded by a *smooth* mathematical boundary $\mathcal{M}$. They describe a methodology to recover $\hat{\mathcal{M}}$, an approximation of $\mathcal{M}$ from a set of sample points $\mathcal{P}$ on the boundary $\mathcal{M}$. They consider a triplet of intersecting spheres b_x, b_y, b_z. The intersection of the three spheres defines *two seed points*, s_1 and s_2, which will become the Voronoi sites in the final mesh. Also, the sphere centroids p_x, p_y, p_z describe a triangular facet T_{xyz}. Since each seed point lies on the surface of each sphere, its distance to the corresponding sphere center equals that sphere's radius

$$\|p_x - s_1\| = \|p_x - s_2\| = r_x, \quad \|p_y - s_1\| = \|p_y - s_2\| = r_z, \quad \|p_z - s_1\| = \|p_z - s_2\| = r_z.$$

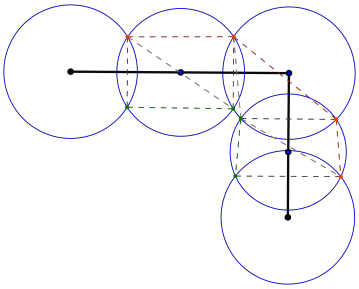
Therefore, the three centers p_x, p_y, p_z all lie in the perpendicular bisector plane of the segment $s_1 s_2$, so the seed points s_1 and s_2 are located at equal distance from the plane of T_{xyz}, but on opposite sides. Furthermore, each sphere b_i generates seeds that lie on the surface of b_i, meaning the centroid of the sphere p_i must be equidistant from all seeds it generates. If b_i does not have any seeds on its bounded side, the seeds it generates are also the closest points to p_i. As discussed in Section 2.2, Voronoi vertices are points equidistant from four or more Voronoi sites. This means p_i must be present in the VD generated by the surface seeds as a vertex. However, it is important to emphasize here that this is only true if the sphere b_i does not have any seeds lying on its bounded side. Figure 3.3 illustrates the generation of seed points in 2D following this approach.

Sampling conditions

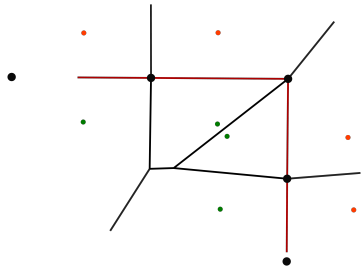
The key challenge in this type of meshing is determining an appropriate number of boundary points, spaced correctly, to ensure that sphere intersections are computed properly. If this is not achieved, some spheres cannot be initialized with a radius that allows them to intersect with neighboring spheres, or they have too large a radius to generate enough Voronoi sites to capture the curvature of the surface properly. Abdelkader et al. (2018) state that, for this methodology to work, the sample points $\mathcal{P}$ must form an *epsilon sample* (Amenta et al., 1999). This means that for each point $x \in \mathcal{M}$, there exists a sample point $p \in \mathcal{P}$ such that the distance $\|x - p\| \leq \epsilon \cdot a$, where a is the minimum distance from x to the medial axis of $\mathcal{M}$ (Amenta et al., 2001). If the sample $\mathcal{P}$ is an *epsilon sample* and the sphere radii are chosen appropriately, the spheres form a continuous overlap along $\mathcal{M}$, completely covering it. The corner points of $\mathcal{U}$ then coincide with the Voronoi sites that generate cells whose boundaries conform to the approximated surface $\hat{\mathcal{M}}$.



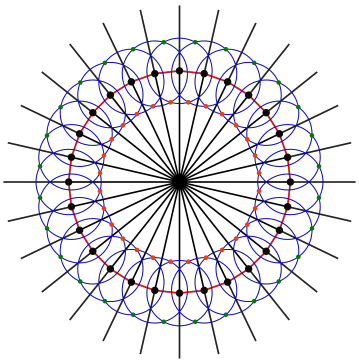
(a) The boundary conforming Voronoi seeds are generated using the intersection points of circles (or spheres in 3D) to initialize the seed points.



(b) This way, the circles used for initialization are the circumcircles of the triangles in the Delaunay triangulation of the generated seeds.



(c) Since the seeds at corners were generated using a single circle centered at the corner vertex, it is now a circumcenter and thus present in the resulting Voronoi diagram as a vertex as well.



(d) This method can recover features with arbitrary angles. The difficult part is finding an appropriate sampling of centroids and radii for the circles (or spheres).

Figure 3.3.: 2D example of an approach which ensures the circumcenters recover the original vertices in the generated Voronoi diagram, by defining the seeds as the intersection of the circles centered on the vertices. The orange and green dots are the sampled seeds on different sides of the boundary. The blue circles illustrate the circumcircles of the triangles in the Delaunay triangulation of the sites. The circumcenters of the circles are the blue dots. The thicker black line denotes the original boundary. The thin black lines indicate the boundaries of the Voronoi cells. Edges of the Voronoi cells that should conform to the boundary are marked in red, while *infinite* edges are in grey.

Given such a sample $\mathcal{P}$, a weight w_i is assigned to each point $p_i \in \mathcal{P}$. This can be interpreted as defining a sphere b_i, centered at p_i, with radius $r_i = \sqrt{w_i}$. The original paper denotes the collection of these spheres as $\mathcal{B}$ and defines their union as $\mathcal{U} = \cup\mathcal{B}$.

A related line of work

Berge et al. (2019) investigate conformity conditions for constructing Voronoi meshes that accurately recover lower-dimensional geometric objects embedded in higher-dimensional domains. Similar to the use of weighted spheres in Abdelkader et al. (2018), they introduce spherical witness regions around vertices, edges, and faces of the target feature and derive necessary and sufficient emptiness conditions that guarantee the inclusion of these entities as Voronoi faces in the final Voronoi diagram. Important distinction from Abdelkader et al. (2018) here is that they clip the Voronoi cells on the domain boundary. Their methodology is explicitly tailored for geological structures, where conformity must be ensured for the geological features, not the domain boundary. By duplicating sites across the target surface and ensuring that all associated witness spheres remain empty, they align the Voronoi bisectors exactly with the geological feature, thereby recovering it as a boundary-aligned subset of the mesh.

3.2.3. Deriving the correct sampling conditions for piecewise linear complexes

Abdelkader et al. (2020) introduces the VoroCrust algorithm, which discusses a more practical approach to Voronoi meshing without clipping. In this paper, they focus on how the appropriate union of spheres $\mathcal{U}$ can be constructed for manifold and non-manifold boundaries with sharp features represented by a PLC $\mathcal{T}$. They describe the process using Maximal Poisson Disk Sampling (MPS) (Ebeida et al., 2012; Guo et al., 2015; Yan & Wonka, 2013) to derive a set of sample points with weights $e_i \in \mathcal{P}$ on $\mathcal{T}$ which forms the basis of the union of spheres $\mathcal{U}$. They implement a recursive MPS (RMPS) algorithm that iteratively refines the surface mesh.

The MPS process is defined on a domain composed of facets and sharp edges of $\mathcal{T}$, which derives the set of sample points $\mathcal{P}$ by iteratively sampling new points on the surface of PLC $\mathcal{T}$ without placing samples unnecessarily close to each other (Abdelkader et al., 2018). A new sample point p_i is only accepted into $\mathcal{P}$ if p_i is not *deeply covered* (see C4 below) by any sphere b_i in $\mathcal{B}$. In this case, p_i is inserted into $\mathcal{P}$; otherwise, the sample is discarded. With the generation of each sample point p_i, they concurrently initialize the respective sphere b_i with radius r_i. To determine the maximal possible radius for a sphere Abdelkader et al. (2020) exposes a user sizing parameter sz and enforces the following four sphere conditions.

(C1) **Smooth coverage:** Given all point intersection of a sphere b_i and the surface $\mathcal{T}$, each intersection point $x \in b_i \cap \mathcal{T}$ can be reached from p_i without crossing a sharp feature (i.e. x and p_i are *co-smooth*) (Figure 3.4a).

(C2) **Smooth overlaps:** Given two spheres that overlap, there must be a path between the two sphere centroids that does not cross any sharp feature. (Figure 3.4a)

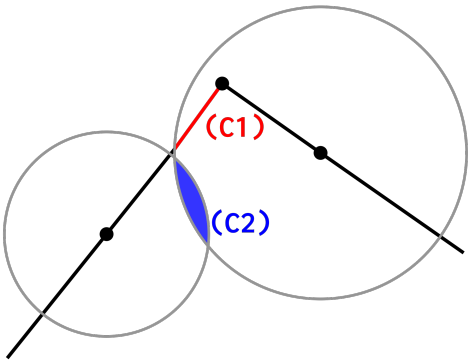
(C3) **Local $\mathbf{L}$-Lipschitzness:** The radii of any two spheres in $\mathcal{B}$ cannot differ more than $\mathbf{L}$ times the distance between the two points. This is enforced to ensure that the sphere does not fully cover a nearby sphere, so that their intersection is empty (Figure 3.4c). The value of $L \in [0, 1)$ controls the allowed amount of overlap.

(C4) **Deep coverage:** A point $x \in \mathcal{T}$ is α deeply covered by a sphere b_i with r_i, if x is also on the bounded side or on the boundary of a smaller sphere centered at p_i with radius $\hat{r_i} = (1 - \alpha) \cdot r_i$, with some constant $\alpha \in (0, 1]$. (Figure 3.4d)

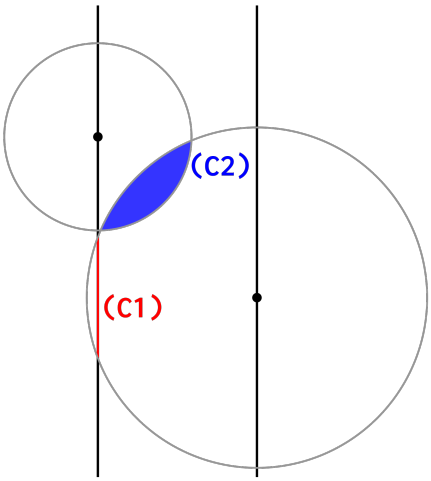
Each sphere condition limits the maximal allowed radius of a sphere. Conditions (C1) and (C2) can be satisfied by finding the closest point $q^* \in \mathcal{T}$ that is not co-smooth with p_i, and limiting $r_i \leq 0.49 \cdot \|q^* - p_i\|$. Violation of (C3) is mitigated by finding the centroid p_c of the nearest sphere b_c with radius r_c and setting the limit $r_i \leq r_c + L \cdot \|p_c - p_i\|$. This constraint does not fully enforce (C3) since there can be a sphere b_q with centroid p_q and radius r_q such that $r_q + L \cdot \|p_q - p_i\| < r_c + L \cdot \|p_c - p_i\|$ even if $\|p_q - p_i\| > \|p_c - p_i\|$, when $r_q + L \cdot \big(\|p_q - p_i\| - \|p_c - p_i\|\big) < r_c$. Combining the limits with the sizing parameter imposed by sz Abdelkader et al. (2020) sets each sphere radius to $r_i = \min(sz, 0.49 \cdot \|q^* - p_i\|, r_c + L \cdot \|p_c - p_i\|)$ when the it is initialized.

Since the sphere conditions cap the maximum radius of each sphere, the inserted samples leave certain regions of the input $\mathcal{T}$ uncovered. To increase the likelihood that new samples are generated in uncovered areas, after 100 consecutively discarded samples, the faces in $\mathcal{T}$ are split into four by subdividing each edge and connecting the midpoints. Once the faces are split, the deeply covered subaces are discarded from the MPS process and no longer get sampled. This means the pool of faces that MPS samples decreases continuously until each subface is deeply covered, enforcing the sphere condition (C4).

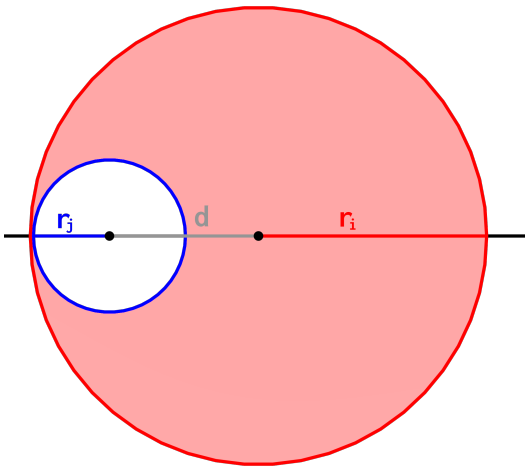
To protect sharp features, Abdelkader et al. (2020) recursively applies the MPS process in three phases. In the first two phases, they initialize the spheres to protect the corners and sharp edges, and in the third phase, the facets of $\mathcal{T}$ are sampled. This way, the spheres of protected features influence the radii of later-inserted surface spheres, not vice versa. After each phase is completed, the MPS applies a *sphere shrinking* step to uncover half-covered seeds (Abdelkader et al., 2018). Whenever a sphere from any previous phase gets shrunk, the MPS process recurses to that phase (e.g. if a corner sphere, to phase one; if an edge sphere, then to phase two) and reassigns the sphere radii to maintain their protection.



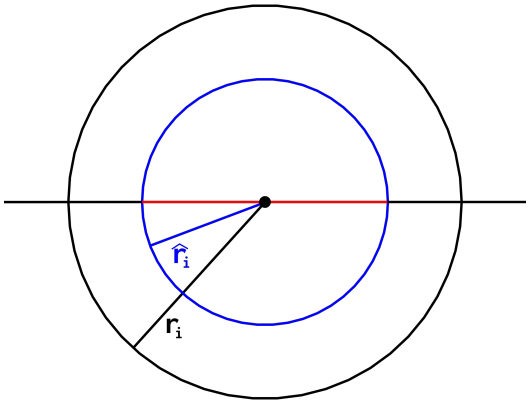
(a) (C1): Spheres cannot cover non-co-smooth surfaces (red). (C2): spheres of non-co-smooth patches are not allowed to overlap (blue).



(b) Another configuration when the (C1) and (C2) conditions are violated.



(c) (C3): If there are two spheres with radii r_i, r_j, and the distance between their circumcenters is d, then $r_i > r_j + d$, one sphere completely covers the other, producing no intersections.



(d) (C4): Parts of the surface that are covered by a sphere $\hat{r_i} = \alpha \cdot (1 - r_i)$ are called α *deeply covered* (red) by the concentrical sphere with radius r_i.

Figure 3.4.: Illustration of the four conditions enforced on the spheres.

Surface seeds & sphere shrinking

Once an iteration of the MPS process terminates, the input PLC is deeply covered by the spheres in $\mathcal{B}$. Based on Abdelkader et al. (2018), the surface seeds can be placed at the intersection point of a triplet of overlapping spheres. However, to ensure the proper sample conditions, all intersection points must remain uncovered by any fourth sphere. Abdelkader et al. (2020) identifies all overlapping sphere triplets and computes their intersection points. Then if an intersection point is covered by a *fourth sphere*, they consider the three spheres that generate the seed and the covering sphere in isolation. Such a sphere quartet produces 4 *half-covered seeds* in total as illustrated in Figure 3.5. To uncover all four half-covered seeds, it is enough to shrink one sphere. They choose the sphere that requires the *least shrinkage*, i.e., the sphere that needs to decrease its radius the least. They define the shrinkage as the difference between the old and the new radii.

This step can reduce the coverage of $\mathcal{B}$ on $\mathcal{T}$, thus requiring the insertion of new samples. Therefore, after shrinking the RMPS process must evaluate if all previously enforced conditions still hold or not, and rerun the MPS stage accordingly. With the increase in the sample density, the shrinkage of the spheres decreases, converging to a termination state.

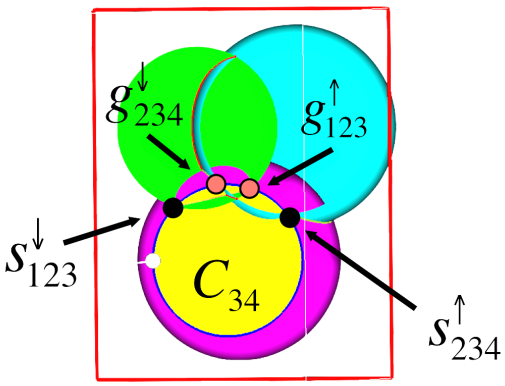


Figure 3.5.: If a fourth sphere covers a seed (generated by a triplet of overlapping spheres), then there are four covered intersection points in total. To prevent this, the sphere requiring the least shrinkage gets shrunk. The shrinking ensures the termination without slivers. (Illustration adapted from Abdelkader et al. (2018).)

3.2.4. Interior seeds

Once the final size of each sphere is reached and the surface of $\mathcal{T}$ is deeply covered by spheres, and $\mathcal{U}$ is refined appropriately to start generating surface seeds. The VoroCrust algorithm, as discussed previously, generates surface seeds by computing the intersection points of spheres in overlapping sphere triplets. To initialize the interior seeds Abdelkader et al. (2020) uses the *spoke-dart algorithm* (Mitchell et al., 2018) to generate weighted interior

seeds. They enforce the sphere condition (C3) on the interior spheres to ensure a gradual transition between cell sizes. Furthermore, they discuss the possibility of using a structured lattice in which the interior seeds are initialized as a grid. This yields a hex-dominant mesh that has Voronoi cells only near the boundary.

3.2.5. Available implementations of the VoroCrust algorithm

A prototype implementation of Vorocrust is available at github.com under sandialabs/vorocrust-meshing, and another at lanl/VoroCrust, the latter being tailored for geological applications. Both implementations rely on the Voronoi approach discussed above. They serve as proof of the possibility of generating a boundary conforming Voronoi mesh without clipping.

However, these implementations were developed for contexts different from urban flow simulation. They are appropriate for meshing B-reps representing smooth geometries and well-suited for finite element analysis, where individual elements must be of high quality. In geological settings, they aim to conform to fault lines and other intersecting surfaces (LaForce et al., 2023). This means they are designed to handle intersecting surfaces and non-manifold domains. However, the boundary information that is crucial for urban flow simulation is not preserved by their implementation.

4. Methodology

In this chapter, I describe the methodology for meshing a volume $\mathcal{O}$, represented by an abstract boundary $\mathcal{M}$ and instantiated as a PLC $\mathcal{T}$, using a Voronoi cell structure. My methodology is tailored to the output of City4CFD, which provides the domain envelope as a triangulated surface, with triangles grouped by surface patch types (see Section 2.4). I explicitly use this group information during processing. However, I assume the geometry is a 2-manifold, meaning it has no self-intersections or overlapping faces. This is not always true for City4CFD output, but I made this assumption due to the time constraints of this thesis.

My methodology aims to derive a polyhedral mesh with adaptive cell sizing, such that the mesh conforms to the input boundary without clipping. I generate the polyhedral cells from a Voronoi diagram, which makes this goal feasible (Berge et al., 2019). Voronoi sites are obtained by intersecting sphere triplets. As described by Abdelkader et al. (2018), if the union of spheres covers the entire input boundary, this ensures that the triangles formed by sphere triplets are present in the resulting Voronoi diagram, meaning the mesh conforms to the boundary. This is why the resulting Voronoi mesh has triangular faces on its boundary.

To derive a set of spheres that satisfies the necessary conditions, I apply an iterative refinement process to the PLC representation of the boundary. During this process, I insert new vertices into the PLC and split faces to maintain a triangulated surface. I then apply centroidal smoothing in flat regions and mean-curvature-based smoothing in curved regions to regularize the triangles. The refinement is guided by the sphere conditions introduced by Abdelkader et al. (2020). I initialize sphere radii such that they satisfy these conditions, which ensures that spheres become appropriately small in regions where non-adjacent parts of the boundary are close. If a sphere radius becomes too small relative to the local vertex density, the three spheres associated with a triangle may fail to intersect. In that case, I split the triangle into three by inserting a midpoint, increasing the local vertex density. This allows the triangulation to adapt its face size to the local feature size of the bounding geometry.

To fill the interior of the volume, I propose approaches based on either uniformly distributed seeds or a grid of seeds. Since cell sizes correlate with the local density of Voronoi sites in the Voronoi diagram, I also propose a method to locally increase site density via iterative uniform random sampling and pruning. This inserts progressively more seeds closer to the boundary, producing a smooth transition from dense boundary regions to sparser interior regions. In what follows, I explain the underlying rationale for my methodology in detail.

4.1. Input & Preprocessing

The meshing domain is defined by the input $\mathcal{T}$. An example of the input is illustrated in Figure 1.2. $\mathcal{T}$ consists solely of simplices (triangular facets) and represents a single manifold boundary $\mathcal{M}$ without holes, self-intersections, or overlapping faces. Consequently, every edge has exactly two incident facets, and each face σ in $\mathcal{T}$ is assigned to a patch group $\mathscr{P}(f_i)$.

I maintain two representations of the surface throughout the entire process: a halfedge data structure $\mathcal{H}$ and a k-d tree $\mathcal{K}$. Both are constructed from the input before refinement and are updated continuously. The halfedge structure encodes connectivity, whereas the k-d tree accelerates spatial queries on vertices. Whenever a new point is sampled on $\mathcal{T}$, it is inserted into both structures. Each point is assigned a unique index to maintain a one-to-one correspondence between the two data structures. Because the k-d tree supports spatial searches, it must be rebuilt whenever vertex positions change. In contrast, $\mathcal{H}$ requires only local updates, since global connectivity is unaffected by insertions; however, its local triangulation must remain Delaunay.

For the moment, I assume that normal vectors $\mathbf{n}_f$ and $\mathbf{n}_p$ are available for every face and every point of the PLC $\mathcal{T}$. Their computation is detailed later in Subsection 5.1.1, as part of the implementation discussion. In the following, I denote normals as $N(x)$, where x refers to either a point p_i or a face f_i, with $p_i, f_i \in \mathcal{H}$. The normal of a face is the vector perpendicular to the plane of the corresponding facet, and a point normal is defined as the normalized average of the normals of all incident faces.

4.1.1. Identifying protected features

I define the set of protected edges $\mathcal{H}_{PE}$, the set of protected vertices $\mathcal{H}_{PV}$, and the set $\mathcal{H}_{SV}$ containing all vertices having at least one incident protected edge. I identify sharp edges using the dihedral angles between incident facets of an edge as described by Abdelkader et al. (2020) and discussed in Section 2.5.1 and collect them into $\mathcal{H}_{PE}$. However, since I aim to preserve the boundaries between different surface patch types, I need to identify additional edges and corners as protected. I also consider an edge e_i protected if the two faces incident to it (f_r, f_l) belong to different patch groups: $e_i \in \mathcal{H}_{PE} \quad iff \quad (f_r \in \mathscr{P}j, f_l \in \mathscr{P}k, \quad and \quad \mathscr{P}_j \neq \mathscr{P}_k) \quad or \quad N(f_r) \cdot N(f_l) < \theta$.

Further, I collect the protected vertices to $\mathcal{H}_{PV}$. A vertex is protected either if it is incident to 3 or more protected edges, or the vertex is incident to two protected edges that enclose an angle "smaller" than the sharp angle threshold. The key difference here is that the set of sharp edges is a subset of the set of protected edges; therefore, the set of protected vertices $\mathcal{H}_{PV}$ can be larger than the set of corner vertices.

4.1.2. Configurable Parameters

To make meshing more flexible, I expose some parameters for user configuration.

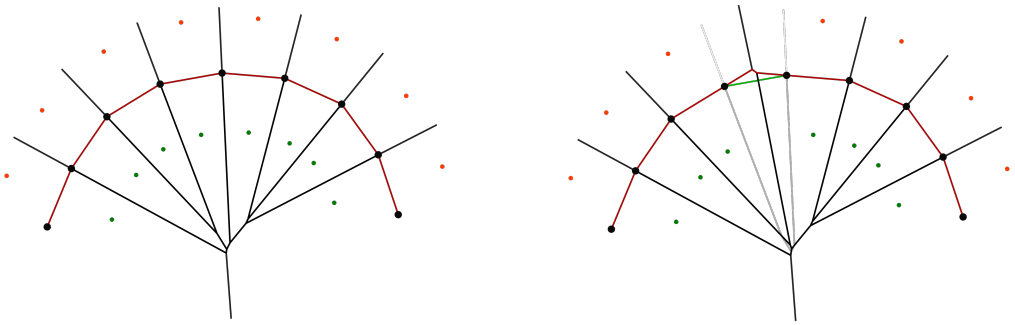
- Sharp angle threshold: controls what dihedral angle is considered sharp. It can be configured for each surface patch type distinctly to allow greater control.
- Sizing parameter: limits the maximum area of the triangles allowed in the triangulation. Different values may be specified for different patch types.
- Lipschit constant L: a value in the range $[0, 1)$. Using $L > 1$ allows a sphere to cover another sphere completely. Using $L = 0$ limits the radii of all spheres to be equal to the distance to their closest neighbor maximum.
- Iteration limits: there is a control parameter exposed for each iterative process in my methodology which controls the maximum number of allowed steps for it to take.

4.2. Generating surface seeds: Triangulation refinement

My first goal is to compute the Voronoi sites that protect the boundary represented by a triangulation of the input surface $\mathcal{T}$ where each triangle vertex p_i has a weight corresponding to the radius of the sphere b_i placed on p_i. I achieve this through an iterative refinement process similar to Abdelkader et al. (2020), but with key differences in how points are sampled on $\mathcal{T}$ and when sphere radii are computed. These changes are made because I impose stricter conditions on the input than they do: I expect a single manifold boundary as input. Since there are no intersections in the geometry, I do not need to protect them as in Abdelkader et al. (2020), allowing me to directly refine the input PLC $\mathcal{T}$ without identifying and protecting such intersections.

The refinement is an iterative process. Starting from the input $\mathcal{T}$, I refine $\mathcal{T}$ (i.e. iteratively split triangles by inserting new vertices) until the three spheres on the vertices of each triangle intersect in exactly two points. In each iteration, I *initialize spheres* by computing a radius for each vertex of $\mathcal{H}$ according to the sphere conditions introduced by Abdelkader et al. (2020). Once the spheres are initialized, they are shrunk to uncover half-covered seeds. Then, I determine the intersection of spheres located on the three vertices of a triangular facet in $\mathcal{H}$. If they do not generate two seeds, I *split* the triangle by adding a center vertex, or in the case of an edge, the bisector point (more detail in Subsection 4.2.3). After the new vertices are added, the faces are smoothed slightly to eliminate sliver or near-sliver surface triangles. Finally, I ensure that the triangulation is Delaunay, thereby ending the iteration. The iteration ends when each facet of $\mathcal{H}$ defines a triplet of overlapping spheres or when the maximum number of iterations is reached. The intersection points of sphere triplets defined by the three vertices of each face $f \in \mathcal{H}$ form the set of surface seeds. In the following, I elaborate on each step of the triangulation refinement.

The objective of my refinement is the same as Abdelkader et al. (2020): find the set of spheres (centroids and respective radii) that satisfy the sphere conditions they propose. However, I aim to arrive at this set differently. My methodology relies on the enforcement of sphere conditions (C1) and (C2) during sphere initialization (Subsection 4.2.1) to produce sufficiently small spheres everywhere to reduce the triangle size enough in regions where the boundary curves a lot or are close to another boundary, as illustrated in Figure 3.4b. By having small spheres in such regions, the corresponding triangles are split (Subsection 4.2.3) to increase sphere density until condition (C4) is not satisfied. Furthermore, (C3) implicitly enforces a gradual transition between sphere sizes. When the objective is reached, the entire boundary surface is covered by spheres, and all triangles can generate a corresponding uncovered pair of seeds. In this case, the resulting seeds generate a Voronoi mesh that perfectly conforms to the input boundary everywhere (Abdelkader et al., 2018, 2020).



(a) The surface reconstructs correctly when all seed pairs are present.

(b) If a seed pair is missing, the reconstruction is incorrect locally. (light grey: the correct Voronoi structure)

Figure 4.1.: When a pair of surface seeds is not getting generated, it only influences the resulting Voronoi diagram locally, and it can still conform to the boundary of the majority of the surface.

If the iterative refinement process is stopped before the objective is reached, some triangles on the boundary surface cannot generate Voronoi seed pairs because their corresponding spheres do not intersect. The number of such cases is correlated with the number of iterations performed, as more iterations yield fewer triangles that do not intersect spheres. A useful property of this behavior is that it affects the resulting mesh only locally. Figure 4.1 illustrates how a missing seed influences the mesh. It shows that the absence of a seed pair affects only neighboring cells, due to the local nature of Voronoi diagrams. As a result, if only a relatively small number of triangles fail to generate seeds, the resulting Voronoi diagram still conforms to the original boundary in most regions, with only minor local errors.

This property allows the user to trade accuracy for meshing speed, since the time required to complete a single iteration increases with the number of faces in $\mathcal{H}$.

In the remainder of this section, I detail the steps taken in each iteration.

4.2.1. Initializing spheres

In each iteration, when the spheres are initialized on the vertices by assigning them the maximal possible radius, the sphere conditions Abdelkader et al. (2020) allow for each sphere. More accurately, I enforce conditions (C1) - (C3) by finding the radius for a sphere that does not violate any of them. Further, I ensure the spheres do not cover any vertices beyond their centroids, in alignment with the sampling described by Abdelkader et al. For each sphere b_i with centroid p_i and radius r_i, I ensure the following properties hold.

Preventing deep coverage of vertices by other spheres

It must hold that $(1+\alpha) \cdot \|p_i - p_j\| \geq r_i$ for any vertex $p_j \in \mathcal{T}$ s.t. $p_i \neq p_j$. This ensures none of the vertices in $\mathcal{H}$ are deeply covered by a sphere. This must be the case, since Abdelkader et al. (2020) ensures that their sample points are not deeply covered and that they discard deeply covered seeds. I ensure this condition by finding the closest vertex $p_c \in \mathcal{H}$ using $\mathcal{K}$ for the spatial lookup and setting $r_i = (1+\alpha) \cdot \|p_i - p_c\|$.

Preventing non-co-smooth overlaps

As Abdelkader et al. (2020) describes, a surface sphere b_i should never cover a surface point x that is not co-smooth with p_i, or intersect with a sphere b_j where p_j is not co-smooth with p_i. This can happen in two configurations as illustrated in Figure 3.4a and Figure 3.4b; *(a)* when p_i is *close* to a protected edge, or *(b)*, when the second when two non-adjacent faces are too close to each other. They prevents these violations by finding the *surrogate* point p^*: the closest non-co-smooth point of $\mathcal{T}$ and set $r_i = \min(r_i, 0.49 \cdot \|p_i - q^*\|)$. They are doing it this way because in a later iteration of their MPS process, a sphere might get inserted with centroid q^*, and in such a case, the two spheres should not overlap. Since I recompute the sphere radii whenever new vertices are inserted into $\mathcal{H}$, I only need to ensure that the spheres with non-co-smooth centroids do not overlap in the actual vertex configuration.

To avoid these violations, for each $p_i \in \mathcal{H}_{RV}$ I find the closest vertex $p_c \in \mathcal{H}_{RV}$ with sphere b_c such that $N(p_i) \cdot N(p_c) > \theta$ and the closest $p_p \in \mathcal{H}_{SV}$. First, I iterate the faces incident to p_c with $N(f) \cdot N(p_i) > \theta$, and try to find the face f, having the smallest orthogonal distance to p_i while having the orthogonal projection $\mathrm{proj}_f(p_i) = p_c^*$ lying inside the triangle defined by the face. No such projection may exist, for example, when the closest point of f is actually p_c. In the absence of an acceptable projection, I use p_c instead of p_c^*. Since

p_p is incident on a protected edge (i.e. the face on the other side is not smooth with p_i), I must ensure b_i does not cover such a protected edge. Therefore, for each incident edge e of p_p where e is protected, I project p_i to e obtaining $\text{proj}_e(p_i) = p_p^*$. As before, I accept p_p^* only if it lies on the corresponding edge. If none of the projections are acceptable, I default back to p_p. To ensure b_i does not cover p_c^* or p_p^* (c1) and does not overlap with b_c (c2), I set its radius to $\min(r_i, \|p_i - p_c^*\|, 0.49 \cdot \|p_i - p_c\|, \|p_i - p_p^*\|)$.

In the case where $p_i \in \mathcal{H}_{SV}$, the vertex is incident to at least two faces whose dihedral angle exceeds the threshold θ. Here, I must apply looser conditions to ensure that the spheres protecting such features obtain a sufficiently large radius (Abdelkader et al., 2020). Consequently, these vertices are ignored during the *(a)* case of Figure 3.4a, and are only processed to prevent violations of the *(b)* case. To this end, I identify the closest non-co-smooth points p_c and p_p and enforce that the sphere at p_i does not cover either of them $r_i = \min(r_i, \|p_i - p_c\|, \|p_i - p_p\|)$. This adjustment may not immediately yield the final radius, but it prevents assigning a radius that is too small. This is permissible because a stricter constraint is applied later during the sphere-shrinking step, which guarantees that all protected spheres are reduced sufficiently so as not to cover any other surface seeds.

Ensuring L - Lipschitzness

The Lipschitzness is enforced in a separate step once all sphere radii is initialized since it needs the information on neighboring sphere radii. For each vertex $p_i \in \mathcal{T}$ with sphere b_i, I find the closest vertex p_c with sphere b_c, and set $r_i = \min(r_i, r_c + L \cdot \|p_i - p_c\|)$ where r_c is the radius of b_c.

4.2.2. Shrinking spheres

Once the spheres are initialized, I must ensure that none of the seed points lie inside any sphere. This step guarantees that the resulting Voronoi cells conform to the surface encoded by $\mathcal{H}$. I follow the procedure of Abdelkader et al. (2020) with a slight modification. Because I explicitly maintain $\mathcal{H}$, I know the exact faces that must be recovered, and therefore I know which sphere intersections correspond to the required surface seeds.

Instead of testing all triplets of overlapping spheres, I restrict the checks to the triplets associated with each triangular face $f_i \in \mathcal{H}$. If the three spheres at the vertices of f_i intersect in exactly two points s_1 and s_2, I test whether a fourth sphere covers either of these seeds. If such coverage occurs, I shrink the entire quartet of spheres as described by Abdelkader et al. (2020) and discussed in Section 3.2.3. I compute the radius for each sphere that would uncover the points, i.e. the radius that places the closest seed on the boundary of the sphere, simply by using $r_{new} = \|p_c - p_s\|$ for the sphere radius, where p_s is the position of the closest seed to p_c, and p_c is the centroid of the sphere covering p_s. I find the new radius for each sphere, but only assign it to the sphere whose difference between

the old and new radii is the smallest. This prevents unnecessary shrinkage of spheres while ensuring each seed is uncovered.

However, shrinking a sphere may cause other seeds to become covered. After a radius update, the corresponding sphere may sample its intersection points at new locations, and one of these points may now fall inside a previously non-covering sphere. To ensure full correctness, sphere shrinking must therefore be performed iteratively. After each update, the configuration is rechecked, and the process continues until an iteration completes without any sphere being shrunk. This fixed-point iteration guarantees that no seed lies inside any sphere in the final configuration.

4.2.3. Subdividing triangles and splitting edges

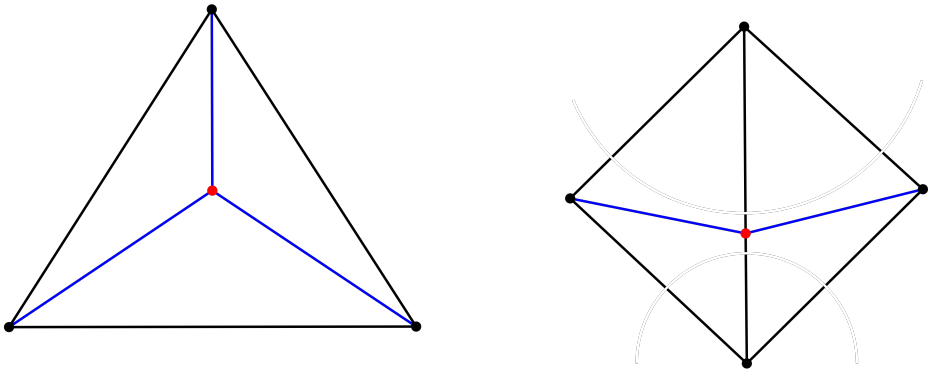
During triangle refinement, a triangular face is selected for subdivision either if its area exceeds the area allowed by the user or if the spheres of the triangle vertices do not intersect in exactly two points. To split the triangle, I insert its centroid as a vertex and connect pairs of its vertices to the midpoint, obtaining three sub-triangles as depicted in Figure 4.2a. I split any protected edge whose endpoint spheres do not overlap. In case of splitting edges, I insert a new vertex at the midpoint of the uncovered portion of the edge, as illustrated in Figure 4.2b. Every time an edge is split, all incident faces are correspondingly subdivided to maintain the triangular structure of the surface representation in $\mathcal{H}$. When a triangle is subdivided, its patch type is propagated to the resulting subfaces. Likewise, when a protected edge is split, the newly created edges inherit the protected status. Furthermore, if the triangle belongs to a curved, smooth patch, the new vertex must be repositioned with respect to the mean curvature flow of the surface.

Together, these two operations increase the local vertex density continuously until it becomes sufficient to guarantee the existence of two intersection points for every triangle in $\mathcal{H}$. Furthermore, to ensure triangles do not exceed the user-specified size parameter, I split triangles whose area exceeds the allowed value for the given patch.

After splitting faces and edges, the surface triangulation may no longer satisfy the Delaunay condition and therefore must be restored. As discussed in Section 2.1, a Delaunay triangulation is guaranteed to exist, and it can be re-established by identifying pairs of triangles in $\mathbb{R}^2$ (or pairs of tetrahedra in $\mathbb{R}^3$) that violate the empty circumcircle (resp. circumsphere) condition. For each violating pair, the shared edge (in 2D) or shared face (in 3D) is flipped to improve local Delaunay conformity, as illustrated in Figure 2.2. By repeatedly applying these flips until no further violations remain, the triangulation (or tetrahedralization) is ensured to be Delaunay once again.

4.2.4. Smoothing step

The smoothing step is crucial in my methodology, as it ensures that the vertices of the surface triangulation are positioned so that the sphere-overlap conditions introduced by



(a) (b)

Figure 4.2.: Illustration of the methodology applied during splitting operations. In red, the inserted vertex, and in blue, the inserted edges. (a) The inserted vertex is placed at the centroid of the triangle, and (b) for edges the inserted point is the midpoint of the uncovered edge section.

Abdelkader et al. (2020) are satisfied. If triangles are only split without redistributing vertex positions, the configuration may never converge to one in which the required overlaps hold, potentially causing indefinite refinement. To prevent this, I introduce a smoothing stage that redistributes vertex positions to satisfy the spherical triplet conditions better while preserving mesh quality. I propose different strategies for flat and curved boundaries.

Flat boundaries

My approach for flat surfaces applies centroidal smoothing to planar surface patches. This procedure distributes vertices uniformly with respect to the local sampling density and improves triangle shape regularity without altering the prescribed density. It is simple, computationally efficient, and well-suited for locally planar regions. By driving the mesh toward configurations in which triangles approximate regular shapes, centroidal smoothing suppresses the formation of sliver elements and enhances the overall quality of the surface triangulation.

In each iteration of the centroidal smoothing, the position of the vertices not incident on any protected edge is updated. Their new position is chosen as the centroid of the 1ring around the vertex (Figure 4.3). Each vertex is updated simultaneously, so one update does not influence the others; hence, it can be done in parallel. By repeating the operation multiple times, the triangulation converges to a state in which the vertices no longer move between iterations; they have reached their optimum. In this state, the distance between the vertex and its neighbors is optimized in the least squares sense.

Curved boundaries

For curved boundaries, planar smoothing strategies are insufficient because they do not preserve the geometric characteristics of the underlying curved surface. I therefore propose using a discrete approximation of mean curvature flow to update vertex positions on the PLC. This corresponds to a constrained variant of Laplacian smoothing that aims to maintain the mean curvature implied by the surface. This method is inspired by Desbrun et al. (1999), who use similar constraints to position the vertices of a surface patch in such a way that best approximates the mean surface curvature.

The mean curvature at a vertex is a second-order quantity influenced by its two-ring neighborhood: the set of adjacent vertices and the neighbors of those vertices (Figure 4.3). Using this two-ring structure, it is possible to construct the *combinatorial Laplace operator* for the local neighborhood, while enforcing positional constraints on the ring vertices to preserve the boundary geometry. The Laplacian then provides a curvature-sensitive direction along which the vertex is updated. In effect, the procedure adjusts each vertex toward the position that best preserves the local mean curvature while regularizing triangle shapes and maintaining consistency with the surrounding curved geometry (Hildebrandt & Polthier, 2007).

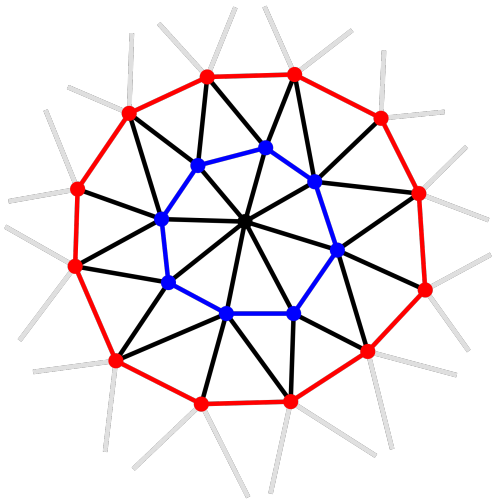


Figure 4.3.: The 1ring (blue) and the 2ring (red) of a middle vertex in an arbitrary surface triangulation.

4.3. Generating interior seeds

There are three ways to generate interior sample points inside the boundary $\mathcal{M}$, represented by $\mathcal{T}$. Each method produces a distinct distribution of interior points. Regardless of the

chosen approach, any sample point that falls inside a sphere of $\mathcal{B}$ must be discarded, as such points would violate the conformity conditions discussed earlier.

4.3.1. Random Scattering

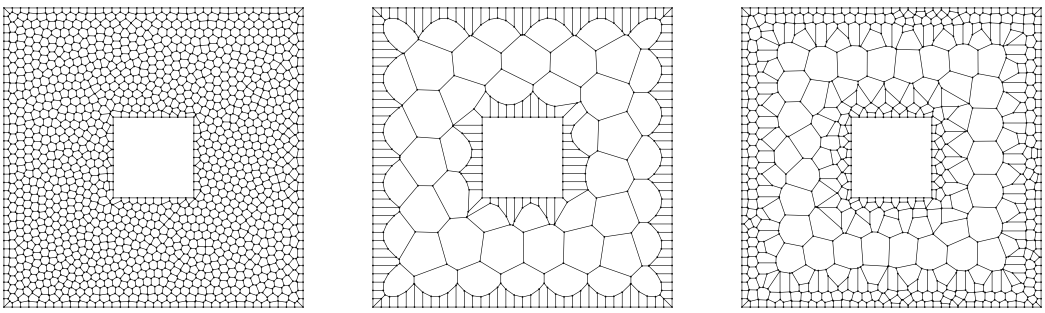
To generate a random set of sample points with a prescribed uniform density (specified in points per unit volume), I proceed as follows. First, I determine the axis-aligned bounding box $(\mathbf{v}_{\min}, \mathbf{v}_{\max})$ of $\mathcal{T}$ and compute its volume,

$$V_{\text{bbox}} = (\mathbf{v}_{\max,x} - \mathbf{v}_{\min,x})\,(\mathbf{v}_{\max,y} - \mathbf{v}_{\min,y})\,(\mathbf{v}_{\max,z} - \mathbf{v}_{\min,z}).$$

Given the target density ρ (points per unit volume), the number of required sample points is then $N = \rho\, V_{\text{bbox}}$. I draw N points independently and uniformly from the bounding box by sampling each coordinate from a uniform distribution on its corresponding interval:

$$x \sim U(\mathbf{v}_{\min,x},\, \mathbf{v}_{\max,x}), \quad y \sim U(\mathbf{v}_{\min,y},\, \mathbf{v}_{\max,y}), \quad z \sim U(\mathbf{v}_{\min,z},\, \mathbf{v}_{\max,z}).$$

This produces a set of random sample points with the desired uniform volumetric density inside the bounding box enclosing $\mathcal{T}$. Points outside the geometric domain can subsequently be removed using a point-in-domain test, which I perform in a later step. The inherent nature of Voronoi diagrams ensures connectivity between the interior and exterior samples (Figure 4.4a). However, if there is a large discrepancy in sample density, the connection between the two regions might be distorted by long cells (Figure 4.4b).



(a) (b) (c)

Figure 4.4.: Conforming Voronoi diagram generated in 2D using my methodology. The surface seeds are generated using sphere intersections, and the interior samples are generated randomly at different densities (a,b). In (c), the density is scaled based on distance to surface seeds.

4.3.2. Adaptive density

To achieve adaptive and smoothly varying cell density in the final Voronoi mesh, I investigated two strategies.

The first approach, described by Abdelkader et al. (2020), extends the dart-spoke sampling paradigm. Similar to surface sampling, it uses local sphere constraints to regulate sampling density and ensure smooth transitions between regions with different resolutions. In this formulation, the weights assigned to the seeds are interpreted as spheres in a weighted Delaunay tetrahedralization constructed over the combined set of surface and interior points. These spheres define exclusion regions in which no additional samples are accepted, and their radii are scaled according to a Lipschitz continuity constraint on the sizing field. As a result, the final seed distribution naturally adapts to geometric complexity and user-prescribed density variations.

The second strategy follows a simpler, distance-based refinement scheme. The initial step is identical to that of a uniform random Voronoi grid, where n seeds are randomly distributed throughout the volume. In an iterative process, additional random seeds are sampled from the volume with a given density d. Samples that are farther from a surface seed than a specified distance threshold are removed. By continuously decreasing this distance threshold between iterations, only points increasingly close to the surfaces are retained, increasing seed density near boundaries and producing a gradual transition between dense and sparse regions. The sampling density d should be adjusted based on the number of iterations, with more iterations requiring a lower density. This approach is illustrated in Figure 4.4c.

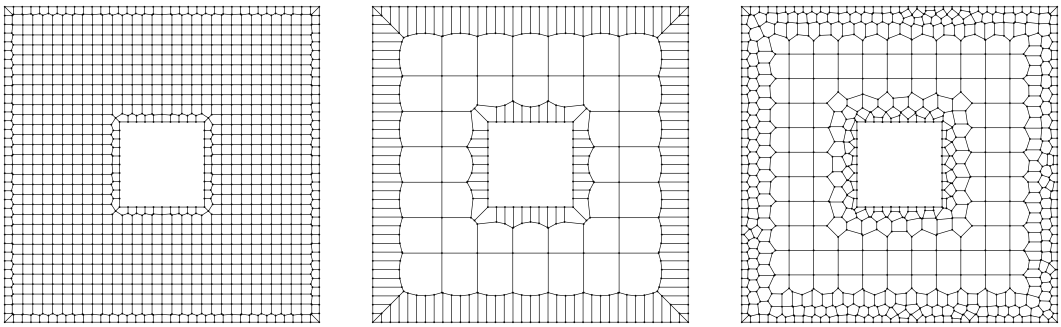
4.3.3. Structured lattice

A structured background mesh can be generated by sampling points at uniform spacing, producing a regular 3D grid. Triangulating these interior points yields overlapping circumcenters that collapse, so the dual of a uniformly spaced point set naturally forms hexahedral cells whose aspect ratio is determined directly by the grid spacing. The hexahedral core connects seamlessly to the Voronoi cells induced by the surface seeds, a consequence of the Voronoi diagram's inherent continuity (Figure 4.5a). However, the same difficulties encountered with randomly distributed points also occur here: a significant mismatch between the interior and surface seed densities leads to poor transitions and distorted cells (Figure 4.5b).

This issue can be mitigated by inserting additional points randomly between the grid lattice and the surface seeds, resulting in a gradual transition, as illustrated in Figure 4.5c. A simple way to achieve this is to generate the Voronoi density transition in the same manner as described for adaptive mesh density, but without performing the initial step of scattering n random points. This produces a gradual transition region without fully populating the interior. Afterward, a uniform, equally spaced grid of points is generated, and points that are too close to existing seeds, as well as points lying outside the boundary, are removed. Once this is done, regions far from the boundary are meshed with hexahedral cells, while near boundary features the polyhedral cells conform well to the geometric details.

For urban CFD, this approach is the most suitable, as it produces a mesh that conforms well to geometric features while allowing simpler computations in large portions of the domain.

The Voronoi grid provides boundary conformity and a natural transition between dense and sparse regions. Generating larger cells farther from boundaries further simplifies the mesh, and constructing hexahedral cells enables the use of simpler forms of the Navier–Stokes equations. Together, these aspects help reduce simulation time in CFD, an effect that becomes increasingly important as the domain volume grows. Since urban CFD is typically performed on large domains, this is a critical consideration.



(a) (b) (c)

Figure 4.5.: Conforming Voronoi diagram of points where the interior samples were sampled along a grid lattice. (a) and (b) showcases cells with uniform sizing but with different interior cell densities, and (c) illustrates adaptive cell density between interior and surface cells.

4.4. Engineering decisions: Deviations from the VoroCrust algorithm

An idea was to modify the existing implementation of the VoroCrust prototype. However, without access to proper source documentation, completing such modifications within the time constraints of this thesis proved impractical. By instead relying on Computational Geometry Algorithm Library (CGAL) and its well-documented Application Programming Interface (API), the resulting codebase becomes easier for others to reuse, extend, integrate, and adapt in future work.

Given this decision, I investigated whether CGAL provides functionality equivalent to the MPS process described by Abdelkader et al. (2020). The only comparable tool available is Poisson surface reconstruction, which generates a surface from a point cloud. However, since this method does not preserve boundary information, it is unsuitable for my purposes.

The conditions for obtaining a conforming Voronoi mesh require a configuration of spheres that overlap correctly to form intersecting triplets, each of which produces two valid seeds. Satisfying these conditions depends critically on the placement of the vertices: they must be arranged such that spheres meeting the criteria can be constructed. By distributing

vertices through centroidal smoothing, I aim to obtain a triangulation with near-uniform element quality, and by inserting additional points iteratively, I seek to achieve the vertex density necessary for the sampling conditions to hold.

5. Implementation, Experiments, and Results

As a proof of concept for my methodology, I implemented a prototype application in *C++*. In this chapter, I discuss the tools I used to realize the method in the previous chapter. My prototype partially implements my methodology and demonstrates that a boundary-conforming Voronoi mesh can be generated for 2-manifold boundaries. However, my result shows that centroidal smoothing on its own is not reliable enough to provide a vertex distribution that can satisfy the sphere conditions introduced by Abdelkader et al. (2020).

I discuss how my implementation reads a PLC $\mathcal{T}$ represented as an `.obj` file, with group information placed on each face using the .obj syntax. I elaborate on the implementation of triangle and edge splitting to generate new vertices for $\mathcal{T}$. I also discuss how the output Voronoi cell structure is generated, which discretizes the volume represented by $\mathcal{T}$ and whose boundary conforms to $\mathcal{T}$. Afterward, I introduce the datasets I used to evaluate the performance of my implementation and present the meshing results. Finally, reflect on my research by discussing its results.

5.1. An implemented prototype

My implementation is available on GitHub at `akossarkany/geomatics-thesis`. For simplicity of implementation and optimized computations, I used CGAL (The CGAL Project, 2025). CGAL provided most of the tools and data structures I needed to implement the prototype, including a halfedge data structure for surface representation, a kD tree for spatial lookups, and a Delaunay triangulation to construct the output Voronoi mesh. Furthermore, it provides implementations of basic linear operations, such as computing Euclidean distances, computing the circumcenters of simplices, and computing the intersections of spheres, all of which are required in my implementation. A summary of all used components is listed in Table 5.1.

To perform geometric operations in my implementation, I rely on two *geometric kernels* provided by CGAL. To represent the input surface vertices, I use the Exact Predicates Inexact Constructions Kernel (EPICK) kernel, which supports exact predicate evaluation—such as determining whether a point lies inside or outside a boundary—while allowing inexact coordinate construction for newly generated points. Although these inexact constructions may introduce minor geometric inaccuracies, I opted for this kernel because it offers substantially better performance compared to kernels that compute all constructions exactly.

In contrast, when generating the surface seeds and evaluating sphere intersections, I employ the Exact Spherical Kernel (ESK) kernel, which guarantees exact intersection tests and robust construction of the Voronoi sites.

Upon launching the application, the user must provide a configuration file in JavaScript Object Notation (JSON) that specifies an obj file defining a single PLC to be meshed, along with the parameters I describe in Subsection 4.1.2. The program reads the file and tries to construct the half-edge data structure $\mathcal{H}$ as a `Surface_mesh` of CGAL. Besides, it constructs a `Kd_tree` struct of CGAL over the vertices of the `Surface_mesh` (Subsection 5.1.1). This step fails if the input geometry contains edges with more than two incident faces. Following this, it performs iterative surface refinement and surface seed generation, I described in Section 4.2 of the methodology. In Subsection 5.1.2, I detail the implementation of the *sphere initialization* and the *triangle and edge splitting* step, and elaborate on the intersection testing. To finish the seed generation, the application generates the interior seeds using the random initialization method from Subsection 4.3.1. The final step is to construct the Voronoi diagram in a fashion suitable for OpenFOAM: a finite set of cells defined by unique vertices and faces.

5.1.1. Surface representation & Range quarries

As mentioned previously, I use the `Surface_mesh` class of CGAL to represent $\mathcal{H}$ in my methodology. A key advantage of `Surface_mesh` is that it implements a variant of a half-edge data structure, allowing the connectivity information to be recovered directly and efficiently from its topology (Section 2.5). CGAL assigns a dedicated handle (i.e. a lightweight reference) to each mesh element: faces, vertices, edges, and half-edges, and each handle uniquely identifies its corresponding element.

Surface mesh attributes with CGAL

CGAL provides an interface for augmenting the `Surface_mesh` handles with user-defined data. For every face, vertex, edge, and halfedge, one can attach a property (i.e. attribute) map that stores arbitrary data types using the element's handle as the key. I employ these property maps to store face centroids, squared sphere radii for vertices, and normal vectors for both faces and vertices.

To accelerate computation, I cache face normals, vertex normals, and triangle centroids of the `Surface_mesh` in property maps. I compute face normals using the built-in `normal` functionality of CGAL, while vertex normals are calculated as the average of the unique normals of their incident faces. The orientation of the face normals is defined by the "right-hand rule" in CGAL . For my implementation, the orientation of the faces does not matter as long as it is consistent throughout the `Surface_mesh`. The circumcenters of the faces are determined using CGAL's `circumcenter` function. Since all these quantities depend

Type	Kernel	Description
Surface_mesh	EPICK	Used throughout the meshing process to represent the connectivity of the vertices.
Kd_tree	ESK	Used during sphere initialization and intersection testing to efficiently find neighbouring spheres.
Delaunay_triangulation_3	EPICK	Used during the mesh initialization step to construct the dual DT of the final Voronoi mesh.
Point_3	EPICK/ESK	Used to represent the 3D points for the respective kernel.
Vector_3	EPICK	Used to perform vector operations since these are not defined for points.
Sphere_3	ESK	Used to represent the sphere centered at an ESK Point_3.
Plane_3	EPICK	Used during projection operations.

Function	Library	Description
normal	(core)	User to comput the normal of a triangle (right hand rule)
circumcenter	(core)	Used to compute the circumcenter of a triangle or tetrahedron.
intersection	(core)	Used to compute the (optional) intersection points of 3 spheres.
squared_distance	(core)	Used to compute the distance between two points.
Side_of_triangle_mesh	(core)	Used to determine if a point is on the bounded side of a boundary represented as a triangular mesh
smooth_shape	PMP	Used to estimate the position of vertices based on the Mean Curvature Flow of the surface
split_edge	EULER	Used to split a constrained edge when the corresponding spheres do not overlap.
split_face	EULER	Used to split the incident faces after an edge split to ensure the Surface_mesh remains triangular.
add_center_vertex	EULER	Used to split a triangle into three when the three corresponding spheres do not intersect in a circle.

Table 5.1.: Overview of core types (top) and library functions (bottom) used in the meshing implementation.

on the vertex positions, they must be recomputed whenever the surface mesh geometry is updated.

Using face normals, I identify sharp edges by computing the dihedral angle between incident faces. Furthermore, since I read the patch information during the input preprocessing step, I can use this information to identify (protected) patch boundaries (Subsection 4.1.1). I store a boolean `is_sharp_edge` predicate for each edge of the `Surface_mesh`. Then, using the sharp edge predicate, I collect all vertices incident on a protected edge and assign the `is_protected` to them as well, forming $\mathcal{H}_{SV}$ from Subsection 4.1.1, and identify the corner vertices $\mathcal{H}_{PV}$ and assign them the `is_corner`.

Range queries

In addition to the surface mesh, I maintain a `Kd_tree` from CGAL to accelerate range queries. As with normals and centroids, whenever the surface mesh is updated, I reconstruct a new `Kd_tree` $\mathcal{K}$ from its vertices. For each point stored in $\mathcal{K}$, I establish a link back to the corresponding vertex in the `Surface_mesh`. This setup provides efficient proximity queries between vertices and reduces the cost of repeated geometric lookups (using brute force).

5.1.2. Generating Voronoi sites

Initializing sphere radii

As discussed in Section 4.2.1 the sphere radii is capped by the following limit: $\min(sz, 0.49 \cdot \|p_i - p_c\|, \|p_i - p_c^*\|, \|p_i - p_p^*\|)$. To compute the minimum, I must find p_c and p_p for each p_i of the `Surface_mesh`. In my implementation, I use the vertex normals I store as a property map. Finding p_p is simple since I can use the `is_protected` predicate I placed on the vertices to find the closest point along the nearest crease. Finding p_c is also straightforward: use the `Kd_tree` and iterate the neighbors of p_i until the first that has $N(p_i)$.

Intersection testing

If three spheres overlap, their two intersection points are computed using a linear system of equations. For more details on the linear system, see Appendix A. Fortunately, CGAL provides functionality for computing sphere intersections through the `Exact_Spherical_Kernel` (ESK). ESK represents spheres by their centroid and squared radius, and exposes the function `intersect` to compute the intersections of two or three spheres. The function `intersect` uses a variant of the linear system in Appendix A

Splitting operations

To split faces in $\mathcal{H}$ represented by the `Surface_mesh` I use the Euler operations sublibrary of CGAL. More specifically, I use the `add_center_vertex` function to handle the vertex insertion. This function implements splitting and rewiring of the surface, but I still need to compute the position of the newly inserted vertex. To ensure the vertex lies inside the boundary of the original triangle, I use the centroid of the triangle as the new vertex position, which I compute as the average position of the three triangle vertices.

To insert a vertex onto an edge, I use the `split_edge` function that handles the operation for me. However, this introduces a new vertex to the incident faces, thus making them have four vertices, which is not allowed. Therefore, to ensure the `Surface_mesh` remains triangular, I use the `split_face` operation on the incident faces to provide the triangulation. It is essential to pay extra attention to which faces are selected for splitting, since each operation changes the structure of the `Surface_mesh`, old handlers can get deprecated, or referenced handles might no longer exist.

Smoothing

For my prototype, I implemented a working version of the centroidal smoothing algorithm, which distributes points evenly over a surface patch bounded by a connected loop of protected edges. This centroidal regularization preserves the protected features; however, on curved surfaces, it gradually flattens the geometry, a behavior that is undesirable in the context of terrain representation for urban CFD.

I implement a functionality for smoothing vertices on curved surfaces. CGAL provides a function `smooth_shape` in the Polygon Mesh Processing (PMP) library, which smooths a `Surface_mesh` according to mean curvature flow. To enforce curvature preservation for a single vertex, it is possible to construct a small local surface patch containing the vertex and its two-ring neighborhood (Figure 4.3). The two rings are sufficient because these vertices determine the second-order behavior of the surface in the discrete cotangent Laplacian. By constraining all vertices in the rings, the optimization effectively updates only the central vertex. The optimized position can then be mapped back to the original geometry. This operation should be applied whenever new points are inserted into the `Surface_mesh` by splitting a triangle.

Generating interior seeds

I the prototype implementation, I managed to implement the random uniform sampling of interior seeds. I sample 3 random values within the bounding box of the input surface mesh and discard samples that either lie outside the surface boundary or inside any sphere protecting the surface features. I did not have time to implement the other approaches properly.

5.1.3. Generating output Voronoi mesh

Once I have all the surface and interior seeds generated, I categorize them depending on if they lie inside or outside of the boundary $\mathcal{M}$ represented by the `Surface_mesh`. I do this by utilizing the `Side_of_triangle_mesh` functionality from CGAL, which returns which side of a manifold triangular mesh a given point lies on. I use this predicate because I assume the input is manifold. During this testing, I assign a Boolean "inside" or "outside" property to each vertex I use in the upcoming step.

To generate the output Voronoi mesh over the set of seed points. Before starting the generation, I first construct a `Delaunay_triangulation_3` data structure from CGAL, then use it to compute the dual VD. Each vertex in the DT is already marked as inside or outside by the previous step. This indicates whether the Voronoi cell generated by the Delaunay vertex is inside or outside the meshing domain defined by the input $\mathcal{T}$. I then assign a unique ID to each cell of the DT. Furthermore, I categorize each edge in DT as one of the following:

1. *Inside edge:* edge connecting two "inside" vertices.
2. *Boundary edge:* edge connecting an "inside" and an "outside" vertex.
3. *Outside edge:* edge connecting two "outside" vertices.

As discussed in Section 2.1 and 2.2, the centroids of Delaunay tetrahedra can overlap if no unique triangulation exists. To prevent the generation of degenerate faces, these points must be merged. I do this by initializing a lookup table, and for each circumcenter, I assign the same ID as the cell, then I identify overlapping points. If two points overlap, I overwrite the lookup table so the entry with the greater cell ID points to the point generated by the cell with the smaller cell ID.

As illustrated in Figure 5.1, the boundary faces of VD are generated by the *boundary edges*, and the faces generated by *outside edges* are not inside the meshing domain, therefore they get discarded. Thus, during the generation of the Voronoi cell faces, I only have to initialize a face for the edges that are either *inside* or *boundary*.

Although this function does not produce the file structure required for the mesh to be used in OpenFOAM, it derives a representation of the polyhedral Voronoi tessellation that is directly translatable into a mesh described in Subsection 2.3.1.

5.1.4. Parallelism

Since most computations operate locally on points or faces of the mesh without affecting the global topological structure (i.e., they are *independent* of one another), these steps are executed in parallel. Typical examples include computing normals, evaluating centroids, updating vertex positions, or determining per-point attributes such as sphere radii. In contrast, operations that modify the mesh topology, such as changing the number of

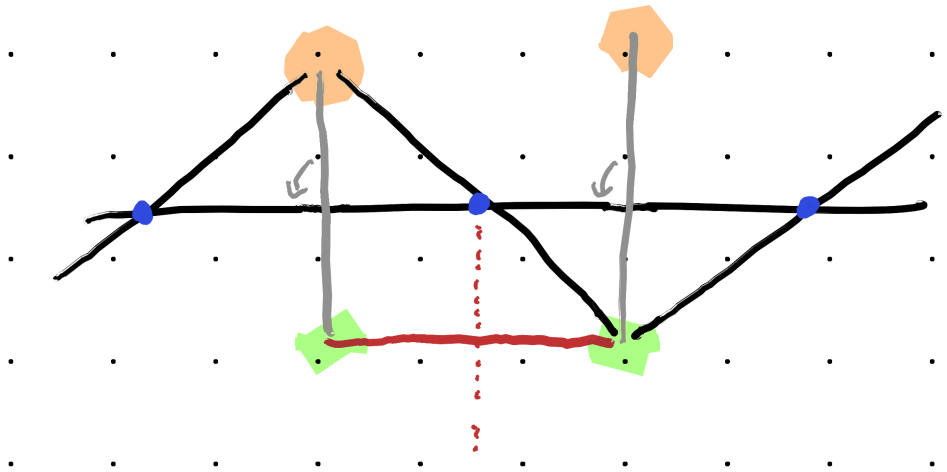


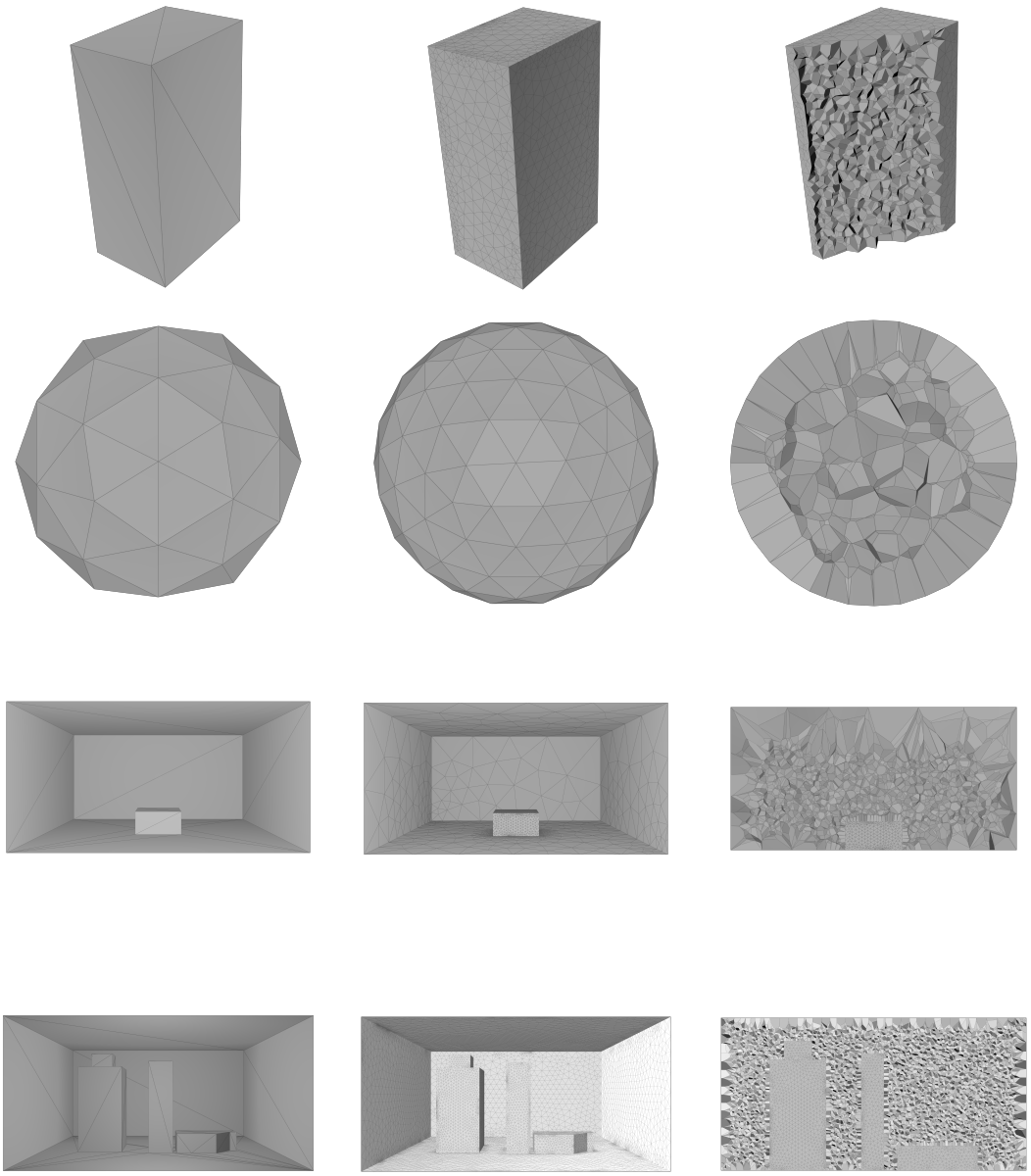
Figure 5.1.: The construction of the conforming Voronoi diagram without clipping. The figure illustrates a case in which the correct seed placement is ensured. Then the *Delaunay edges* connecting corresponding *Delaunay vertices*, one on the bounded and one on the unbounded side of the surface (grey), generate the face of the boundary cell. By design, the circumcenters of the *Delaunay cell* overlap with some neighboring circumcenters and produce a single vertex in the resulting mesh (blue). The edges connecting two *Delaunay vertices* on the unbounded side of the surface (red) should be discarded since these would generate unnecessary faces (doted red) outside the meshing domain.

vertices, altering edge connectivity, or inserting new triangles, cannot be parallelized safely. Any such update affects subsequent queries on the mesh, rendering concurrent modifications invalid. This restriction applies to all procedures that change the surface structure, such as isotropic remeshing or triangle splitting.

To parallelize independent computations, I use the OpenMP library, which provides a lightweight, convenient API for parallelizing loops. I also use OpenMP synchronization primitives to avoid concurrency issues, in particular to prevent multiple threads from writing to the same data simultaneously.

5.2. Testing the prototype

To evaluate the performance of the implemented prototype, I used four benchmark datasets. The input PLCs for the meshing, the output of the surface refinement step, and a cross-section of the resulting Voronoi meshes are illustrated in Figure 5.2. Top to bottom, the first and second are the simplest showcases of some characteristics of the output mesh, which uses a box and a sphere, respectively. The latter two are designed to test more realistic scenarios representing domains enclosing a single building and multiple buildings. The



(a) The input surface.

(b) The resulting surface of the refinement step.

(c) The internal cell structure of the resulting mesh.

Figure 5.2.: Input and results of the mesh generation process implemented by my prototype. For the experiments, I run up to 25 surface-refinement iterations, using $L = 1$ for the Lipschitz constant, 60° for the sharp-angle threshold, and various target triangle sizing depending on the surface patch types. The input geometries and experiment configuration files are available at akossarkany/geomatics-thesis.

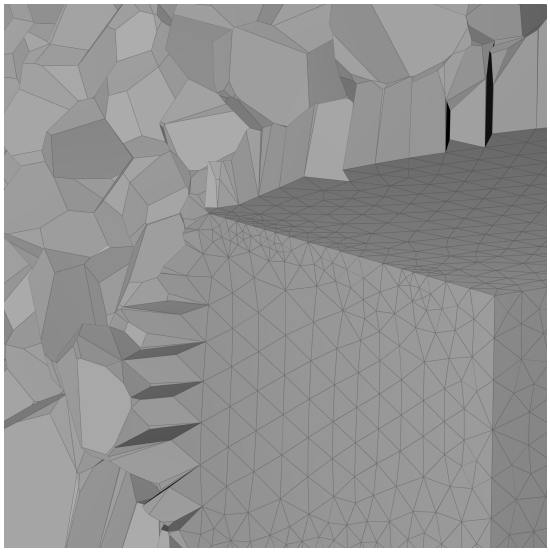


Figure 5.3.: A correctly constructed mesh where the mesh closely follows the input boundary.

`.obj` files used for the experiments and their corresponding configuration file are available on GitHub at `akossarkany/geomatics-thesis`.

During testing, the prototype had varying performance. For the first three (simplest) inputs, it could generate the output Voronoi mesh within a few iterations, and the solution converged to yield a Voronoi mesh that perfectly conforms to the input boundary. However, in other cases, it could not converge in a *reasonable* number of steps (Figure 5.3). In the most complex case, convergence could not be reached in certain parts of the boundary. Here, surface triangles could not reach a configuration such that each triangle has intersecting sphere triplets. Mainly because they got too small, and the radii of the spheres started approaching zero, yielding degenerate settings (Figure 5.4).

5.2.1. Reflection on smoothing

Centroidal smoothing

Centroidal smoothing relocates a vertex to its centroid, that is, the average position of its neighbors, which corresponds to minimizing the sum of Euclidean distances to those neighbors. While this improves local regularity, the criterion does not account for proximity to protected features. As a result, the centroid may lie inside regions where neighboring sampling spheres overlap, forcing the sphere associated with the optimized vertex to shrink and leading to degraded sampling density in heavily covered areas.

In certain configurations, this effect causes the corresponding spheres to shrink continuously until their radii approach zero. This makes it increasingly difficult to reach a configuration in which the three spheres associated with the vertices of a triangle intersect at two unique

points. When this occurs, the refinement process is more likely to terminate because the maximum number of iterations is reached, rather than due to convergence. In such cases, the resulting mesh does not perfectly conform to the input boundary (Figure 5.4). A more problematic consequence arises when the maximum number of iterations is set to a large value, as repeated triangle subdivision produces an excessive number of extremely small triangles in regions affected by uncontrolled shrinking, thereby generating many undesirable cells for the CFD simulation.

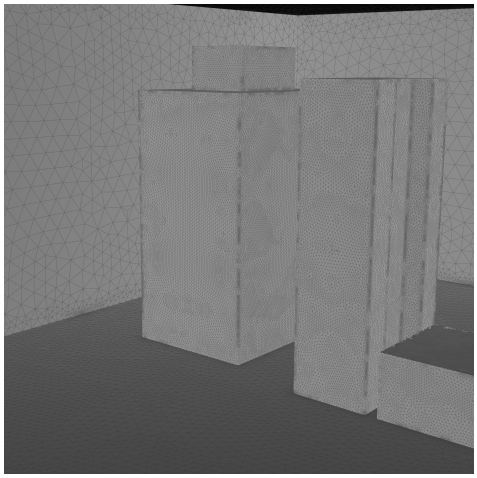
One positive property of this behavior is that it remains localized. Therefore, if the refinement process converges everywhere except in regions where uncontrolled shrinking occurs, the mesh fails to conform to the boundary only locally, while remaining conforming in regions where convergence is achieved. However, the extent of this effect highly depends on how many steps before convergence the process was stopped.

Mean curvature based smoothing

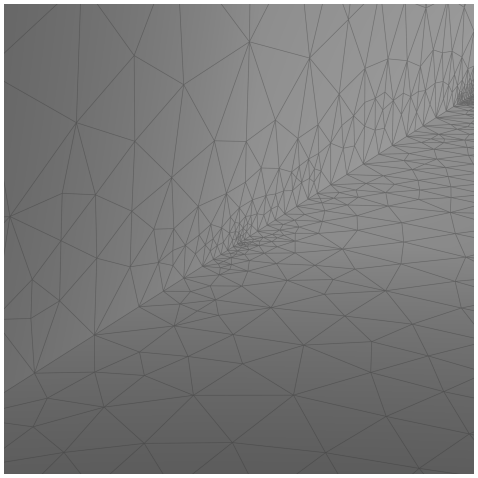
My implementation of smoothing vertices on curved surfaces occasionally fails to converge to a valid solution and returns infinite coordinates for some vertices due to an issue I was unable to identify within the timeframe of this thesis. Figure 5.2 illustrates a case when the implementation behaves as expected when applied to a discrete representation of a sphere. Given the priorities of urban CFD, where accurately capturing building geometry, dominated by planar surfaces and sharp features, is more critical than preserving terrain curvature, I directed my efforts toward stabilizing the power-distance-based regularization, which still required substantial refinement within the developed implementation. One way to currently accurately mesh smooth features is to set a near-zero sharp feature threshold, which marks every non-flat edge in the input PLC sharp, and thus preserves them.

5.2.2. Generating output Voronoi mesh for OpenFOAM

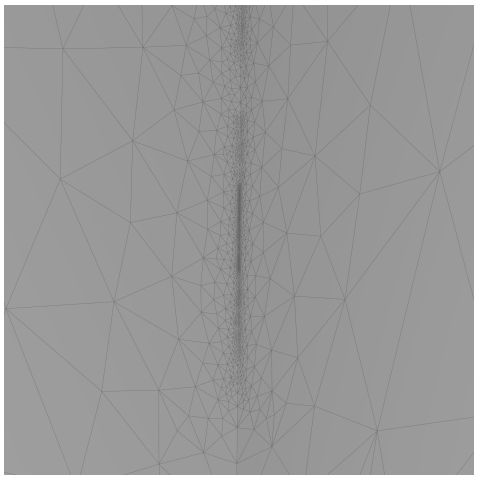
My implementation for generating the mesh still has a bug when, in some configurations, it returns infinite values, which offsets the numbering of the vertices in the Delaunay triangulation of the Voronoi seeds (Figure 5.5a). But in other cases, it constructs the mesh correctly. The resulting mesh follows OpenFOAM guidelines for face orientation and cell referencing (Section 2.3.1). Since sometimes parts of the mesh are constructed incorrectly, to reamin consistent I exported the generated Voronoi sites as a Coma Separated Values (CSV) file, and used external software, Houdini Educational (SideFX Houdini, Side Effects Software Inc.), to generate the Voronoi tessellation of the exported sites using its built-in methods, which I do not disclose here (Figure 5.5b).



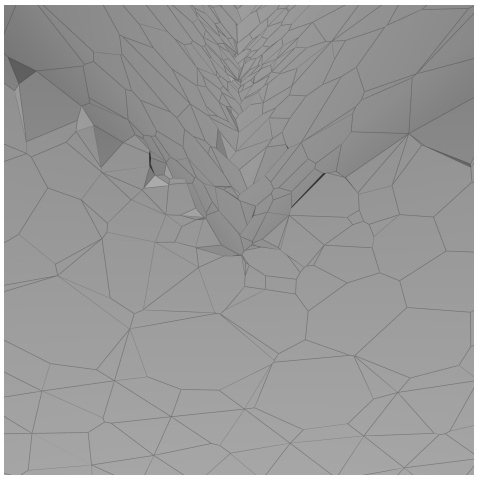
(a) The overall result of the surface refinement step is a densely triangulated mesh with regularized triangles.



(b) However, in some cases, the triangles in certain regions get subdivided too many times.

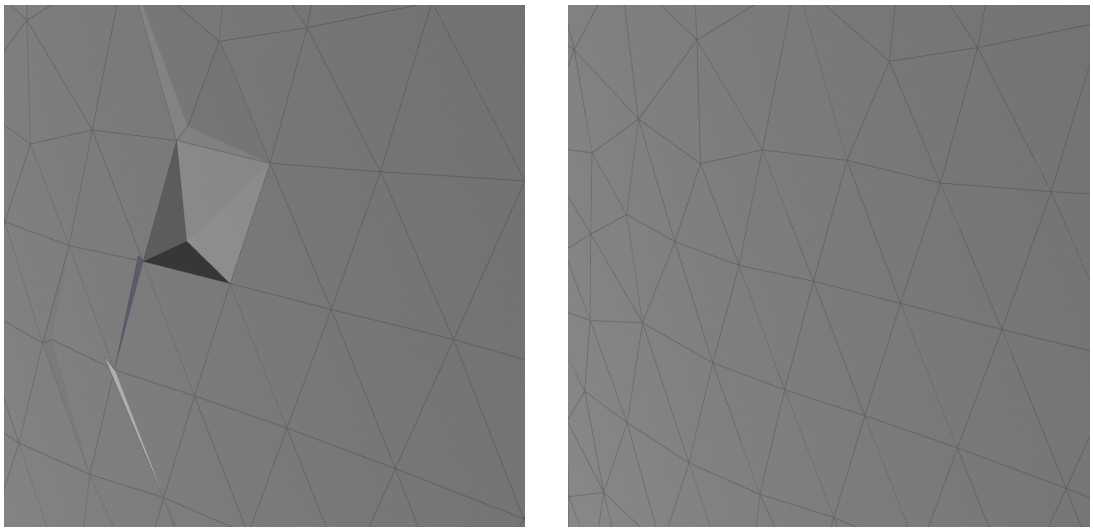


(c) If the maximum number of iterations is set to a high number, the triangle subdivision can happen uncontrollably, yielding extra small faces.



(d) If the maximum number of iterations is reached before the refinement process can converge, there are some triangles that have spheres on their vertices that do not intersect. At these places, Voronoi sites are not generated, and thus the Voronoi mesh does not conform to the input boundary locally.

Figure 5.4.: If the surface refinement step does not converge within the specified iteration limits, the resulting mesh still exhibits acceptable quality, with only local suboptimalities in boundary conformity and cell shape. However, overall acceptability depends on how close the process is to convergence when it is stopped. The fewer iterations that remain until convergence, the better the resulting mesh quality.



(a) There is a bug in my code, which constructs the Voronoi diagram from a set of Voronoi sites. For certain inputs, it correctly generates the sites, but the reconstructed mesh contains degenerate configurations.

(b) The mesh constructed with an external program from the same seeds is correct, i.e., the seed placement is correct, and the issue is with the Voronoi mesh construction.

Figure 5.5.: The reconstructed Voronoi mesh of the Voronoi sites.

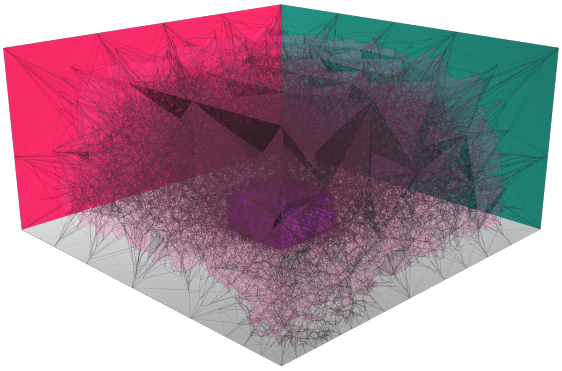


Figure 5.6.: The patch types (different colors) are correctly preserved during meshing.

5.3. Results

My methodology produces a Voronoi mesh that conforms to a 2-manifold boundary specified by a PLC, without requiring any cell clipping. The resulting mesh preserves the boundary information present in the input geometry by conforming not only to sharp geometric features, but also to edges separating different boundary patch types (Figure 5.6). The methodology requires configuring only two groups of parameters, with each parameter independently controllable for each surface patch type on the boundary. The first parameter is a sharp-angle threshold that determines which edges are considered sharp based on their dihedral angle. The second parameter is a sizing parameter that defines the maximum allowable area of a cell face incident on the domain boundary. Additional parameters are used to limit the iterative refinement processes.

The meshing process terminates only when the union of spheres fully covers the boundary surface. Therefore, if the process converges, the resulting mesh necessarily conforms to the boundary. If convergence is not achieved before the maximum number of iterations is reached, the method still produces an acceptable result. Even if the mesh exhibits local issues in boundary conformity, it overall conforms to the input geometry more closely than the mesh generated by snappyHexMesh, which is used in the current City4CFD pipeline (Pađen et al., 2022). This is because boundary conformity is determined locally. If Voronoi site initialization fails at certain locations, it affects only neighboring cells and does not impact other parts of the mesh where the process succeeds. Furthermore, such local errors do not propagate to subsequent iterations when placing interior cells.

In this methodology, I also discuss an approach to achieving a gradual transition from high cell density near the boundaries to lower density in the interior of the mesh. I propose generating hexahedral cells in low-density interior regions, far from geometric features, while using polyhedral cells only near buildings and terrain. This approach leverages the geometric flexibility of polyhedral cells where boundary conformity is required and exploits the simpler computations enabled by hexahedral cells in regions without such constraints. The gradual density transition, combined with the properties of Voronoi diagrams, ensures that cells remain regularly sized and shaped.

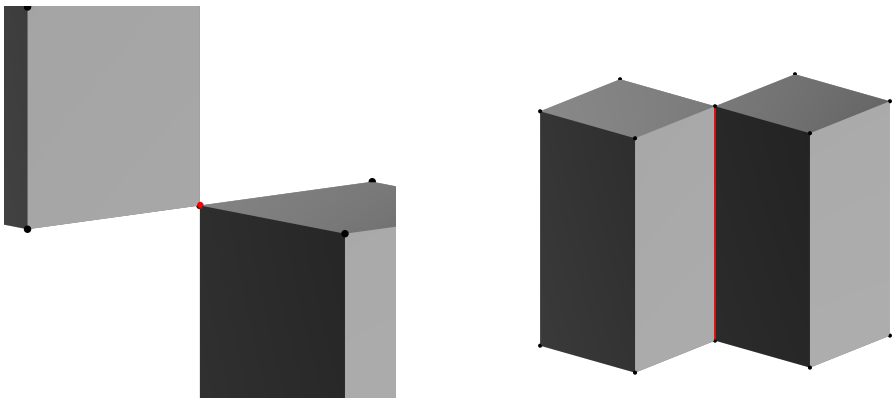
The prototype implementation can construct a polyhedral mesh tailored for OpenFOAM. It directly generates a boundary-conforming polyhedral mesh from Voronoi seeds using the dual Delaunay triangulation, eliminating the need for any clipping during mesh construction. This improves robustness, since clipping operations scale poorly with the number of cells. During mesh construction, boundary patch information is propagated from the Voronoi seeds to the cells adjacent to the boundary. Cell faces are constructed in a form that can be directly translated into an OpenFOAM mesh without further processing.

Overall, the main contribution of this work is to consolidate existing Voronoi-based meshing strategies and to formulate a new methodology for meshing manifold boundaries in urban CFD applications while preserving semantic patch information. In addition, I developed a partial implementation of a Voronoi mesh reconstruction pipeline tailored to the boundary

and connectivity requirements of OpenFOAM meshes. Furthermore, the method requires minimal configuration, making it well-suited for automated pipeline processing.

5.3.1. Limitations

The prototype cannot robustly handle all input geometries. If the input is non-manifold, constructing a valid `Surface_mesh` fails, rendering further processing impossible, and the process terminates with an error. Although non-manifold configurations are uncommon in typical datasets, situations such as those illustrated in Figure 5.7 can still arise, most notably when non-adjacent faces share a vertex or when an edge has more than two incident faces. These edges are problematic because they can occur in real configurations. A further problematic case is when the input contains overlapping faces. In such configurations, the distance from the vertices of one face to the closest non-co-smooth point on the overlapping face is zero, resulting in spheres with zero radius. These spheres cannot intersect with neighboring spheres, causing the associated faces to be split indefinitely and preventing the algorithm from converging. This is a less common problem, since such configurations arise primarily from incorrect geometry (re)construction.



(a) (b)

Figure 5.7.: The cases of problematic non-manifold geometry. In the first case (a), the vertex is shared by non-edge-adjacent faces, while in the second case (b), the blue edge is shared by four faces.

Further limitation of the prototype is the occasional uncontrolled local shrinking of the spheres. As discussed in Subsection 5.2.1, the root cause of this is the drifting side effect of the constrained centroidal smoothing. This prevents the final mesh from closely conforming to the input boundary and introduces local suboptimalities. Additionally, the prototype implementation still contains bugs in the mean-curvature-based smoothing and the generation of the output Voronoi mesh. In both cases, occasionally infinite values are encountered, resulting in errors and halting the process.

6. Conclusions & Future work

In this thesis, I develop a mesh generation technique tailored for urban CFD. I investigate the properties of meshes relevant to urban flow simulations and review existing methodologies for polyhedral mesh generation, focusing on the characteristics of the meshes they produce. In particular, I study methods that generate polyhedral meshes whose cell boundaries directly conform to a given input boundary, and I review approaches for constructing such boundary-conforming Voronoi configurations. This addresses my first research subquestion on generating a boundary-conforming Voronoi grid. I also review work describing how to achieve such sampling on a PLC, which answers my second research subquestion.

Building on these existing techniques, I propose a novel approach for meshing any volume represented by a valid 2-manifold boundary. The proposed method generates cells that conform not only to the boundary itself, but also to the boundaries between different surface patch types on the manifold. This allows boundary patch information to be migrated to the corresponding cell faces, answering my third research subquestion. In the methodology chapter, I discuss how boundary triangles are split when they are too large relative to the local feature size, following the sphere-based refinement strategy of the VoroCrust algorithm (Abdelkader et al., 2020). This ensures that boundary faces of the resulting Voronoi cells have smaller areas near fine geometric features and larger areas farther away, which also influences the volume and sizing of the cells. Additionally, by splitting triangles based on area thresholds and applying centroidal smoothing, the sizing of surface triangles and the corresponding Voronoi cells transition gradually between neighboring regions. I further review a Poisson-disk-sampling-based approach and propose an iterative random sampling strategy to achieve a smooth transition from smaller cells near the boundary to larger cells toward the interior of the mesh, addressing my fifth and final research subquestion.

Based on this analysis, I answer the main research question at a theoretical level. It is possible to construct a boundary-conforming polyhedral mesh from a Voronoi diagram that preserves boundary patch information, since such a structure can conform both to the geometric boundary and to the patch boundaries present on the surface.

My experimental results show that the implemented prototype can derive a mesh from a 2-manifold boundary using the proposed methodology. The resulting mesh conforms well to the input boundary and, in some cases, matches it exactly. Cell sizing adapts to both user-defined thresholds on boundary-adjacent face areas and the local feature size of the boundary. Moreover, faces adjacent to the boundary successfully store the surface patch information corresponding to the input geometry. To ensure that cells do not span multiple patch types, the mesh also conforms to 1-manifold boundaries between patches. When the

refinement process does not converge before reaching the maximum number of iterations, the method still produces a mesh of acceptable quality, where boundary conformity is maintained in most regions, and deviations are limited to small local areas. This allows accuracy to be traded for reduced meshing time.

The main contribution of this research is the prototype implementation, which can serve as a foundation for developing a fully automatic reconstruction pipeline for urban CFD. To achieve this, further work is required to stabilize and finalize the mesh output stage. An additional contribution of this thesis is the academic discussion of the research process itself, in which I answer the research question and develop a sound methodology for generating meshes tailored to urban flow simulations. Nevertheless, the broader problem remains open, and several directions for future work remain.

6.1. Future work

Improving triangle splitting and smoothing

Triangle splitting is effective for increasing vertex density, but if left unchecked, the density may grow unreasonably over some areas of the surface mesh. It can happen when centroidal smoothing of the vertices optimizes only by distance, which occasionally yields solutions that are not optimized for sphere overlaps. A potential but unproven fix would be to modify the objective itself. Instead of minimizing only Euclidean distances, one can additionally maximize the distance from the circumferences of the neighboring spheres (Figure 6.1b). Let p_i be the neighbor centers with radii r_i, and define the signed distance

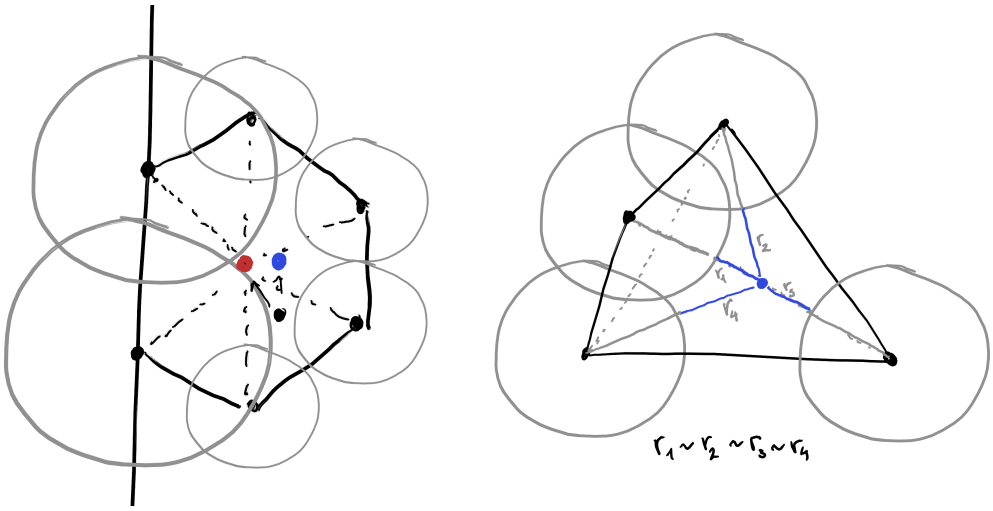
$$d_i(x) = \|x - p_i\| - r_i,$$

which is negative when x lies inside sphere i.

To avoid drifting into deeply covered regions, we maximize the *minimal* signed distance to the sphere boundaries, pushing the vertex away from all overlaps in a worst-case sense, while simultaneously minimizing its average distance to the neighbors to ensure a bounded, non-trivial solution:

$$\max_{x \in \mathbb{R}^2} \left(\min_i d_i(x) \; - \; \lambda \frac{1}{n} \sum_{i=1}^{n} \|x - p_i\| \right).$$

Here $\lambda > 0$ controls the tradeoff between staying close to the neighbors and avoiding deep penetration into their spheres. This augmented objective places the vertex as far as possible from all sphere boundaries while remaining geometrically coherent, preventing it from collapsing into deeply covered regions whenever an admissible position exists. Such a formulation might be optimized with standard nonlinear solvers, e.g. Gauss–Newton or Levenberg–Marquardt, after replacing the hard minimum with a differentiable soft-min approximation, or simply, by modifying to centroidal smoothing algorithm.



(a) (b)

Figure 6.1.: Illustration of vertex placement optimization with respect to overlaps in 2D. Figure (a) illustrates how using only centroidal vertex updates makes the vertex move closer to the boundary (red) while the appropriate placement is where the overlap with each surrounding circle is maximal (blue), while Figure (b) illustrates the optimal position for the vertex.

6.1.1. Interior seed sampling

In my prototype implementation, I realized the random uniform sampling strategy from my methodology to generate interior seeds. However, as discussed in Section 4.3, this approach is not the most suitable for urban CFD. Instead, it would be beneficial to implement the methodology described earlier to obtain a mesh with gradually increasing cell size farther from the boundary, with a hexahedral grid in regions distant from boundary features. Such a mesh would provide a more efficient computational domain for CFD simulations.

Preserving curved patches

The current prototype does not implement Laplacian-based smoothing correctly. Therefore, it cannot preserve curved geometries in all cases, specifically in surface regions where dihedral angles do not exceed the sharp-feature threshold and are not separated by patch boundaries. While this limitation is negligible for building geometries in an urban context, it becomes significant when the terrain is relevant to the simulation and contains smooth curvature. Supporting such cases would thus be an appropriate extension of the methodology.

Fixing overlaps at partitioning walls

Overlaps remain an issue because `City4CFD` does not handle partition walls correctly (Section 2.4). For the algorithm to terminate reliably, these overlaps must be resolved before the prototype can derive surface seeds. Otherwise, the spheres associated with vertices along the partition shrink to zero radius to eliminate intersection with the overlapping face. This leads to an uncontrollable subdivision of triangles, since spheres of zero radius cannot overlap.

Compare with `snappyHexMesh`

Compare the results from my mesher and `snappyHexMesh` to empirically evaluate their relative performance. In terms of conforming to boundaries, cell regularity, number of cells, meshing time, and level of automatization.

Bibliography

Abdelkader, A., Bajaj, C. L., Ebeida, M. S., Mahmoud, A. H., Mitchell, S. A., Owens, J. D., & Rushdi, A. A. (2018). Sampling conditions for conforming voronoi meshing by the vorocrust algorithm. *LIPIcs: Leibniz international proceedings in informatics*, *99*, 10.4230/LIPIcs. SoCG. 2018.1. https://doi.org/10.4230/LIPIcs.SoCG.2018.1

Abdelkader, A., Bajaj, C. L., Ebeida, M. S., Mahmoud, A. H., Mitchell, S. A., Owens, J. D., & Rushdi, A. A. (2020). Vorocrust: Voronoi meshing without clipping. *ACM Transactions on Graphics (TOG)*, *39*(3), 1–16. https://doi.org/10.48550/arXiv.1902.08767

Amenta, N., Bern, M., & Eppstein, D. (1999). Optimal point placement for mesh smoothing. *Journal of Algorithms*, *30*(2), 302–322. https://doi.org/10.1006/jagm.1998.0984

Amenta, N., Choi, S., & Kolluri, R. K. (2001). The power crust, unions of balls, and the medial axis transform. *Computational Geometry*, *19*(2-3), 127–153.

Aurenhammer, F. (1987). Power diagrams: Properties, algorithms and applications. *SIAM Journal on Computing*, *16*(1), 78–96.

Berge, R. L., Klemetsdal, Ø. S., & Lie, K.-A. (2019). Unstructured voronoi grids conforming to lower dimensional objects. *Computational Geosciences*, *23*, 169–188. https://doi.org/10.1007/s10596-018-9790-0

Blocken, B. (2015). Computational fluid dynamics for urban physics: Importance, scales, possibilities, limitations and ten tips and tricks towards accurate and reliable simulations. *Building and Environment*, *91*, 219–245. https://doi.org/10.1016/j.buildenv.2015.02.015

Desbrun, M., Meyer, M., Schröder, P., & Barr, A. H. (1999). Implicit fairing of irregular meshes using diffusion and curvature flow. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 317–324.

Ebeida, M. S., & Mitchell, S. A. (2012). Uniform random voronoi meshes. *Proceedings of the 20th international meshing roundtable*, 273–290. https://doi.org/10.1007/978-3-642-24734-7

Ebeida, M. S., Mitchell, S. A., Patney, A., Davidson, A. A., & Owens, J. D. (2012). A simple algorithm for maximal poisson-disk sampling in high dimensions. *Computer Graphics Forum*, *31*, 785–794.

Edelsbrunner, H., & Shah, N. R. (1996). Incremental topological flipping works for regular triangulations. *Algorithmica*, *15*(3), 223–241.

Guo, J., Yan, D.-M., Jia, X., & Zhang, X. (2015). Efficient maximal poisson-disk sampling and remeshing on surfaces. *Computers & Graphics*, *46*, 72–79.

Hildebrandt, K., & Polthier, K. (2007). Constraint-based fairing of surface meshes. *Symposium on Geometry Processing*, 203–212.

Ju, L. (2007). Conforming centroidal voronoi delaunay triangulation for quality mesh generation. *Inter. J. Numer. Anal. Model*, *4*, 531–547.

LaForce, T., Ebeida, M., Jordan, S., Miller, T. A., Stauffer, P. H., Park, H., Leone, R., & Hammond, G. (2023). Voronoi meshing to accurately capture geological structure in subsurface simulations. *Mathematical Geosciences*, *55*(2), 129–161. https://doi.org/10.1007/s11004-022-10025-x

López-Pachón, M., & Marcé-Nogué, J. (2025). The crucial role of meshing in computational fluid dynamics simulations for organic geometries in paleobiology: Describing fluid dynamics performance through best practices. *Methods in Ecology and Evolution*, *16*(10), 2170–2194. https://doi.org/https://doi.org/10.1111/2041-210X.70146

Merland, R., Caumon, G., Lévy, B., & Collon-Drouaillet, P. (2014). Voronoi grids conforming to 3d structural features. *Computational Geosciences*, *18*(3), 373–383. https://doi.org/10.1007/s10596-014-9408-0

Mitchell, S. A., Ebeida, M. S., Awad, M. A., Park, C., Patney, A., Rushdi, A. A., Swiler, L. P., Manocha, D., & Wei, L.-Y. (2018). Spoke-darts for high-dimensional blue-noise sampling. *ACM Transactions on Graphics (TOG)*, *37*(2), 1–20.

Pađen, I., García-Sánchez, C., & Ledoux, H. (2022). Towards automatic reconstruction of 3d city models tailored for urban flow simulations. *Frontiers in Built Environment*, *8*. https://doi.org/10.3389/fbuil.2022.899332

Pađen, I., Peters, R., García-Sánchez, C., & Ledoux, H. (2024). Automatic high-detailed building reconstruction workflow for urban microscale simulations. *Building and Environment*, *265*, 111978. https://doi.org/10.1016/j.buildenv.2024.111978

The CGAL Project. (2025). *CGAL user and reference manual* (6.1). CGAL Editorial Board. https://doc.cgal.org/6.1/Manual/packages.html

Yan, D.-M., Wang, W., Lévy, B., & Liu, Y. (2013). Efficient computation of clipped voronoi diagram for mesh generation. *Computer-Aided Design*, *45*(4), 843–852.

Yan, D.-M., & Wonka, P. (2013). Gap processing for adaptive maximal poisson-disk sampling. *ACM Transactions on Graphics (TOG)*, *32*(5), 1–15.

A. Intersection of 3 balls

Let $b_i = (p_i, r_i), \quad b_j = (p_j, r_j), \quad b_k = (p_k, r_k)$, be three balls in $\mathbb{R}^3$ with centroids p_ℓ and radii r_ℓ. To determine whether their boundary spheres intersect in zero, one, or two points, we reduce the problem to a linear system followed by a single quadratic equation.

Linear Reduction Each sphere satisfies $\|x - p_\ell\|^2 = r_\ell^2, \quad \ell \in \{i, j, k\}$. Expanding and subtracting the equation for b_i from the one for b_j eliminates the quadratic term $\|x\|^2$ and yields the linear constraint $2(p_j - p_i) \cdot x = \|p_j\|^2 - \|p_i\|^2 + r_i^2 - r_j^2$. Repeating the same operation using the pair (i, k) gives $2(p_k - p_i) \cdot x = \|p_k\|^2 - \|p_i\|^2 + r_i^2 - r_k^2$.

These two constraints define the linear system $Ax = b$, where

$$A = \begin{pmatrix} 2(p_j - p_i)^\mathsf{T} \\ 2(p_k - p_i)^\mathsf{T} \end{pmatrix}, \qquad b = \begin{pmatrix} \|p_j\|^2 - \|p_i\|^2 + r_i^2 - r_j^2 \\ \|p_k\|^2 - \|p_i\|^2 + r_i^2 - r_k^2 \end{pmatrix}.$$

The solution set of $Ax = b$ is an affine space of dimension $3 - \operatorname{rank}(A)$. For generic configurations, $\operatorname{rank}(A) = 2$, and the solution is a line $x(t) = x_0 + tn$, where x_0 is a particular solution and n spans the nullspace of A. If the system is inconsistent, the three spheres cannot intersect.

Quadratic Constraint Assuming $Ax = b$ is consistent, any common intersection point must also satisfy one sphere equation, e.g. $\|x(t) - p_i\|^2 = r_i^2$. Substituting the line parameterization yields the quadratic equation $\alpha t^2 + \beta t + \gamma = 0$, obtained by expanding $\|x_0 + tn - p_i\|^2 - r_i^2 = 0$.

Let $\Delta = \beta^2 - 4\alpha\gamma$ be the discriminant. Intersection classification:

- $\Delta > 0$: two distinct solutions t_1, t_2, yielding intersection points $x(t_1)$ and $x(t_2)$.
- $\Delta = 0$: one real solution; the spheres are tangent.
- $\Delta < 0$: no real solution; the affine line generated by the first two spheres does not intersect the third.

Recovering the Intersection Points If the quadratic equation admits real solutions, the corresponding intersection points are obtained by substituting the roots back into the line parameterization. Let t_1 and t_2 be the real solutions of $\alpha t^2 + \beta t + \gamma = 0$. The intersection points of the three spheres are then

$$x_1 = x(t_1) = x_0 + t_1 n, \qquad x_2 = x(t_2) = x_0 + t_2 n.$$

In the degenerate case $\Delta = 0$, the two roots coincide, and the spheres intersect at exactly one point $x^* = x_0 + t^* n$. Thus, once the linear system has been solved to obtain x_0 and the nullspace direction n, recovering the actual intersection coordinates amounts to evaluating the affine map $x(t)$ at the real roots of the quadratic constraint.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class scrbook. The main font is Inconsolata. The figures were rendered with OpenGL through Houdini FX, using an *Educational* license.