UNIVERSITÀ DEGLI STUDI DI CAGLIARI

FACOLTÀ DI SCIENZE

Corso di Laurea Magistrale in Informatica

Advancing Volumes

A Tetrahedral Mesh Generation Method

Supervisors
Dott. Gianmarco Cherchi
Dott. Marco Livesu

Candidate
Federico Meloni
Matr. N. 60/73/65243

ACADEMIC YEAR / ANNO ACCADEMICO 2022/2023

This thesis presents an algorithm for the automatic generation of volumetric meshes from surface meshes. Volumetric meshes are essential for representing the internal volume of objects and find applications in various fields such as industry, medicine, video games, and art. The proposed algorithm aims to fill the entire volume described by the surface mesh by growing a volumetric mesh inside it. The algorithm employs a combination of refinement and smoothing steps to maintain mesh quality and regularize the structure. The refinement step adds smaller polyhedra to increase resolution, while the smoothing step equalizes the distance and size of elements. The algorithm focuses on generating tetrahedral meshes and operates in a fully automatic manner, taking into account input constraints. In future developments, this algorithm has the potential to serve as a tool for more intricate tasks, including the creation of volumetric maps.

Contents

Chapter 1

Introduction

Meshes are widely used for the digital representation of an object. These consist of the discretization of the space in order to represent an object with a set of primitives such as points, segments, polygons, and polyhedra. The most popular types of meshes are undoubtedly surface meshes, where only the skin of the object is represented with a set of connected polygons. Surface meshes are used in numerous fields, such as data visualization, video games, entertainment, etc.

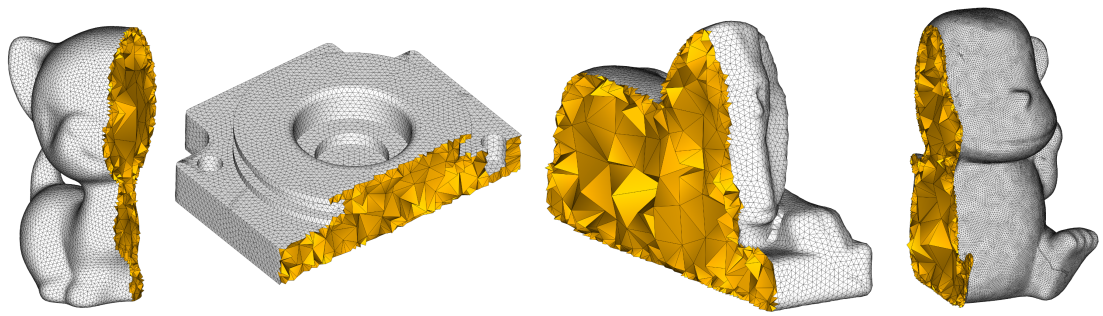

Figure 1.1: *Some examples of volumetric meshes composed of tetrahedra*

In some contexts, the surface representation is not enough; it becomes necessary to depict the internal volume of the object. The introduction of internal connectivity has resulted in the representation of objects as a set of polyhedra connected to each other instead of simple polygons. These meshes are called volumetric meshes.
Volumetric meshes will hold an increasingly important place in a world where nothing is left to chance, and efforts are made to minimize cost and waste. These meshes are mainly used in industry, where, due to their internal structure, they allow us to simulate stresses and internal forces in an object through its volume without having to produce any physical prototype first and with a low margin of

error in simulations. These methods are called Finite Elements Analysis.
We find these simulations in disparate cases, such as machine components where we need to understand how heat propagates, wind tunnels where fluid-dynamics simulations are performed, bridge construction with load simulations, and so on. We can also use volumetric meshes to prevent any malfunctions and schedule maintenance. Volumetric meshes can be found not only in industry but also in medicine, video games, and art. Typically, volumetric meshes consist of either tetrahedra or hexahedra, but it is also possible to employ polyhedra of various kinds, including a combination of them if needed.
This thesis proposes a new algorithm for automatically generating a volumetric mesh from a surface one provided as input. The idea behind the algorithm is to let a volumetric mesh grow inside the surface mesh in a way that fills the entire volume described by the surface mesh. We can imagine it as a balloon being inflated inside a model, filling the entire available space.
When the algorithm grows the volumetric mesh, it has to deform it to fill increasingly more and more volume. As we deform the mesh, the elements that compose it stretch, losing quality. We define a high-quality polyhedron when it is as close as possible to being equilateral.
The algorithm alternates steps of refining and smoothing with expansion ones to avoid stretches and loss of quality. The refinement step is used to add polyhedra and increase the resolution; therefore, the space occupied by a distorted polyhedron is occupied by smaller polyhedra with a better shape. This step is performed with topological operators, which act on the connectivity of elements. Instead, the smoothing step is used to equalize the distance and size of the elements in order to regularize the structure of the entire mesh.
The algorithm proposed in this thesis aims to generate a mesh of tetrahedra in a fully automatic manner, respecting the constraints imposed by the input and trying to maintain a good shape of the elements that compose the mesh. In future developments, this algorithm can be seen as an aid for more complex operations, such as that of creating volumetric maps.

Chapter 2

State of the art

2.1 Background on Meshes

A mesh is a data structure that allows us to represent an object discretely in three-dimensional space through primitives such as vertices, edges, faces, and polyhedra (in the case of volumetric meshes). In contrast to surface meshes, which only represent the surface of a three-dimensional object, volumetric meshes also depict the internal structure of the entities. This internal structure allows us to save a lot of additional information and perform a whole set of additional operations compared to surface meshes.

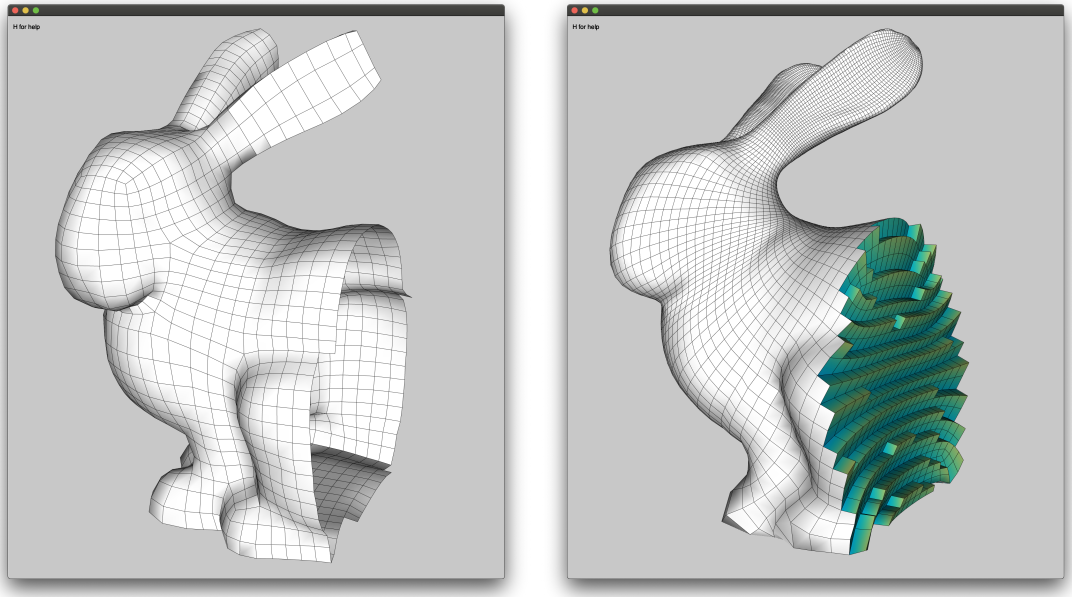

Figure 2.1: *Comparison of the interior of a surface (left) and a volumetric (right) mesh.*

Meshes are defined by:

- **Geometry**: the set of vertices composing the model, each of which is characterized by a position in $\mathbb{R}^3$ described by a three-dimensional vector $\mathbf{v} = (x, y, z)$;

- **Connectivity (topology)**: edges, faces, and polyhedra are various ways and levels of connecting vertices:

  - **Edges** are defined by a pair of vertices $(\mathbf{v}_i, \mathbf{v}_j)$;
  - **Faces** are defined by a set of vertices $\mathbf{v}_i$;
  - **Polyhedra** are defined by a set of faces $\mathbf{f}_i$.

- **Attributes**: any value (scalar or vector) that linearly varies associated with the elements of the mesh (e.g., normals).

General characteristics Given a geometry that defines the mesh's vertices, its topology is its most crucial aspect. Topology defines various levels (and dimensions) of connection between vertices, thereby establishing adjacencies among different mesh parts. Edges represent connections between two vertices (1 dimension), faces are polygons formed by a set of edges (2 dimensions), and polyhedra are, in essence, a collection of faces that bind a closed space (3 dimensions). This final step in topology enables us to describe the internal structure of a mesh and, as previously mentioned, distinguishes volumetric meshes from surface meshes. Each of these connectivity-defined elements has adjacencies, specifically:

- A vertex can be adjacent to another vertex (connected by an edge), a face (part of it), or a polyhedron (part of it).

- An edge can be adjacent to another edge (sharing a vertex), a face (part of it), or a polyhedron (part of it).

- A face can be adjacent to another face (sharing an edge) or to a polyhedron (part of it).

- A polyhedron can be adjacent to another polyhedron (sharing a face).

It is noteworthy that these adjacency relationships are symmetric.

Classifications In general, volumetric meshes can be categorized based on the polyhedra that compose them. The most commonly used polyhedra are **tetrahedra** and **hexahedra**. However, volumetric meshes can be composed of polyhedra with any number of vertices or with different types of polyhedra concurrently, referred to as hybrid meshes (e.g., a mesh can be composed of both tetrahedra and hexahedra). Meshes composed of tetrahedra are called **tetmeshes** and are the simplest to construct. Meshes composed of hexahedra (also known as cuboids) are referred to as **hexmeshes**. These are more complex and challenging to manage.

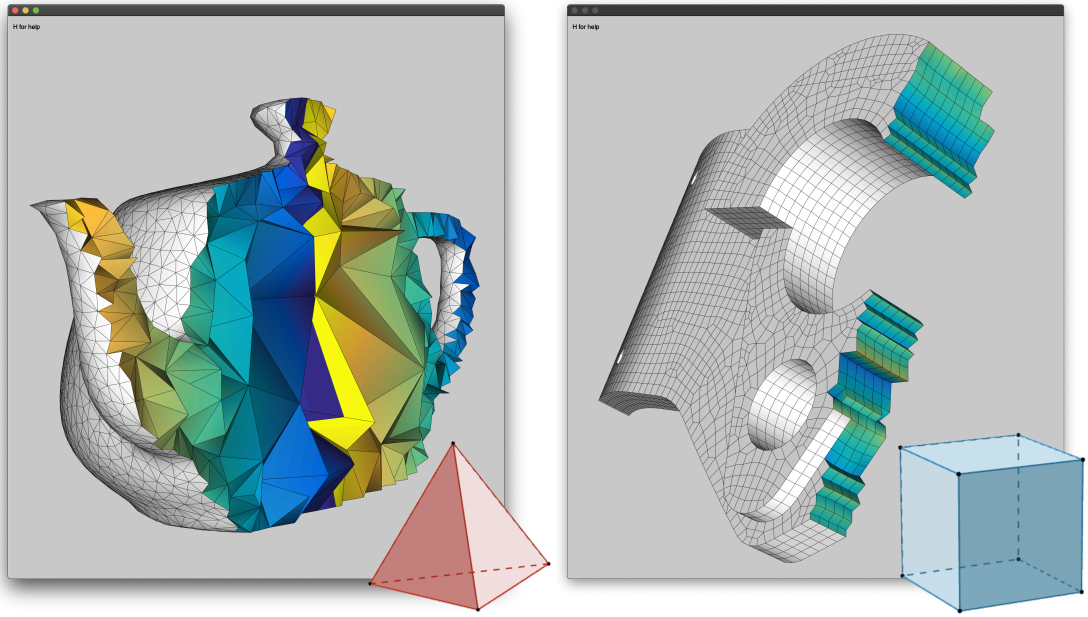

Figure 2.2: *Comparison of sections of a tetmesh (left) and a hexmesh (right)*

Application While surface meshes find application in various fields such as video games and modeling, volumetric meshes are primarily employed for physical simulation, particularly in Finite Element Analysis (FEA). Volumetric meshes enable the virtual simulation and verification of structural properties of represented objects. An example includes load simulations or thermodynamic simulations. Another emerging application domain for volumetric meshes is animation, as they can simulate a more realistic physics of objects in a scene. For effective use in simulations, a volumetric mesh must possess elements of good "quality," meaning their shape should be far from degenerate and as close as possible to their corresponding equilateral form. In general, constructing a hexahedral mesh is more challenging. Still, it can offer more efficient simulations (at the same resolution) or more accurate results (at the same execution time) than a tetrahedral mesh representing the same object.

2.2 Tetrahedral mesh generation

The generation of tetrahedral meshes has always been a recurring challenge, involving both experts in computational geometry and professionals across domains like graphics, physics, and engineering because of its inherent complexity. Established methodologies frequently fail to support automatic pipelines, requiring manual intervention to produce a valid mesh. Users often find themselves forced to "fix" input surface meshes to make mesh generation processes go smoothly due to implicit prerequisites, or, alternately, post-processing becomes imperative for rectifying tetrahedral meshes due to the inability to meet basic post-generation conditions, such as manifoldness. In many cases, even if meshing procedures can achieve some degree of success, the resulting mesh size may prove excessive for various applications, largely due to the absence of a mechanism that allows optimal control over the balance between the quality of approximation and the resulting mesh's dimensions. Moreover, even in scenarios where such control mechanisms are present, inconspicuous attributes of the input mesh, which are hard to detect, may not be preserved.

The following is a compilation of some prominent approaches to tetrahedral mesh generation, along with their respective advantages and disadvantages.

Advancing Fronts This is one of the first attempts to generate tetmeshes algorithmically and is derived directly from its surface counterpart. Starting from a triangle boundary, advancing front methods incrementally insert tetrahedra. The volume is progressively filled until small voids are solved with simple patterns made of a few tetrahedral cells. Such an approach is challenging on one main point: fronts can collide during their generation, and geometric intersection must be performed to solve collisions, which is a costly operation. This problem has multiple solutions, such as backtracking and retrying to insert the tetrahedron in a different position, but this does not always work.

Background Grids Background grids are the most straightforward approach to mesh generation. They are based on the concept of a regular grid, which is a set of points in space that are equidistant from each other, and they take advantage of the fact that a cube is easy to tetrahedralize with five, six, or twelve tetrahedra. The grid is then deformed to fit the input geometry, and its resolution determines the mesh's quality, with higher resolutions resulting in better-quality meshes. Grid-based methods can fill the ambient space with a uniform or an adaptive octree. The latter is more suitable for complex geometries, as it allows for a higher resolution in the areas that require it while maintaining a lower resolution in the rest of the space.

Grid cells located at a significant distance from the surface can be readily and efficiently tetrahedralized using a pre-defined, combinatorial stencil, with excellent quality. Challenges, however, emerge when dealing with grid cells situated near the boundaries.

Have been proposed several approaches to deal with these challenges, such as:

- *Molino et al. 2003*[6]: the cells intersecting the domain boundary are pushed into the domain via physics-based simulation;
- *Bronson et al. 2012*[1]: boundary cells can be cut into smaller pieces
- *Labelle & Shewchuk 2007*[4]: snap vertices to the input surface and cut crossing elements

And many others.

Delaunay-based Methods There is an entire class of methods for tetrahedralization that are based on the Delaunay triangulation of the input geometry and are very well-studied [2, 9]. These methods are efficient and scalable, but when the input includes surface mesh constraints, the challenge is to extend the notion of a Delaunay mesh in a meaningful way. A simple and elegant idea of Delaunay refinement was proposed by [3, 7, 10], and it consists of inserting new vertices at the center of the circumscribed sphere of the worst tetrahedron measured by radius-to-edge ratio. This approach guarantees termination and provides bounds on the radius-edge ratio.

Variational Meshing Variational techniques are also based on the Delaunay triangulation and, on its dual, the Voronoi Diagram. The idea is to cast the problem as one of minimizing energy that measures the quality of the mesh. Low levels of this energy correspond to high-quality meshes. An example is the relaxation through Lloyd's algorithm, which moves each point to the centroid of its Voronoi region; in this way, when the method converges, we obtain a regular meshing.

Surface Envelope Explicit envelopes have been used to guarantee a bounded approximation error in surface reconstruction.

Chapter 3

The Algorithm

The goal of this thesis is to implement an algorithm for generating tetrahedral meshes from a surface mesh. The starting idea was taken from the work "*Competing Fronts for Coarse-to-Fine Surface Triangulation*"[8], which proposes an algorithm for generating surface meshes from a point cloud. The concept is to extend and adapt this approach to the volumetric case, starting from a surface mesh and generating a tetrahedral mesh in a fully automatic fashion.

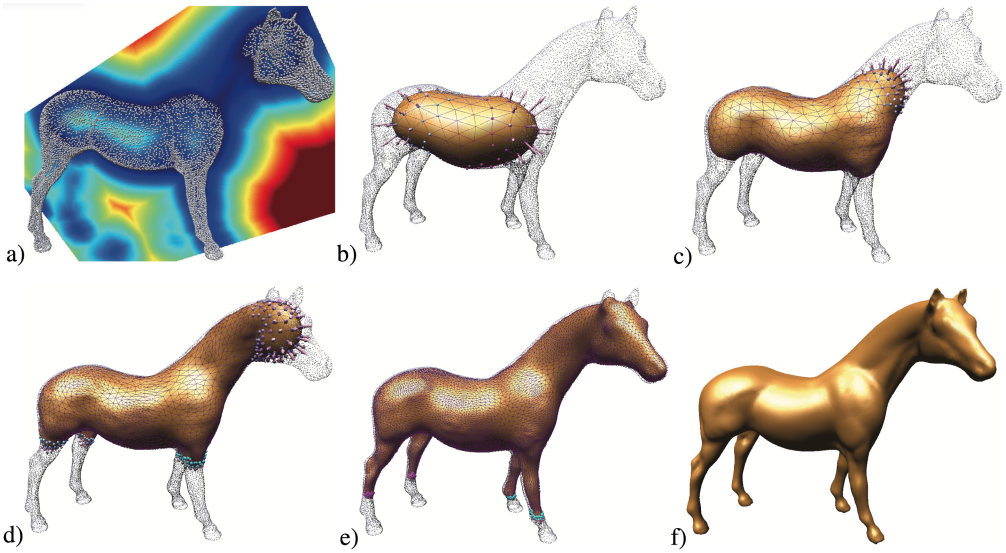

Figure 3.1: *"Competing Fronts for Coarse-to-Fine Surface Triangulation" pipeline*

In their work, Sharf et al. manually place a coarse triangulation of a sphere within the point cloud, after which, using an advancing front approach, they expand to fill the entire space implicitly defined by the point cloud. The pipeline of their work is shown in the Figure 3.1.
The expansion of the triangulation is done through a distance field calculated on

the point cloud, which allows the direction and speed of expansion of the sphere to be determined. A distance field is a function that, given any point in space, returns the minimum distance between the point and the elements of the domain on which the distance field was constructed, in this case, the point cloud. We can see an example of distance field in Figure 3.3.a.
Then, constraints are used during the expansion to avoid forming low-quality triangles and maintaining a uniform structure.

In the extension to volumes of this algorithm, the entire internal structure must be handled. We will no longer have to worry only about the surface, but we will also have to make sure that the polyhedra that make up the mesh remains acceptable.
In the approach proposed in this thesis, we start with a target surface mesh that must be manifold, watertight, and genus zero, then calculate the distance field within the mesh target. The distance field is used to automatically compute the optimal point for automatically positioning the starting tetrahedral object (typically a coarse approximation of a sphere or a single tetrahedron) within the surface mesh. Once positioned, the tetrahedral object is deformed until it fills the entire volume. During deformation, we try to maintain a good shape of the tetrahedra using topological operators and to ensure conformity to the target surface mesh.

3.1 Main steps

The algorithm is structured into three main ingredients:

1. Initialization
2. Advancing
3. Refinement and quality preservation

In the initialization step, the distance field is constructed, the initial object within the target surface mesh is initialized and positioned, and the problem constraints are defined. In the advancing phase, we divide the parts of the initial object that we have to expand into fronts; then, we expand them until the model fills the entire volume of the target mesh. In the refinement and quality phase, topological operators are used to keep the element shape quality of the mesh high and to add elements within the model.
The algorithm is structured in such a way that it can be parallelized whenever possible, as operations involving one front are independent of the others. After each operation, we make sure that the constraints defined at the beginning of the

problem are satisfied; if not, one looks for a feasible intermediate state between the original and the next or restores the previous state.
The expansion and quality maintenance steps both succeeded by smoothing the surface and volumetric mesh. Smoothing consists of deforming individual mesh elements so that they have a better shape and also form fewer irregularities in the overall mesh. This step allows the mesh to be regularized and reduces the probability that subsequent steps will fail.
The advancing and refinement steps are repeated iteratively until all the volume is filled.

3.2 Input and starting tetrahedral object

As anticipated earlier, the algorithm takes a starting surface mesh, which must be manifold, watertight, and genus zero as input. These conditions are necessary to ensure correct generation and conformity to the target surface mesh. This input surface mesh becomes our target mesh T .

We compute the distance field $D(x)$ within the target mesh. As mentioned earlier, the distance field is a function that, given any point in space, returns the minimum distance between the point and the domain on which the field is constructed, in our case, the target mesh T.
The distance field allows us to search for the interior point with maximum distance from the surface, then choose this point as the optimal starting point s to place the tetrahedral starting object M.
The distance field is computed with the help of an octree data structure, which allows us to search between vertices and compute distances efficiently.
The best starting object is typically a coarse tetrahedral sphere, but any other tetrahedral object with the same topology (e.g., a single tetrahedron) can be used.

Figure 3.2: ***a)*** *Input model* ***b)*** *Distance field* ***c)*** *Coarse sphere placement*

The starting mesh M is placed so that it is centered in s, then it is scaled so that the diagonal of its bounding box is equal to $D(s)$, so the two surfaces will be at least half as far apart as $D(s)$. This measure, taken arbitrarily, allows for a starting object that has a size proportional to the model we are going to fill to avoid having a mesh that is too coarse or too fine.

3.3 Advancing Volumes

Once the mesh M is placed, it is deformed to fill the volume of the target mesh T. The goal is to fill the entire volume, so we try to expand the mesh in all directions in order to iteratively bring the surface vertices as close to the target surface as possible. Surface vertices are classified into active and inactive vertices according to their distance from the target surface. Given a threshold d, inactive vertices are those that lie on the target surface or have a distance $D(v) < d$, so they have a position that we consider good, while active vertices are those that are at a distance $D(v) > d$, they therefore have a position that we consider not good and that we want to improve by moving them closer to the target surface. When all the vertices are inactive, they are projected onto T for a better approximation of the shape. The threshold d is defined in function of a target edge length l introduced in the 3.4 section. The target edge length l defines the ideal length that all the mesh edges should have.

The expansion is done through an advancing fronts approach, which expand in a competing manner. By "competing manner," we mean that the various fronts move independently and, for this reason, can also move simultaneously. A front is a connected component of active vertices, so a front is a set of adjacent vertices at a distance $D(v) > d$ and moving in the direction of the target surface. While fronts are independent and can expand concurrently, vertices of the same front are dependent and move sequentially, so a vertex only moves when the previous vertex has finished its movement.

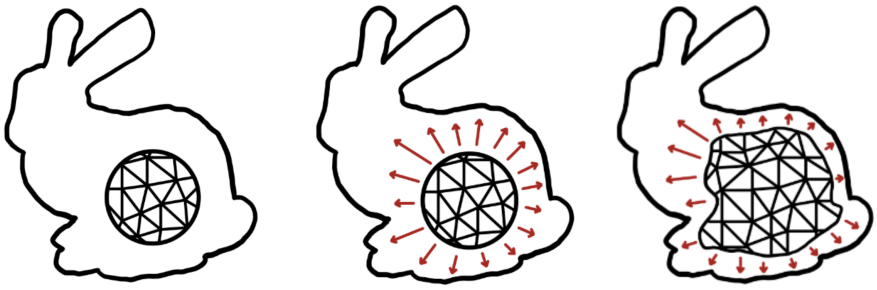

Figure 3.3: ***a)*** *Initial state* ***b)*** *Movement vectors* ***c)*** *Advancing volumes*

The motion of each active vertex is characterized by a direction and a speed, as shown in Figure 3.3.b and 3.3.c. The direction is equal to the vertex normal, which is defined as the average of the normals of adjacent surface faces. Speed is proportional to the distance of the vertex from the target surface T, $D(v)$, so the further a vertex is from T, the faster it moves.

When a vertex reaches a distance $D(v) < d$, it is a candidate for becoming inactive. To confirm inactivity, the distance is recalculated no longer with the distance field but with a raycast that starts at the vertex and has a direction equal to the vertex normal. The distance $D(v)$ is a good proxy, but it is a lower bound and not the actual distance the vertex must travel to reach its destination. A raycast consists of drawing a half-line, giving a starting point and a direction, and finding the various points of intersection with the half-line. We will take the first point of intersection of the raycast with the target surface T, so the distance of the vertex from T will be equal to the distance between the vertex and the point of intersection. The difference between the two is shown in Fig. 3.4.

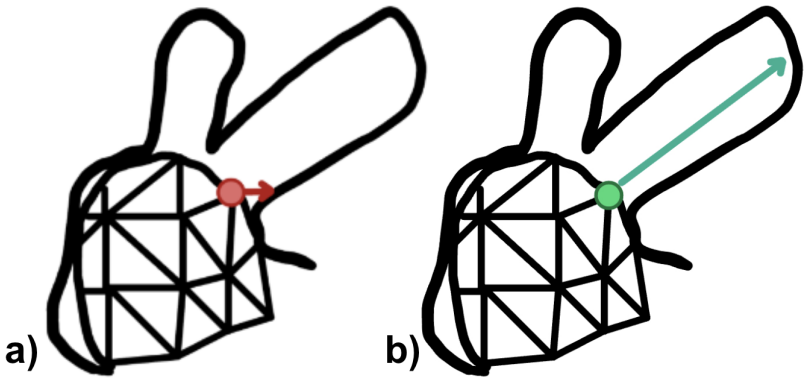

Figure 3.4: ***a)*** *$D(x)$ distance* ***b)*** *Raycast distance*

If the distance calculated with the raycast is less than d, the vertex becomes inactive, otherwise, we continue to expand the front by placing a velocity proportional to the raycast-calculated distance until we can use the distance field again. This latter aspect prevents the vertex from stopping at the entrance of mesh choke points, which could stop the front from expanding properly. It is preferred to use the distance field to calculate the distance in "common" iterations because it allows the front to expand more uniformly, as adjacent points have similar distances, whereas raycast could lead to very different distances between adjacent points, therefore raycast is used only to confirm that the vertex is actually close to the target surface and, in case it is not, to unlock its expansion.

Whenever a vertex is moved, we ensure that we do not invert tetrahedra orientation and we have no intersections between target surface T and the tetmesh M. In particular, we check that the vertex does not move in such a way that any of its adjacent tetrahedra are flipped, hence, we check that the volume of the tetrahedron does not change sign. Another important thing to check is that there are no self-intersections in the model, so we check that the vertex does not move in a manner such that one tetrahedron intersects another tetrahedron of the mesh, which, in addition to breaking the mesh, would lead to a situation where the algorithm would get out of control. Finally, we check that the vertex umbrella does not intersect the target surface, so we check that there are no intersections between T and the edges/faces incident on the vertex.
If one of these checks fails, we bring the vertex back by half the distance that it moved, with the intent of looking for an intermediate position where the mesh is correct. We continue to halve the move until we position the vertex in an acceptable position where all the checks are passed. This tactic of backwarding by bisection is applied until a correct position is found or until a maximum number of attempts is reached, at the moment in which the operation is canceled and the previous state of the vertex is restored.

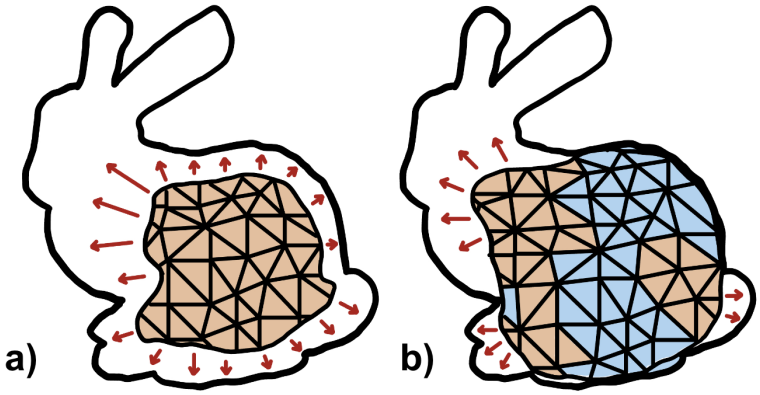

Figure 3.5: *Approaching the target surface, the vertices become inactive*

At the beginning of the algorithm, we have only one front and all surface vertices are part of it, while no vertex is inactive. As the tetmesh expands, more and more vertices will reach their optimal position and become inactive, so it could happen that one vertex by becoming inactive, splits a front into two or more subfronts. Figure 3.5 shows an example of this process.
A flooding algorithm is used to update the fronts, which allows us to calculate the connected components of active vertices after each advance.

3.4 Refinement and quality preservation

During the advancement, it is possible for mesh elements to deform in such a way that they have low shape quality or their vertex remain stuck, and the model can no longer advance, so topological operators are used to refine the model. Topological operators are operators that change the connectivity of the mesh, so they act on the link between the various vertices of the mesh. In particular, we use local topological operators since they change the internal topology of a cluster of tetrahedra adjacent to the element targeted by the operation while keeping the topology to the outside of the cluster unaffected in order to avoid undesirable propagation and cascading effects. To avoid excessive stretching of tetrahedra, a target edge length l is defined. As anticipated earlier, this threshold l represents the ideal length that an edge should have, so if an edge exceeds this length, it will be split.

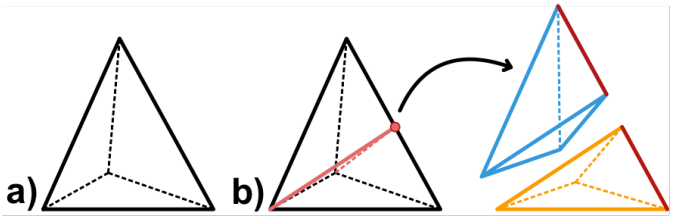

Figure 3.6: *Edge split:* ***a)*** *Starting tetrahedron* ***b)*** *Edge splitted*

The "edge split" operation is used to halve the length of an edge and consists of inserting a vertex in the midpoint of the original edge e, which is then connected with an edge with the opposite vertices of e on each face, in this way, the correctness of the mesh is preserved.
From an implementation point of view, for each tetrahedron adjacent to the edge, we take the two faces opposite to the edge and use them as a basis of the two new tetrahedra we want to create, then we connect all the faces with the midpoint of the edge we are going to break in order to create two new tetrahedra. Once all the new tetrahedra are made, we delete the original tetrahedra. This operation makes it possible to split a tetrahedron into two smaller tetrahedra with a better edge length, in contrast, it introduces suboptimal angles, as each new edge inserted makes a bisection of the original opposite angle. For this reason, we try to preserve the original angles, so we are going to split all the tetrahedra incident on the edges that exceed the target edge length l. The tetrahedra are split according to the pattern shown in Figure 3.7.h.
To achieve this, once the edge that needs to be split is identified, all adjacent tetrahedra are obtained, and all their edges are split. This operation leads to a suboptimal pattern (Fig. 3.7.g) that can be adjusted with some edge flip operators.

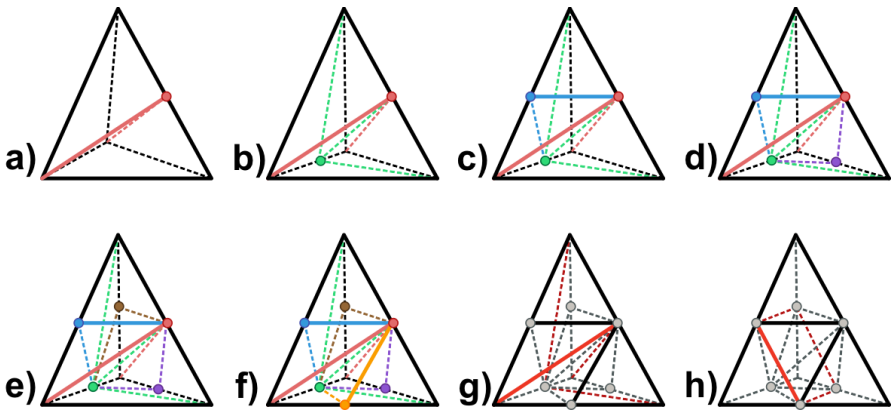

Figure 3.7: *Split steps:* ***abcdef)*** *Iteratively all edges are split* ***g)*** *The edges we want to flip are highlighted in red* ***h)*** *Final result with flipped edges*

In particular, we talk about flip 2-2 and flip 4-4, where the first number represents the number of starting tetrahedra, and the second number represents the number of tetrahedra present in the cluster after the flip operation (there are also other flip operators, such as the 3-2 ones). The flip operators allow us to shift the incidence of edges between vertices to balance it out, specifically, we want to avoid new edges incident on the original vertices of the cluster, and all new edges must be incident on vertices inserted during the edge split. This allows us to keep the original corners and have better tetrahedron shapes.

The flip 2-2 operators are used for edges that lie on the surface and can be used only when the two surface faces adjacent to the edge are coplanar, otherwise, the geometry of the cluster is altered, but we guarantee this property by construction since the two tetrahedra involved are the result of an edge split and these faces are two parts of the same original face. To perform this flip, on the two adjacent surface faces we identify the two vertices opposite the edge we want to flip, then we remove the edge to be flipped and insert an edge to connect the two previously identified vertices.

The flip operators 4-4 are used for internal edges. This type of flip, in addition to being more complex, is not unique, so it adds a layer of complexity in figuring out which of the two flip configurations must be used. The Figure 3.9 shows the two possible flips. To figure out which two vertices to join should be chosen, we must first identify the vertices that composed the edge that was split from the edge we want to flip. The vertices we want to connect to perform the flip are the only two that are adjacent to both the vertices of the original edge and the vertices of the edge to be flipped. Once the two vertices to be connected have been identified, we proceed in a similar manner to flip 2-2 by removing the edge and inserting the

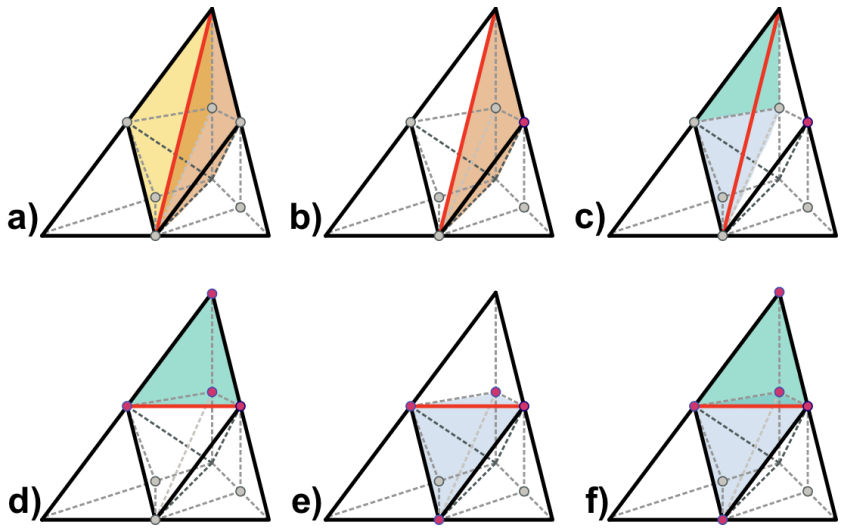

Figure 3.8: *Flip 2-2 steps:* ***a)*** *The edge to be flipped and its two adjacent tetrahedra are identified* ***b)*** *One of two opposite vertices on the surface is chosen* ***c)*** *The two opposite faces from the other tetrahedron are identified* ***de)*** *The two new tetrahedra are created joining the two faces with the vertex* ***f)*** *Final results*

new one.

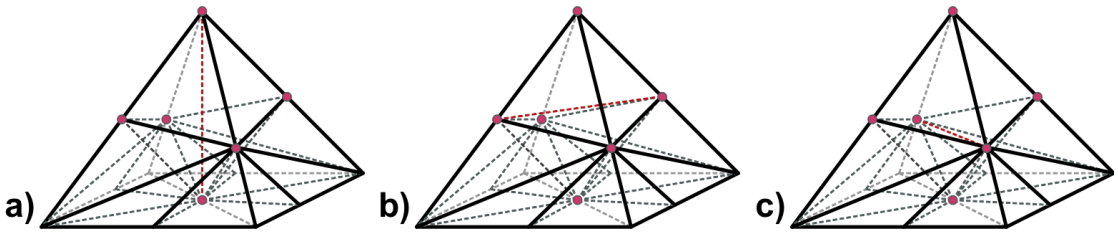

Figure 3.9: *Flip 4-4:* ***a)*** *The edge that we want to flip is highlighted in red* ***bc)*** *The two possible flips, we want the one shown in Figure C*

Depending on the order in which we perform the split and flip operations, we may have situations in which the edges to be flipped are not immediately flippable. If they are on the surface, they have more than 2 adjacencies, and if they are internal, they have more than 4 adjacencies. In these cases, we postpone the flip, as it may be that a successive flip will make the edge in question flippable. The flip is postponed until the number of adjacencies is correct or until the number of adjacencies has decreased from the previous attempt; if the number of adjacencies is the same for two consecutive attempts, the edge is declared unflippable and the next edge is flipped.

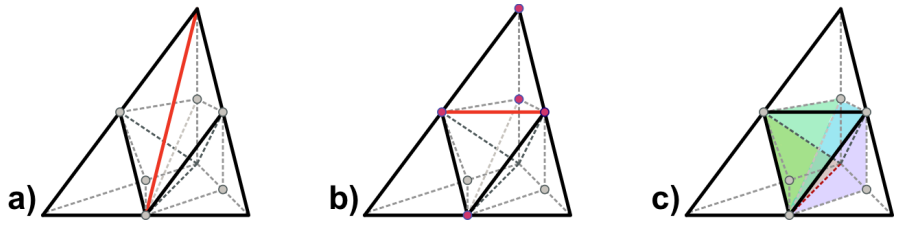

Figure 3.10: ***a)*** *The red edge is selected to be flipped* ***b)*** *The edge is flipped* ***c)*** *Now we cannot perform a 2-2 flip on the edge highlighted in red because it is adjacent to 4 tetrahedra, so the flip is postponed*

3.5 Smoothing

After each front advance and each mesh refinement, a smoothing of the mesh is performed. When the surface vertices advance, their incident tetrahedra are stretched, and their shape gets worse. Smoothing allows the mesh to be regularized in terms of shape and reduces the chances that later passes will fail due to some tetrahedra close to collapsing which blocks its vertices from moving. The smoothing applied is the Laplacian smoothing, so we move each vertex to the average of adjacent vertices. Smoothing on the surface is done by taking into account only the adjacent vertices on the surface, this is done to contain the phenomenon of "shrinkage," because if you consider the interior vertices as well, they tend to pull the surface inward, so you run the risk of having a surface that shrinks and fails to fill the entire space. Another expedient to mitigate this phenomenon is to make a projection of the surface vertices onto the pre-smoothing surface, so the volume does not regress too much. Internal vertices, on the other hand, take into account all adjacent vertices, both on the surface and internal, as there is no risk of them contracting, therefore, more aggressive smoothing can be done.

For each smoothing step, the same checks are applied as are applied when advancing the fronts, so we check that the constraints of the mesh are maintained, that no intersection is occurring, and that we are not going to flip the volume of any adjacent tetrahedron. On every vertex smoothing movement, if one check is not passed the same policy of backwarding by bisection is applied until a correct state is reached or the maximum number of iterations allowed is reached, after which the vertex is returned to its starting position.

Laplacian smoothing of the mesh is to be considered as temporary and not definitive, since it gives good results but not the desired ones in terms of mesh shape regularization. We are able to bring the innermost tetrahedra close to equilaterality, but the closer we get to the surface, the closer the tetrahedra gets to becoming

degenerate. The limitations of this smoothing can lead to problems in advancing the fronts in some cases since, given non-regular or rippled target surfaces, it can happen that the fronts tend to crumple and try to collapse into each other near the target surface.

3.6 Convergence

The algorithm is considered done when all vertices are inactive and we have no more active fronts. As a final step, when there are no more active fronts, a projection of all vertices onto the target surface is made to ensure the closest possible approximation of the target shape.

However, due to some of the above-mentioned implementation limitations, it may be that some fronts fail to converge within a reasonable number of iterations, so it was decided to set a maximum number of iterations of fronts progressing, in order to prevent the algorithm from going into an infinite loop because it fails to fill the volume as expected. Another reason for the sudden stop, as mentioned above, is the insufficiency of Laplacian smoothing and the consequent crumpling of the fronts, which could lead to self-intersection of the fronts and the consequent error of the algorithm.

Nevertheless, failure cases are marginal, and in most cases the algorithm converges correctly and produces a good-quality tetmesh.

Chapter 4

Discussion and Results

This project was implemented in C++ using CinoLib[5], a library that provides data structures and algorithms for surface and volume mesh manipulation.

Model	Iterations	Time
cubespikes	64	70.88s
armadillo	316	225.27s
spider	617	220.59s
ant	277	40.15s
hand_olivier	353	298.58s
sphere	44	110.30s
femur	269	107.92s
buste	161	160.95s
bone	241	87.55s
foot	135	118.29s
duck	92	138.88s
lion	1000	493.59s
armadillo_deformed	1000	453.92s
camile_hand	239	213.98s
dilo	642	153.35s

Table 4.1: *Time and iterations of converged models*

The algorithm was tested using a comprehensive set of 49 meshes of diverse types and sizes. All the tests were performed on a workstation with Ubuntu 18.94 LTS operating system, Intel I9-7920X processor, and 128 GB of RAM. Within the evaluation, the algorithm successfully completed the mesh generation in 11 cases. For two of the meshes, even though the algorithm's maximum iteration limit, set empirically at 1000 after multiple attempts, was reached, it was insufficient to achieve convergence. Nevertheless, these results are still considered acceptable

approximations. The algorithm in the remaining meshes failed by producing self-intersection; this is due, as anticipated, to the insufficiency of Laplacian smoothing in regularizing the mesh surface. In the tests, the max edge length parameter l was set to 2% of the diagonal of the bounding box of the input surface model, while the idle threshold is always 10% of the previous parameter, so in our case, it is 0.2% of the diagonal of the bounding box. As we can see from Table 4.1, when the algorithm converges, it takes an average of 2 minutes to generate the mesh. Figure 4.1 shows a sequence of how the algorithm advances the fronts.

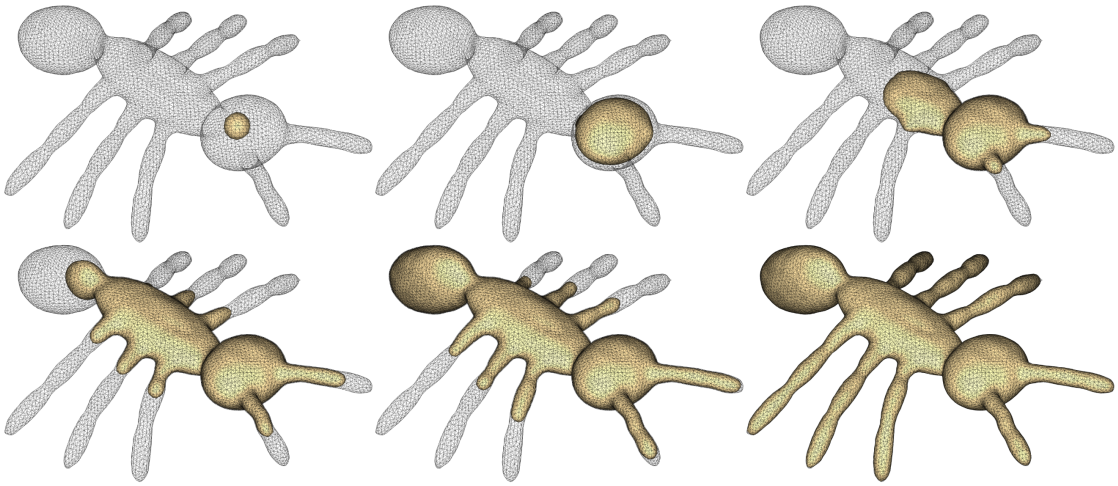

Figure 4.1: *Advancing sequence of the proposed algorithm*

The quality can be described with the Scaled Jacobian. This metric informs us about how much a tetrahedron diverges from the canonical (equilateral) tetrahedron, ranging between 0 and 1.
From the data shown in Table 4.2, we can see that the minimum is always an ϵ and is due to the shape of the tetrahedra closest to the surface, as the Laplacian smoothing fails to "pull" inside the inner vertices of the surface tetrahedra after a refinement step has been performed. In the current state of the project, smoothing is a major limitation for the quality and convergence of the algorithm.
Despite being unable to keep a very high quality of the tetrahedra, the shape approximation is done with acceptable accuracy. The Hausdorff Distance, which takes the most significant distance between the target surface and the surface of the generated model, was used to measure the accuracy of the approximation. This distance is expressed as a percentage of the diagonal of the bounding box. As we can see from the low values in Table 4.3, when the algorithm converges, we are able to fill the volume almost totally.

Model	Avg SJ	Min SJ	Max SJ
cubespikes	0.34	1×10^{-16}	0.97
armadillo	0.39	8×10^{-16}	0.96
spider	0.45	1×10^{-16}	0.97
ant	0.44	5×10^{-18}	0.97
hand_olivier	0.35	2×10^{-15}	0.98
sphere	0.30	2×10^{-18}	0.97
femur	0.39	2×10^{-16}	0.97
buste	0.33	2×10^{-15}	0.96
bone	0.39	2×10^{-17}	0.97
foot	0.31	3×10^{-15}	0.97
duck	0.33	8×10^{-17}	0.98
lion	0.38	5×10^{-15}	0.97
armadillo_deformed	0.40	2×10^{-16}	0.95
camile_hand	0.35	2×10^{-14}	0.98
dilo	0.36	9×10^{-17}	0.97

Table 4.2: *Scaled Jacobian of converged models*

Model	Avg dist.	Min dist.	Max dist.
cubespikes	2×10^{-3}	1×10^{-3}	5×10^{-3}
armadillo	2×10^{-3}	0	1×10^{-2}
spider	2×10^{-3}	6×10^{-5}	5×10^{-3}
ant	3×10^{-3}	1×10^{-4}	2×10^{-2}
hand_olivier	2×10^{-3}	0	1×10^{-2}
sphere	2×10^{-3}	2×10^{-3}	3×10^{-3}
femur	4×10^{-4}	9×10^{-3}	2×10^{-3}
buste	2×10^{-3}	9×10^{-5}	9×10^{-3}
bone	2×10^{-3}	9×10^{-4}	5×10^{-3}
foot	2×10^{-3}	4×10^{-4}	5×10^{-3}
duck	2×10^{-3}	2×10^{-4}	9×10^{-3}
lion	2×10^{-3}	0	2×10^{-2}
armadillo_deformed	2×10^{-3}	0	1×10^{-2}
camile_hand	2×10^{-3}	0	1×10^{-2}
dilo	2×10^{-3}	1×10^{-5}	1×10^{-2}

Table 4.3: *Distance between M and T in converged models*

Some examples of successfully generated tetmeshes are shown in Figure 4.2.

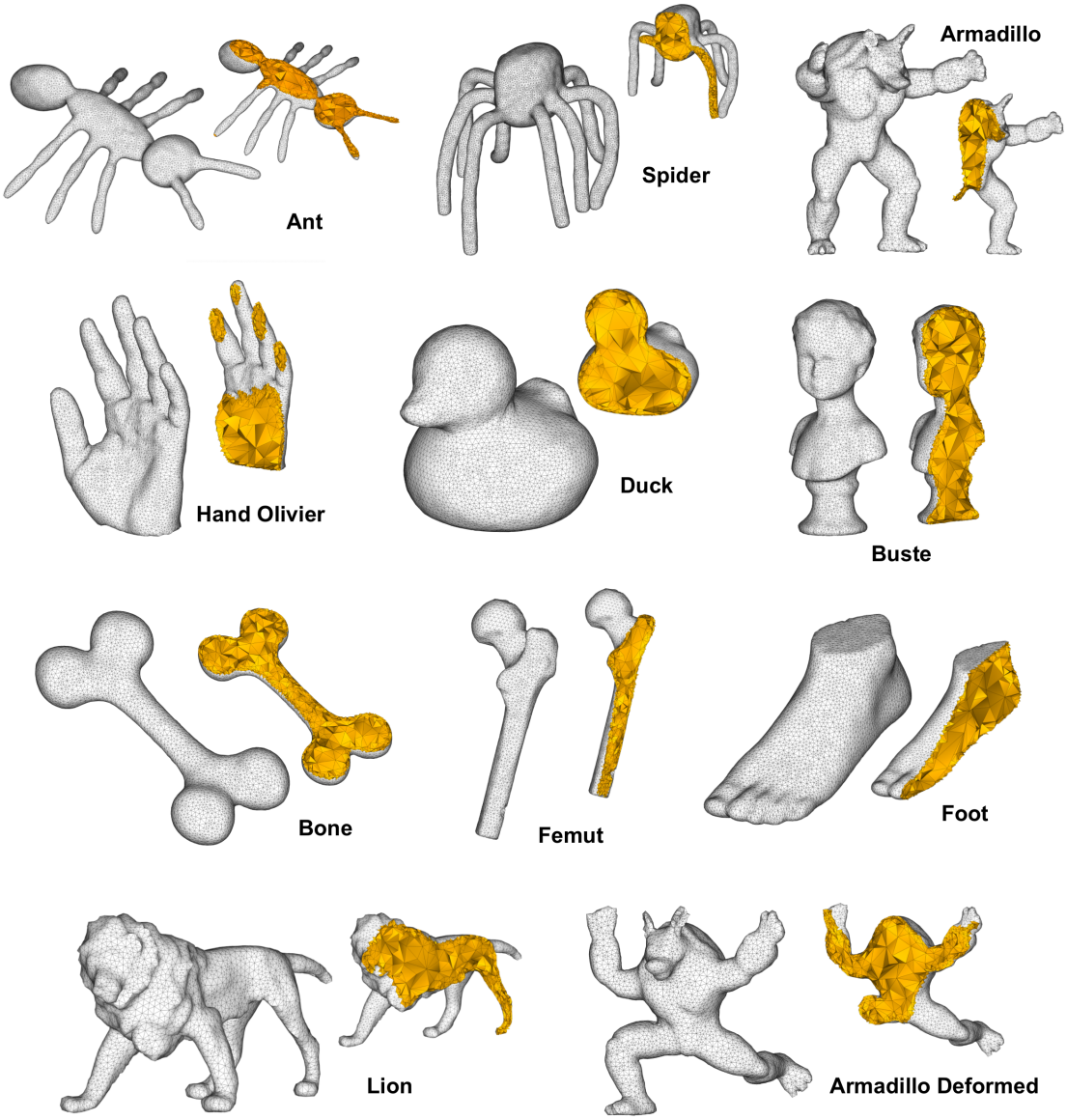

Figure 4.2: *Some successfully generated tetmeshes*

As we can see, the inner tetrahedra present good shape quality, in contrast to the outer ones which are more flattened.

In Figure 4.3 are some cases of failure, with emphasis on self-intersection points:

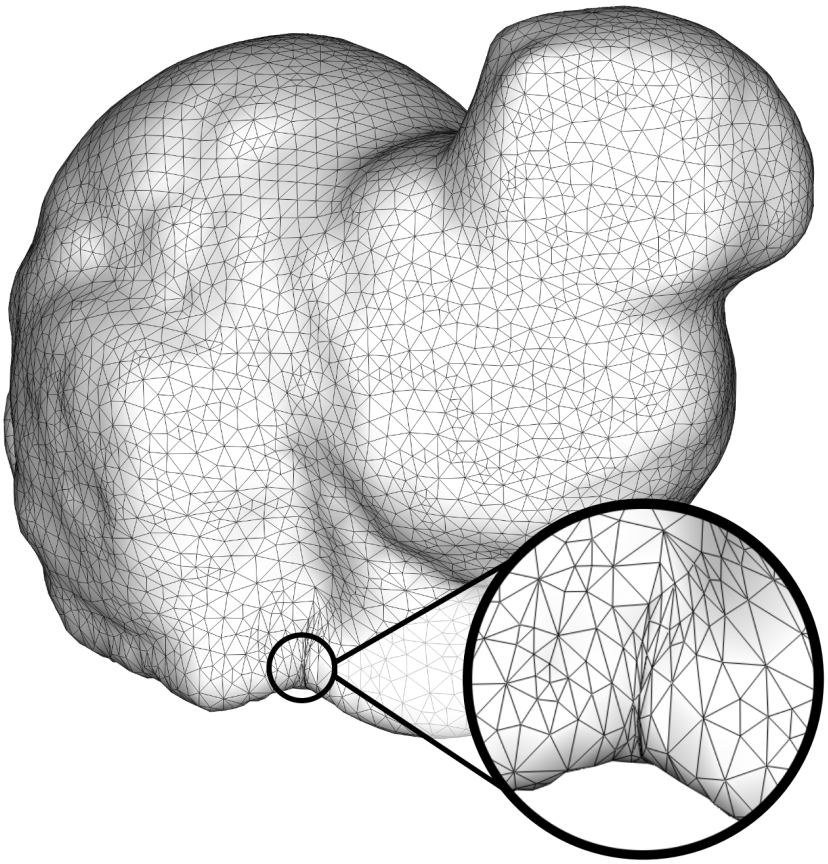

Figure 4.3: *Example of self-intersection*

As formerly discussed, this problematic is a consequence of the smoothing that is not able to polish enough the surface.

Chapter 5

Future Works and Conclusions

The work done in this thesis is the initial stride towards a novel approach for automatically generating tetrahedral meshes from a surface mesh of any topological genus.
In particular, the work done in this thesis focuses on generating tetrahedral meshes from a target surface mesh. The proposed algorithm has been tested on 49 surface models of varying nature and complexity, obtaining satisfactory results.
The limitation mentioned in this thesis must be solved in future improvements. First of all, it is necessary to investigate a better smoothing algorithm able to regularize the mesh and reduce the probability of subsequent steps to fail.
Specifically, the future developments can be divided into two main lines:

- The first one is about extending the algorithm to generate tetrahedral meshes from a surface mesh with a genus greater than zero; this would allow tetrahedral meshes with holes to be generated and make the algorithm more flexible;
- The second line of work concerns incorporating this algorithm within a much larger picture, which involves the creation of bijective maps between two volumetric objects, as this algorithm is suitable for simultaneous executions. By being run at the same time between two models, the same refinement steps can be carried out in both models, so the same topology is preserved, and the map bijectivity is maintained, while the front advance steps would be independent between the two models. This latter aspect would allow for a bijective map between two volumetric objects, which is an open problem.

Beyond the stated limitations, the results obtained are promising, leaving much room for improvement.

Bibliography

[1] Jonathan Bronson, Joshua Levine, and Ross Whitaker. Lattice cleaving: Conforming tetrahedral meshes of multimaterial domains with bounded quality. volume 2013, 10 2012.

[2] S.W. Cheng, T.K. Dey, and J. Shewchuk. *Delaunay Mesh Generation.* Chapman & Hall/CRC Computer and Information Science Series. CRC Press, 2016.

[3] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. SCG '93, page 274–280, New York, NY, USA, 1993. Association for Computing Machinery.

[4] François Labelle and Jonathan Richard Shewchuk. Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles. *ACM Trans. Graph.*, 26(3):57–es, jul 2007.

[5] Marco Livesu. Cinolib: A generic programming header only c++ library for processing polygonal and polyhedral meshes. *Trans. Comput. Sci.*, 34:64–76, 2019.

[6] Neil P. Molino, Robert Bridson, and Ronald Fedkiw. Tetrahedral mesh generation for deformable bodies. 2003.

[7] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995.

[8] Andrei Sharf, Thomas Lewiner, Ariel Shamir, Leif Kobbelt, and Daniel Cohen-Or. Competing fronts for coarse–to–fine surface reconstruction. *Comput. Graph. Forum*, 25:389–398, 09 2006.

[9] Donald Sheehy. New bounds on the size of optimal meshes. *Computer Graphics Forum*, 31:1627–1635, 08 2012.

[10] Jonathan Richard Shewchuk. Tetrahedral mesh generation by delaunay refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational*

Geometry, SCG '98, page 86–95, New York, NY, USA, 1998. Association for Computing Machinery.

List of Figures

List of Tables

Ringraziamenti