3D Snap rounding

Léo Valque

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

3D Snap Rounding

THÈSE

présentée et soutenue publiquement le 13 décembre 2024

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Léo Valque

Composition du jury

Présidente :	Marie-Odile Berger (INRIA Grand-Est, LORIA)
Rapporteuse et rapporteur :	Julie Digne (CNRS, LIRIS)
	Raimund Seidel (Universität des Saarlandes)
Examinateurs et examinatrice :	Sébastien Loriot (Geometry Factory)
	André Leutier (Retraité de Dassault Systèmes
	Marie-Odile Berger (INRIA Grand-Est, LORIA))
Directeur de thèse :	Sylvain Lazard (INRIA Grand-Est, LORIA)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thesul.

Remerciements

Je tiens à exprimer ma profonde gratitude envers toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de cette thèse de doctorat.

Tout d'abord, je remercie chaleureusement Sylvain, mon directeur de thèse, pour son accompagnement, ses précieux conseils et son soutien constant tout au long de ces années. Ton enseignement, ta disponibilité et ta patience ont été essentiels à l'avancement de nos travaux.

Je souhaite également remercier tous les membres de l'équipe Gamble pour m'avoir tenu compagnie toutes ces années et plus spécifiquement à tous les non-permanents qui sont passés dans l'équipe. Merci à Nuwan, Florent et Léo pour la bonne camaraderie, à Camille pour l'aide mutuel sur la thèse et aux nouveaux arrivant Sarah, Marguerite et Dorian pour perpétuer cette belle ambiance.

Je remercie également tous mes nombreux amis en dehors du laboratoire, particulièrement Margot, Léah, JP et Valentin, pour les moments partagés loin de la recherche et votre soutien lors de cette période parfois compliqué.

Enfin je n'oublie pas ma famille, particulièrement ma soeur pour leur soutien inconditionnel et prolongé. Merci d'avoir traversé les kilomètres pour assister à cette soutenance.

Cette thèse est le fruit d'un long effort, et chacun d'entre vous a joué un rôle important dans ce cheminement.

Merci à tous.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

> "Solving the snap rounding problem is the ultimate solution for all these issues, but this is a remarkably difficult problem..."
>
> Cherchi et al., 2022 [6]

In computer science, geometric algorithms are predominantly described using the real RAM model [3]. In this model, each memory unit can store a real number, and accessing any memory unit requires constant time, independently of its location. All standard operations (addition, subtraction, comparison, etc.) are also performed in constant time with exact results. This model is practical for describing algorithms without addressing the numerical accuracy limitations inherent to real computers. However, the real RAM model is purely theoretical, and any actual implementation must represent geometric objects using a finite number of bits, leading to finite precision.

Many implementations that work with 3D polygonal objects, both in academia and industry, require input polygons that are pairwise-disjoint, with vertex coordinates given in fixed-precision (typically 32 or 64 bits). When geometric algorithms generate new objects from such input, their exact representations often require more bits for accurate representation than the original input. For instance, the intersection of two line segments with b-bit integer coordinates at their endpoints results in an intersection point whose exact rational coordinates require in general $5b + O(1)$ bits (see Section 2.2). Similarly, the intersection point of three triangles in 3D require in general rational coordinates of $13b + O(1)$ bits (see Section 3.2.2). Applying a rotation to a polyhedron introduces new vertices with coordinates involving trigonometric functions, while

sampling algebraic surfaces yields vertices as solutions to algebraic systems, that can be arbitrarely close to one another (depending on the degree of the surface). This discrepancy between the precision of input and output is problematic, especially in industry, as it often prevents the direct use of an algorithm's output as input for subsequent algorithms. Consequently, rounding 3D polygonal structures between operations is crucial, making it a fundamental problem in both computational geometry and computer graphics.

Most implementations of geometric algorithms round their output to the same precision as their input, typically using single or double precision floating-point numbers. In these implementations, rounding is often performed directly by the processor's floating-point unit, following the IEEE 754 standard [1]. In this standard resulting numbers are rounded to one of the nearest floating-point numbers after each operation.[1]

However, when rounding operations are performed using the IEEE standard or other "naive" methods, the topological properties of geometric objects may be broken. For instance, as illustrated in Figure 1.1, a triangle and a quadrilateral may intersect after rotation and rounding, even though they were disjoint prior to these operations. Such issues are even more significant for 3D objects, where rounding can create self-intersections within polygonal structures, making them unusable for many algorithms. Numerous studies in computer graphics [2, 5, 6, 9, 19, 20, 31] have highlighted the significant challenges posed by rounding, emphasizing its importance in algorithm and software development. Intersection-free rounding of 3D polygonal structures is thus fundamental in computational geometry, computer graphics, and in the industry.

Some use exact arithmetic in geometric computations to address these issues, such as Pion and Fabri [27], as implemented in libraries like the Computational Geometry Algorithms Library (CGAL) [28]. Despite this, rounding is still often necessary due to the exponential growth in representation size and practical considerations like storage and diffusion of results. Indeed, when geometric constructions are cascaded, where one algorithm's output is used as input to another, the precision required for accurate representation can quickly become unmanageable. For instance, cascading k line segment intersections would require $5^k(b + O(1))$ bits to represent the final output accurately. Additionally, some operations, such as trigonometric functions or

[1] The IEEE 754 standard defines five rounding modes: rounding to the nearest lower value ("floor"), rounding to the nearest higher value ("ceil"), rounding to the floor or ceil nearest to zero ("trunc"), and two other modes for rounding to the nearest value, distinguished by their handling of ties: rounding to the value with a null last digit ("ties to even") and rounding to the value further from zero ("ties to away").

roots, cannot be exactly represented with rational numbers, requiring complex extensions for accurate description. Therefore, the number of feasible cascaded constructions without rounding is limited and again, as most algorithms take as input fixed precision floating point coordinate, exact unrounded output can rarely be used as input in other algorithms.

As a response, the development of algorithms to round the coordinates of geometric points while preserving the topology of the output has been the focus of a lot of attention. Such algorithms must ensure that the output remains close to the input, with an adequate definition of closeness. For example, the rounded result does not always have to retain the highest precision but must accurately reflect the combinatorial structure of the original result, albeit with lower precision. This problem, known as the Snap Rounding Problem, involves rounding geometric inputs (often vertices of segments or triangles) while maintaining certain topological properties. The problem in two dimensions has been extensively studied since the late 1980s [7, 13–16, 18, 22], culminating in a stable 2D snap rounding algorithm by Hershberger in 2013 [17].

However, despite the significance of the problem, 3D snap rounding results are notably scarce. In the 1990s, two algorithms were proposed [12, 23], but both were later found flawed by Fortune [11], who introduced two alternative algorithms [10, 11] that still did not really resolve the problem (see Section 3.3). Milenkovic and Sacks [25] presented an algorithm in 2019, but without guarantees of closeness between input and output. It was not until 2020 that Devillers et al. [8] proposed a theoretical solution to 3D snap rounding, although its complexity renders it impractical (see Section 3.3).

Given the lack of theoretical solutions, computer graphics researchers have developed heuristics to solve the problem. The current state of the art is Zhou et al.'s paper [31], which successfully addresses most, but not all, of the rounding issues without any termination or topology guarantees (see Section 3.3).

Building on the work of Devillers et al. [8], this thesis presents the first practical, and certified algorithm to solve the 3D snap rounding problem. The first part of this thesis reviews the state of the art in 2D and 3D snap rounding and highlights the associated difficulties. The second part details our proposed 3D snap rounding algorithm, followed by a description of its implementation in the third part. The fourth part presents the results of our experiments. The fifth part presents a new uncertified but very efficient heuristic to round vertices.

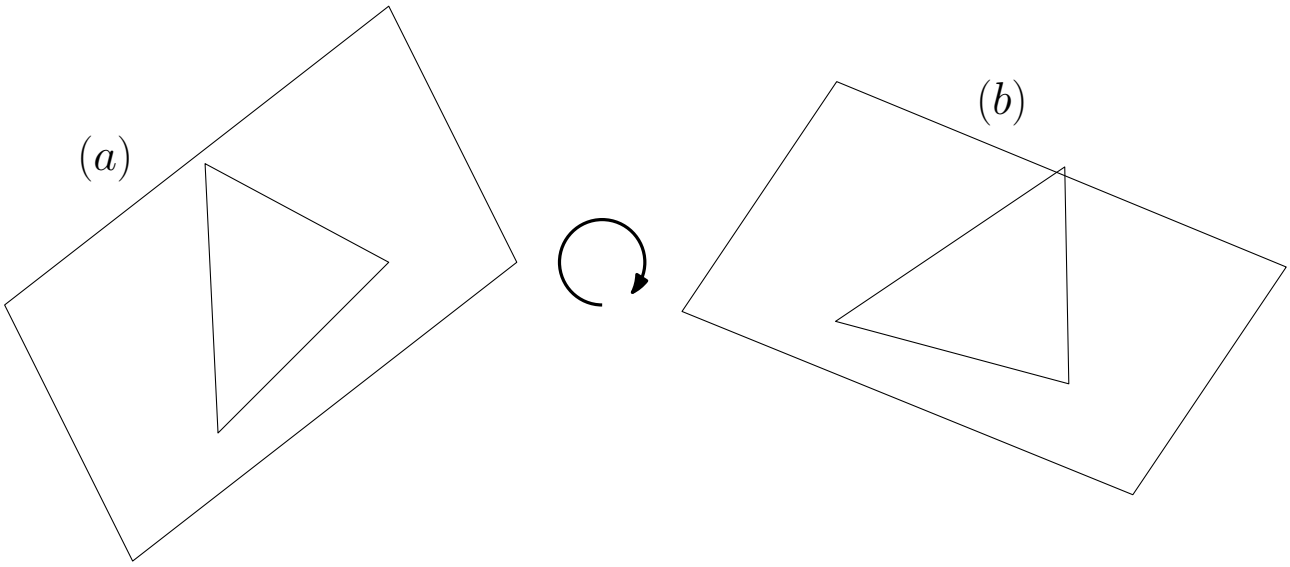


Figure 1.1: (a) A quadrilateral containing a triangle. (b) Input rotated by $-\frac{\pi}{3}$. Due to rounding, the triangle now intersects the quadrilateral.

Chapter 2

2D snap rounding

2.1 Introduction to 2D Snap Rounding

Before delving into the subject of 3D Snap Rounding, we first introduce its counterpart in 2D. Hershberger [17], in the introduction of his paper on Stable Snap Rounding, provides an excellent summary of the subject, which we follow in this section.

Recall that geometric algorithms are often described using as real numbers coordinates and other values. However, implementations almost always represent these geometric objects using finite precision, characterized by the number of bits used. Geometric coordinates may be represented as rational numbers, fixed-precision integers, or, more commonly, as floating-point numbers.

When a geometric algorithm constructs new geometric objects based on its input, the resulting objects typically require more bits to be represented exactly than the input objects. For instance, as already mentioned in Chapter 1, consider the intersection of two line segments where the endpoints have b-bit integer coordinates. This intersection point coordinates are rationals where the numerator and denominator respectively require in general $3b + O(1)$ bits and $2b + O(1)$ bits to be represented (see next section).

For this reason, implementations of geometric algorithms often round their output to the same precision as their input (usually single or double precision floating-point numbers). To be usefull, a rounded geometric result must be close to the true result, for some definition of "close" (geometrically, topologically, ...).

In the next section, we establish the proof for the size of the representation of the intersection

of two segments. The third section provides a review of the state of the art on the subject. In the fourth section, we outline the theoretical constraints for constructing a solution. The fifth section introduces the Greene and Hobby algorithm for 2D Snap Rounding. The sixth section presents a variant of this algorithm that outputs an unrounded triangulation of input with certified rounding motion. The final two sections discuss the difficulties of rounding to a float grid and present an algorithm for rounding coordinates to the nearest floats.

2.2 Preliminaries: growth in the size of representation

As said before, when a geometric algorithm constructs new geometric objects based on its input, the resulting objects typically require more bits to be represented exactly than the input objects. For instance, consider the intersection of two line segments where the endpoints have b-bit integer coordinates. This intersection point are a rational where the numerator and denominator respectively require in general $3b + O(1)$ bits and $2b + O(1)$ bits to be represented.

We prove here this statement. First, the result of a product two integer of k and k' bits requires $k + k' + O(1)$ bits and the result of an addition or a soustraction of two integers of k and k' bits requires in general $\max(k, k') + O(1)$ bits.[2] Define the vector cross product as $V \times U = V_x U_y - V_y U_x$. Consider two line segments AB and CD, with direction vectors $V = B - A$ and $U = D - C$. The intersection point, if it exists, is determined by the intersection of the parameterized lines $A + tV$ and $C + sU$, where t and s are the parameters. The intersection occurs when $A + tV = C + sU$. Multiplying both sides by the vector U (noting that $U \times U = 0$), we obtain $t(V \times U) = (C - A) \times U$. Solving for t, we get:

$$t = \frac{(C - A) \times U}{V \times U}.$$

The cross product $V \times U$ of two vectors with b-bit integer coordinates requires $2b + O(1)$ bits, as it involves the subtraction of products of b-bit integers. Therefore, t is a rational number with a numerator and denominator each requiring $2b + O(1)$ bits. The intersection point's coordinates are given by $A + tV$, where the multiplication of t (with a numerator and denominator of $2b + O(1)$ bits each) by an integer vector of b bits results in coordinates with a numerator requiring $3b + O(1)$ bits and a denominator of $2b + O(1)$ bits. Thus, the intersection point have rational coordinates

[2] In some case, such as the soutraction of a number by itself, the output of operation may require less bits.

requiring in general $5b + O(1)$ bits for an exact representation.

2.3 State of the Art

In 1986, Greene and Yao [14] were the first to propose an algorithm for rounding a line segment arrangement (the planar subdivision induced by a collection of line segments) to the integer grid. Their approach deforms each segment to a polygonal path, moving each of its intersections to the nearest grid point while ensuring that the resulting path does not pass over any other grid point. Simultaneously, each moving intersection acts as a "hook" to pull other segments to the same grid point, thereby preserving the topological relationships between segments and vertices. This algorithm guarantees an accurate representation of the arrangement, but it may increase the total number of vertices in the arrangement by a logarithmic factor.

In 1989, Milenkovic [22] presented an algorithm for rounding an arrangement of lines as part of his "double precision geometry" framework. His approach replaces each line with a nearby line, then applies an algorithm for rounding the line arrangement. The algorithm maintains topological correctness for the line arrangement by replacing each line with a shortest path among the rounded positions of nearby arrangement vertices. This algorithm does not suffer from the logarithmic blowup of Greene and Yao's approach, but it may not preserve the topology of the original input.

In both of these algorithms, the interaction between each arrangement vertex and each nearby segment is computed explicitly. Snap rounding, independently invented by Greene [13] and Hobby [18] in 1996, simplifies rounding by separating the computation of arrangement vertices from segment rounding. Their approach involves tagging "hot" pixels that contain a vertex of the arrangement and subdividing the segments according to these "hot pixels". Snap rounding guarantees two very important qualities [15]. First, the rounded version of each input segment lies within half a pixel distance of its unrounded version. Second, the topology of the input segments is preserved in the rounded arrangement, up to collapsing of features (see Section 2.5). This algorithm is currently considered a standard for rounding segments in 2D and is detailed in Section 2.5.

Canonicity is another important snap rounding feature. The result of snap rounding depends only on the input segments, and not on any algorithmic choices or order of operations. Canonicity

is important because it simplifies the proof of metric and topological accuracy. Implementability is a final important feature in snap rounding and many efficient algorithms have been developed to compute a snap rounded arrangement of line segments [7, 12, 15, 18].

Although Greene and Hobby's algorithm has many strength, it suffers from the weakness that it is not stable (i.e., it is not idempotent). In other words, applying snap rounding to its own output may change the result. Although a single application of snap rounding guarantees that segments do not move by more than half a pixel, segments may drift arbitrarily far when snap rounding is applied repeatedly. This issue is not critical to the purposes of this manuscript and is thus not discussed further but many papers propose variants of Greene and Yao's algorithm to address this issue [16, 17, 26].

2D Snap Rounding is defined on a pixel grid, rounding coordinates to the nearest integer values. However, most implementations of geometric algorithms use floating-point number representations rather than integers. In some cases, it may be desirable to round coordinates to the nearest floating-point values to maintain as much precision as possible. Rounding on floating-point values means that vertices may shift by varying distances depending on their position within the model, making it difficult to adapt traditional 2D Snap Rounding to such a grid. For further details, refer to Section 2.8 in which, we present the first algorithm that rounds vertices to the nearest points with floating-point coordinates, while preserving the same guarantees of topology and canonicity as the Green and Yao algorithm.

In conclusion, 2D Snap Rounding is a well-understood and solved problem, both in theory and in practice.

2.4 Necessity of subdivision and impossibility to preserve the "entire" topology

All rounding algorithms perform subdivisions, and none preserve the topology while maintaining a bounded distance between input and output. Indeed, Milenkovic and Nackman [24] demonstrate that avoiding subdivisions is difficult. They investigated a specific case of rounding a simple polygon, aiming to round it without subdivisions and without causing self-intersections. To achieve this, they allowed adjusting the k smallest digits of each coordinate after rounding. However, they proved that it is not always possible to avoid self-intersection this way, and the

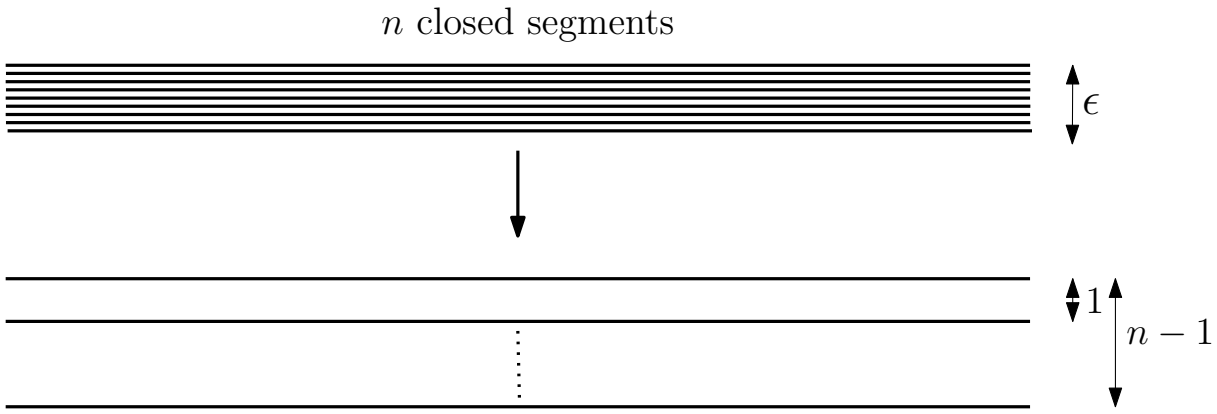


Figure 2.1: Topology versus distance between the input and the output. n long horizontal segments in a strip of height ϵ. To round and preserve the topology, the distance between the first and last segments must grow to $n-1$ pixels.

associated decision problem is NP-complete. Therefore, rounding problems without subdivisions are difficult and sometimes impossible.

Some rounding algorithms, such as the one by Greene and Hobby (see Section 2.5), provide certain guarantees regarding topology but do not preserve it entirely; features may collapse. If we aim to avoid collapsing, the Hausdorff distance between a segment and its rounded version may become unbounded. For example, consider n very long horizontal segments contained in a strip of height ϵ. To avoid collapsing, each consecutive segment must be separated by at least one pixel, causing the distance between the first and last segments to grow to $n-1$ pixels, losing the constraint on the distance (see Fig. 2.1). Thus, preserving both topology and geometry simultaneously is often not possible, and thus the notion of topology preserved "up to collapse" (see Def. 1) is usually considered.

Definition 1. *A set of segments (or triangles in 3D) S_1 preserves the topology up to collapse of a set of segments (or triangles in 3D) S_0 if it exists a continous motion from S_0 to S_1 such that if two points become equal during the motion then they remain equal during the rest of the motion. More formally, in dimension d, a set of points S_1 preserves the topology up to collapse of S_0 if and only if it exists a continous function $\sigma : \mathbb{R}^d * [0,1] \to \mathbb{R}^d$ such that $\sigma(S_0, 0) = S_0$, $\sigma(S_0, 1) = S_1$ and $\sigma(p,t) = \sigma(p',t) \implies \sigma(p,t') = \sigma(p',t'), \forall t' \geqslant t$.*

2.5 Greene and Hobby's Algorithm

Since some of our work is inspired by Greene and Hobby's algorithm, and we use some variant of their algorithm, we detail their approach to solving 2D Snap Rounding [13, 18]. Roughly speaking, the algorithm tag "hot" all pixels that contains a vertex of the arrangement, subdivide all segments that cross hot pixels and then round all the vertices to the center of their pixel.

More formally, snap rounding is defined using pixels, which are unit squares centered at integer grid points. To ensure that each point in the plane belongs to exactly one pixel, pixels are defined as closed at the left and bottom edges and open at the right and top edges. Thus, the bottom left corner is included in the pixel, while the other three corners are not. An input segment intersects a pixel if and only if it intersects the pixel's interior or a closed boundary point. The algorithm proceeds as follows: Mark as "hot" each pixel that contains a vertex of the input or an intersection point of two non-collinear segments (Fig. 2.2(b)). Subdivide the input segments by their intersections with other segments and by the boundaries of the hot pixels (Fig. 2.2(c)). Subdivision here means creating a new vertex at the intersection point and replacing the segment with two new segments: one from the original start to the new vertex, and another from the new vertex to the original end. [3] Round the x-coordinates of vertices at a constant speed from $t = 0$ to $t = 1$ (Fig. 2.2(d-e)). Round the y-coordinates of vertices at a constant speed from $t = 1$ to $t = 2$ (Fig. 2.2(f)).

Each input segment becomes a polyline in the output. Since all vertices move by at most half a pixel, the segments also move by at most this distance, ensuring that the Hausdorff distance between any input segment and its corresponding output polyline is at most one half. Guibas and Marimont [15] proved that the topology is preserved up to collapse (Theorem 2) according to Definition 1. This theorem guarantees that a vertex does not cross a segment during the coordinate-by-coordinate rounding motion. As a result, a polygon may collapse to a line segment or a point, and a line segment may collapse to a point, but no polygon will reverse its orientation. Additionally, if an input segment s intersects two segment s_1 and s_2 in that order, the rounded polyline of s intersects the rounded version of s_1 and s_2 in that order too.[4] The complexity is

[3] The vertices that are created on the boundaries of a hot pixel are associated with it so that they are rounded to the center of this pixel, ensuring that no intersections occur during the rounding motion; see Chapter 9 for details. For the proof of Theorem 2, it is necessary to subdivide by the boundary of the hot pixel. However, the output remains the same if we subdivide a segment by any point inside the pixel instead.

[4] They can be crossed at the same point.

optimal both in space and time with a $O(n^2)$ worst-case complexity (Theorem 3).

Theorem 2 (Guibas and Marimont [15, Theorem 1]). *Considering the subsegments resulting from the subdivision of input segments by Greene and Hobby's algorithm, vertices of those subsegments do not cross another subsegment during the rouding motion of Greene and Hobby's algorithm.*[5]

Theorem 3 (Hobby [18, Theorem 2.1], Guibas and Marimont [15, Theorem 2]). *The number of vertices of the subsegments before the rounding is $O(n(n+k))$ where n is the number of input segments and k the number of intersections of those segments and the number of vertices in the rounding output is at most $O(n+k)$. The output can be computed in time $O(n \log n + k)$.*

2.6 Triangulated variant

As discussed in the previous section, Greene and Hobby's 2D snap rounding algorithm takes segments (in 2D) as input and subdivides them by the boundaries of hot pixels before rounding the vertices to the centers of their pixels. However, when the input is a set of (possibly intersecting) polygons, their algorithm does not triangulate the subdivided faces. For most applications, this is not an issue, as we can triangulate the faces after the rounding, if needed. However, when snap rounding faces in 3D, it is sometimes useful to consider the projections of the faces in 2D, subdivide them as done in the 2D snap rounding algorithm (without actually rounding the vertices), and lift back these subdivisions onto the 3D faces. These resulting 3D faces need to be triangulated before rounding their vertices (to the centers of their voxels) because, in general, the non-triangulated faces do not remain planar during the rounding motion. To avoid 3D inconsistencies, the 2D faces should ideally be triangulated before lifting them on the 3D faces.

Devillers et al. [8] presented a 2D snap-rounding variant in which the boundaries of the hot pixels are added to the set of segments before triangulating the resulting arrangement.[6] See Figure 2.3 for visualization. Since no segment of the resulting triangulation crosses the boundaries of the hot pixels, no intersection occur during the rounding of the vertices to the center of their pixels.

We will use this 2D snap-rounding variant algorithm in several place in this manuscript (Sections 3.4, 6.5, 8.2) and we present an optimized lazy version in Section 8.4.

[5] A vertex can lie on a subsegment without crossing it during the rounding motion.

[6] The vertices that form the corners of a hot pixel are associated with it so that they are rounded to the center of this pixel, ensuring that no intersections occur during the rounding motion; see Chapter 9 for details.

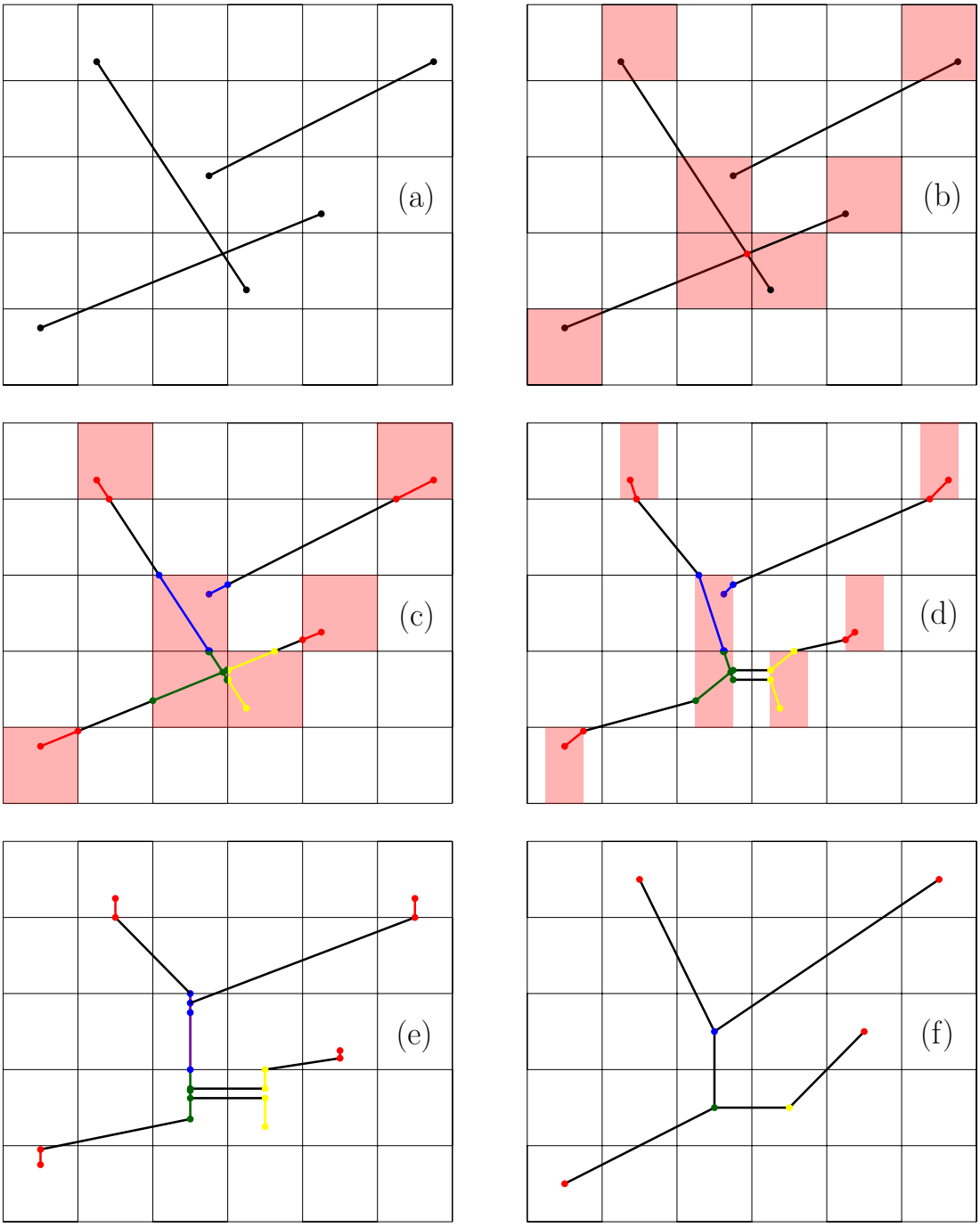


Figure 2.2: Greene and Hobby's algorithm [14]: (a) Input segments. (b) "Hot" pixels contain a vertex of the arrangement. (c) Segments subdivided by the boundaries of the hot pixels. (d-e) x-coordinates are rounded to the closest integer. (f) y-coordinates are rounded to the closest integer.

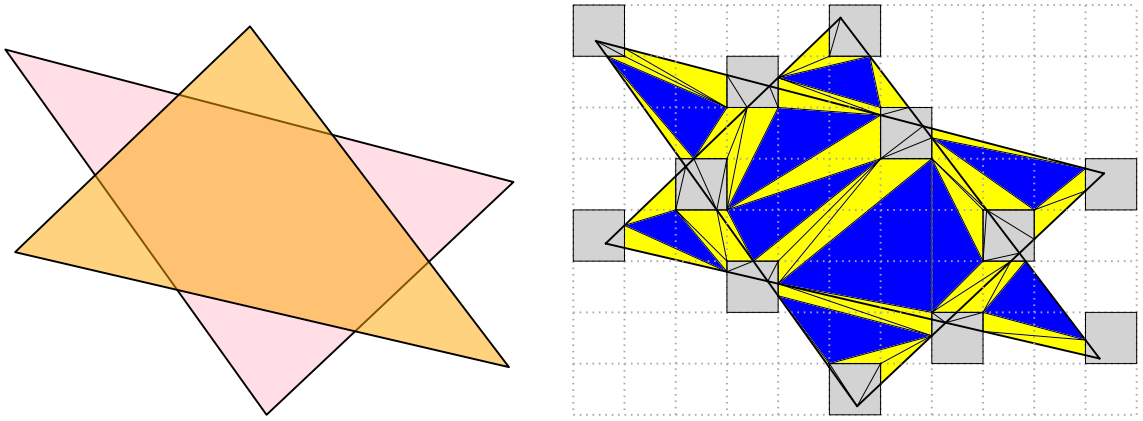


Figure 2.3: Triangulated snap rounding on two triangles (Devillers et al. [8]). Hot pixels are in grey. Yellow triangles are degenerated at the end of the rounding. Blue triangles are not. Triangulation edges outside the two input triangles are not drawn for readability.

2.7 Rounding on integer versus rounding on floats

As a reminder, classical 2D snap rounding is defined on a pixel grid, rounding coordinates to the nearest integer values. However, most implementations of geometric algorithms use floating-point representations rather than integers. In most cases, this is not an issue; one can take the finest possible regular grid that can be represented by floating point numbers. Rounding a positive floating point number $f = m2^k$ consist to set all the digits of the mantissa after the k^{th} to zero.[7] To get the finest possible regular grid, scale all the coordinates in the model such that the highest exponent of coordinates are exactly the size of the mantissa, round each value to the nearest integer, and then reverse the scaling. This method round to values representable with floating point numbers and values are drifted by the worst precision of output values.[8]If values are enclosed by 0 and 1, this precision is $2^{-|m|}$ where $|m|$ is the size of the mantissa of the floating representation. It is roughly 10^{-7} on a grid of single-precision floats (23 bits for the mantissa and 8 for the exponent) and roughly 10^{-15} on a grid of double-precision floats (52 bits for the mantissa and 11 for the exponent). To vizualise this relative precision, this approach provides a precision of about 100 nanometers for an object with a width of one meter on a grid of single-precision floats and for double-precision floats, the precision is about one nanometer for an object the width of the Earth. Therefore the precision given by rounding on integer up to a scaling is enough for a majority of applications. Despite this, it might be interesting to round coordinates to the

[7]Here, we considered the "trunc" mode of norm IEEE, other modes add ± 1 to this value depending of the case.
[8]We store the values with floating numbers while they are integers.

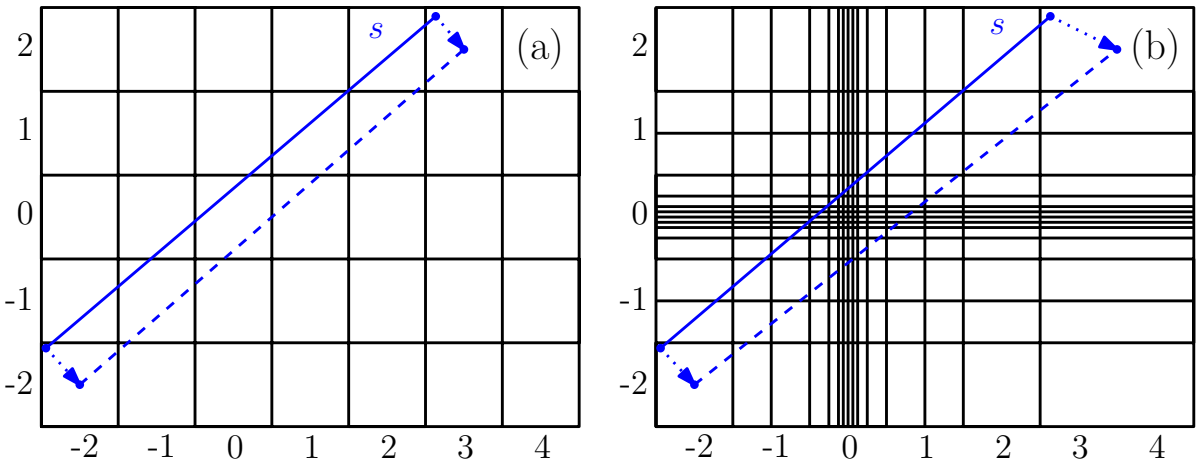


Figure 2.4: Issues with rounding on irregular grids. (a) Regular rounding grid for snapping to the nearest integer coordinates; segment s does not traverse an entire cell during the motion. (b) Irregular rounding grid for snapping to the nearest floating-point coordinates.[11]Segment s traverses multiple cells during the motion.

nearest floating-point values to maintain maximum precision, especially around zero coordinates. Rounding on floating-point values results in vertices shifting by varying distances, depending on their position within the model, complicating the adaptation of traditional 2D snap rounding to such a grid.

More formally, given a number type $\mathbb{K}$ and $\mathbb{K}^2$ be the set of points with coordinates in $\mathbb{K}$, we define the rounding grid as the decomposition of $\mathbb{R}^2$ into cells such that each cell contains only one point from $\mathbb{K}^2$, referred to as the "center" of the cell.[9] For all other points in this cell, the center is their nearest point in $\mathbb{K}^2$ (in other words, the rounding grid is the Voronoi diagram of the points in $\mathbb{K}^2$). For $\mathbb{K}$ as integers, the rounding grid cells are pixels, and the grid is regular, periodic, and identical across all cells (see Fig. 2.4(a)). For $\mathbb{K}$ as floating numbers, the grid cells are still axis-aligned rectangles. Given a cell and p its center, the width (respectively height) of this cell is $2^{k-|m|}$, where k is the exponent and $|m|$ the size of the mantissa of the x-coordinate (respectively y-coordinate) of p (see Fig. 2.4(b)). This grid is irregular, with significantly higher precision near zero. One key property of integer rounding is that a segment does not entirely traverse a pixel during the motion, a property that no longer holds on a floating-point rounding grid, making it difficult to adapt the 2D snap rounding algorithm.

[9]The "center" of the cell is not necessarily the geometric center.

[11]The number of cells between two scaling changement is 2^m where m is the size of the mantissa, for readability this figure is drawing supposing a very short mantissa.

2.8 Floating-point 2D snap rounding

We present the first floating-point 2D snap rounding algorithm. Given a set of input segments, the algorithm outputs a simplicial complex with coordinates using floating-point numbers (of any precision) such that the Hausdorff distance between an output vertex and its corresponding point on the input is at most one fourth of the perimeter of the cell containing this vertex, ensuring that the topology is preserved "up to collapse". The algorithm follows the approach of Devillers et al. [8] by slicing the scene into parallel and vertical slabs. Refer to Figure 2.5 for vizualisation. We tag "hot" each cell of the rounding grid containing a vertex of the segment arrangement (an endpoint of a segment or an intersection of two segments) (Fig. 2.5(b)). For each hot cell, we consider the narrow slab defined by the two vertical lines supporting the vertical boundaries of the cell, referred to as a "hot slab". Subdivide all segments by the boundaries of all hot cells and hot slabs (Fig. 2.5(c)). Subdivision here involves creating a new vertex at the intersection point and replacing the segment with two new segments: one from the original start to the new vertex and another from the new vertex to the original end.[12] Round the x-coordinates of vertices at constant speed from $t = 0$ to $t = 1$, followed by rounding the y-coordinates at constant speed from $t = 1$ to $t = 2$ (Fig. 2.5(d)).

The output of the algorithm maintains the topological guarantees as 2D snap rounding (Theorem 4) and the worst-case complexity of the output is $O(n^3)$ (Theorem 5). Although this complexity is higher than that of traditional 2D snap rounding, which outputs at most $O(n^2)$ rounded vertices, we can design a lazy version of this algorithm similar to lazy triangulated 2D snap rounding (see Section 8.4). By simulating the rounding motion and subdividing an edge by the hot slab and the hot cell associated with a vertex only whenever this vertex crosses the edge, the worst-case complexity remains the same, but we can expect a nearly-linear algorithm in practice.

Theorem 4. *Considering the subsegments resulting from the subdivision of input segments by the floating-point 2D snap rounding algorithm, the vertices of those subsegments do not cross another subsegment during the rounding motion.*[13]

[12] The vertices created on the boundaries of a hot cell/slab are associated with it so that they are snapped to the center of this cell/slab, ensuring no intersections occur during the snapping process. See Chapter 9 for details. For the proof of Theorem 4, it is necessary to subdivide by the boundaries of the hot cells/slabs. However, the output remains unchanged if we subdivide a segment by any point within the same cells.

[13] A vertex can lie on a subsegment without crossing it during the rounding motion.

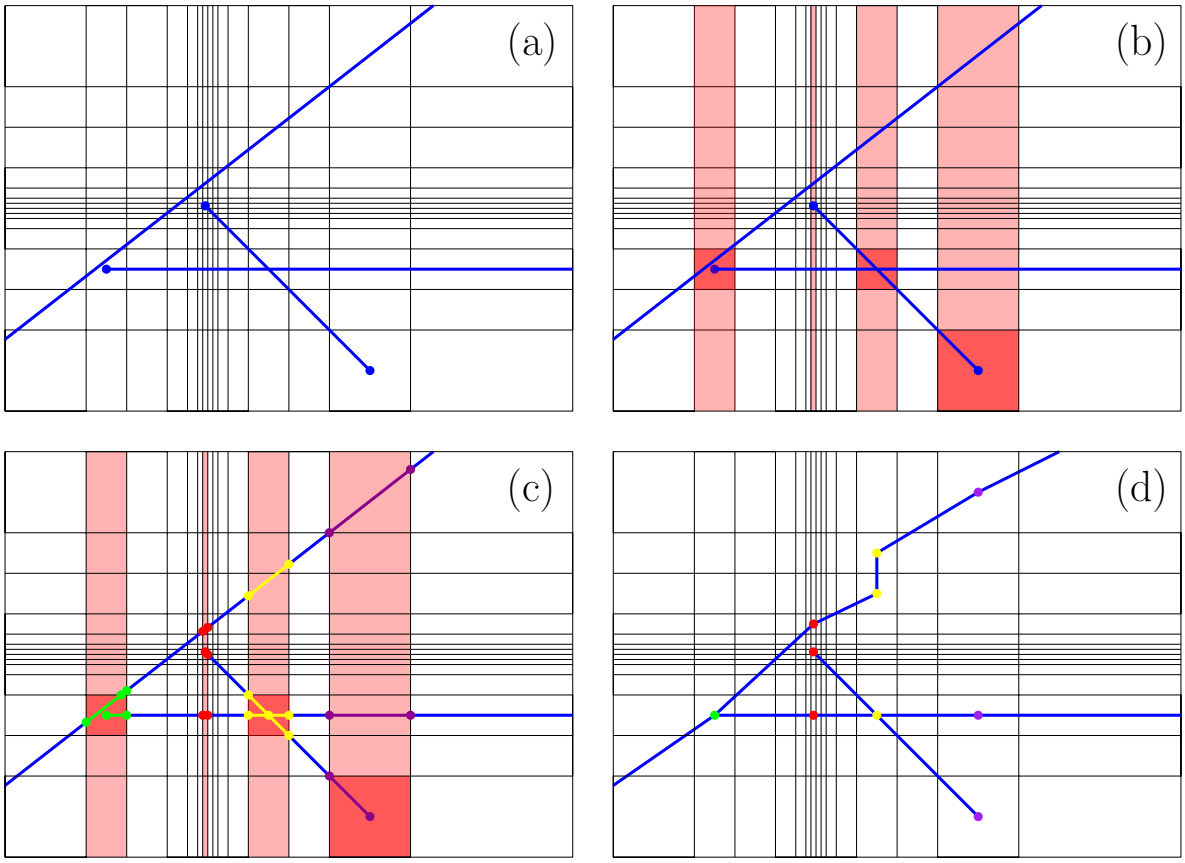


Figure 2.5: Float 2D snap rounding algorithm: (a) Input segments. (b) "Hot" cells and slabs contain a vertex of the arrangement. (c) Segments subdivided by the boundaries of the hot cells and the hot slabs. (d) x then y-coordinates rounded at a constant speed.

Proof. We separate the proof between the hot slabs and the "wide slabs" defined as the slabs between two consecutive hot slabs.[14] Roughly speaking, in a hot slab, all vertices snap onto the line at the "center" of the slab,[15] preventing crossings during the motion, and in a wide slab, all segments span from one boundary to the other, maintaining the vertex orderings on the boundaries and preventing crossings during the motion.

More formally, elements resulting from the subdivision of input segments by the floating-point 2D snap rounding algorithm are called subsegments. To prove that a vertex does not cross a subsegment during the rounding motion, we consider cases where two features are in the same hot slab, the same wide slab, or not in a common slab. In the last case, a subsegment remains within the same slab, hot or wide, during the motion, and hence a vertex does not cross a subsegment if

[14] A wide slab can have zero width before the rounding motion.

[15] This "center" line is the line $x = cst$ where the constant is a floating number, note that this line is not necessarily the geometric center of the slab.

they are not both in the same slab.[16]

For subsegments within a hot slab, during the x-motion, all vertices round to the "center" of the slab, this motion is equivalent to scaling the x-coordinates except at the end of the motion and prevents crossings. During the y-motion, all segments lie on the vertical line at the center of the slab, so no vertex crosses a subsegment. Since segments are subdivided by the boundaries of the hot cells, a vertex does not lie on the interior of a subsegment at the end of the motion as the segment would have been subdivided within the cell of this vertex.

For subsegments within a wide slab, there are no vertices inside the wide slab; all vertices are on its vertical boundaries. Hence, vertices on the left (or right) boundary round to the same x-value, therefore the x-motion is also equivalent to a scaling preventing crossings during the x-motion. During the y-motion, if a vertex were to cross a subsegment, it would do so at a vertex since all vertices are on the boundaries of the slab. The order of vertices along the y-direction on the left (or right) boundary does not change during the y-motion, preventing crossings and thus concluding the proof. □

Theorem 5. *Given n input segments, the floating-point 2D snap rounding algorithm outputs a set of $O(n(n+k))$ vertices, where k is the number of intersections of the input segments.*[17]

Proof. Hot slabs are defined by the presence of a vertex of the arrangement and are therefore bounded by $n+k$. Each hot slab or hot cell subdivides each segment at most twice, creating at most $O(n)$ vertices per hot slab or hot cell, thus concluding the proof. □

This algorithm for 2D snap rounding is an independant contribution of this thesis, not related to 3D snap rounding.

[16] A vertex on the boundary of a hot slab is considered part of both the hot slab and the adjacent wide slab.
[17] The number of intersections k is at most $O(n^2)$.

Chapter 3

3D snap rounding

3.1 Introduction to 3D snap rounding

After discussing the 2D Snap Rounding problem, we now return to the main focus of this thesis: 3D Snap Rounding. In 2D, the number of bits required to describe a constructed object increases rapidly. We demonstrated that the intersection of two line segments, where the endpoints have b-bit integer coordinates, requires in general $5b + O(1)$ bits for represent the result as a rational. This issue of increasing representation size is even worse in 3D. For instance, consider the intersection point of three triangles whose vertices have b-bit integer coordinates. This intersection point is a rational where the numerator and the denominator require in general respectively $7b+O(1)$ and $6b+O(1)$ bits to be represented (see next section). As mentioned before, if these intersection points are to be used as input to some algorithms, most implementations requires floating-point numbers for the coordinates.

To avoid these issues, in practice, coordinates are often rounded but without guarding against changes in topology. There is no guarantee that the rounded objects do not properly intersect one another. For instance, two triangles that do not intersect initially may intersect after rounding (see Fig. 3.1). In Thingi10K [32], a dataset of 10,000 meshes from Thingiverse, about half of the models exhibit self-intersections. Models can have self-intersections for various reasons, but rounding issues are a very common source.

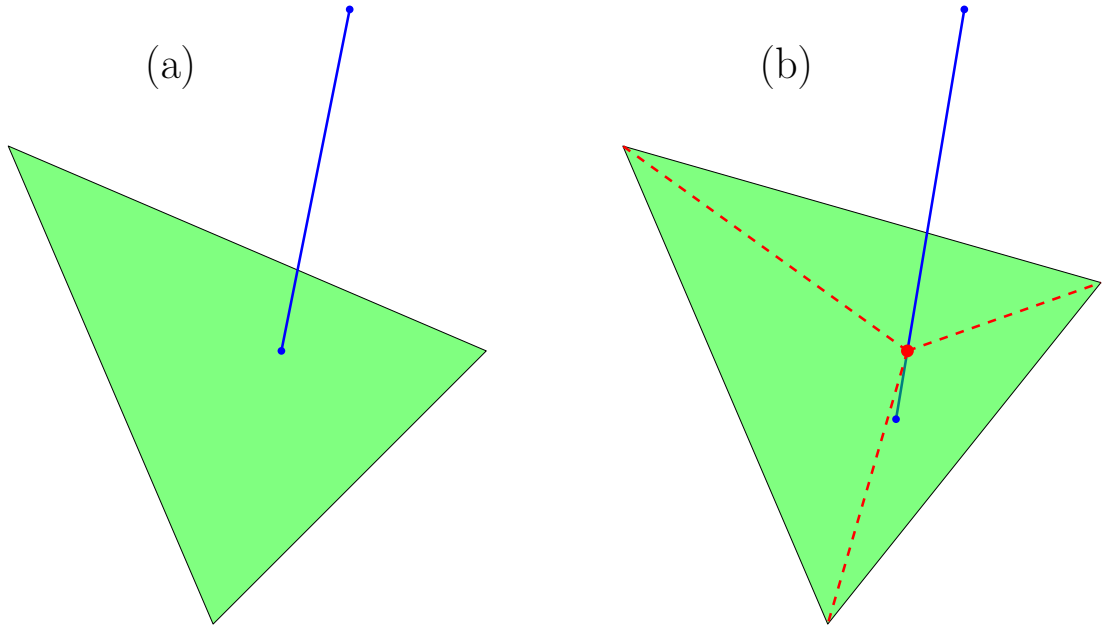


Figure 3.1: Two triangles that do not intersect initially may intersect after rounding. (a) A segment close to a triangle. (b) The segment intersect the triangle after being rounded.

Some exact geometric arithmetics have been developed [30] to avoid roundings and their topology errors but as shown with the intersection of three triangles, this consumes a lot of space and computation time. In 2006, to avoid the cost of exact computations, Fabri and Pion [27] introduced a lazy exact geometric arithmetic implemented in the Computational Geometry Algorithms Library (CGAL) [28]. Their approach represents each value by an interval of fixed-size floating-point numbers enclosing the exact value and a Directed Acyclic Graph (DAG) of the operations performed to obtain this value. Each operation is initially performed using only the intervals and by adding a node to the DAG. If a predicate is called (e.g., testing if a number is greater than zero), their construction first tries to answer using the intervals. If this is not possible (e.g., if zero is within the interval of the representation), they repeat the operations in the DAG using an exact arithmetic. The hope is that most computations are successfully done by the fast interval arithmetic and only a few requiring to use costly exact arithmetic. This approach guarantees the results of algorithms with expected low computation time costs. However, the precision of the intervals decreases as operations are cascaded, which limits the effectiveness of these approaches, and the results are hard to diffuse.

Rounding polygonal meshes accurately is a significant problem in computational geometry,

computer graphics, and industry. Numerous studies [2, 5, 6, 9, 19, 20, 31] in computer graphics have highlighted the significant issues posed by rounding. Unlike the 2D problem, results to solve these issues in 3D are extremely scarce (see section 3.3).

Like 2D Snap Rounding, it is more convenient to round on integer grids to provide a certified result. Thus, most of the following algorithms assume rounding each vertex to the closest point with integer coordinates, often described using voxels, where a voxel is a unit cube centered on vertices of the integer grid.

In the next section, we provide the definition of proper intersection and establish the proof for the size of the representation of the intersection of three triangles. The third section reviews the state of the art on the subject, and the fourth section presents a flawed algorithm to illustrate the difficulties associated with 3D snap rounding.

3.2 Preliminaries

3.2.1 Proper intersections

The objective of any rounding algorithm is to produce a simplicial complex. Therefore, we focus on intersections that prevent this structure. We refer to such intersections as *proper intersections* as defined below.

Definition 6. *Two polygons, edges, or vertices are said to properly intersect if their intersection is non-empty and not a common face of both.*

3.2.2 Growth in the size of representation

As mentioned before, the size of the representation of constructed object increase quickly. For instance, consider the intersection point of three triangles whose vertices have b-bit integer coordinates. This intersection point is a rational where the numerator and the denominator require in general respectively $7b + O(1)$ and $6b + O(1)$ bits to be represented. We prove this statement. Given a triangle $T = ABC$, the equation of the plane supporting T is:

$$\begin{vmatrix} 1 & x & y & z \\ 1 & x_A & y_A & z_A \\ 1 & x_B & y_B & z_B \\ 1 & x_C & y_C & z_C \end{vmatrix} = \begin{vmatrix} x_A & y_A & z_A \\ x_B & y_B & z_B \\ x_C & y_C & z_C \end{vmatrix} - x \begin{vmatrix} 1 & y_A & z_A \\ 1 & y_B & z_B \\ 1 & y_C & z_C \end{vmatrix} + y \begin{vmatrix} 1 & x_A & z_A \\ 1 & x_B & z_B \\ 1 & x_C & z_C \end{vmatrix} - z \begin{vmatrix} 1 & x_A & y_A \\ 1 & x_B & y_B \\ 1 & x_C & y_C \end{vmatrix} = 0. \quad (3.1)$$

As a result, the supporting plane of T is represented by an equation of the form $ax+by+cz+d = 0$ where a, b, c are sums of products of two input coordinates, generally requiring $2b + O(1)$ bits, and d is the sum of products of three input coordinates, generally requiring $3b + O(1)$ bits. Given three triangles T_1, T_2, T_3, the intersection point of their supporting planes solves the following linear system:

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = 0. \quad (3.2)$$

By Cramer's rule, x satisfies the next equation:

$$x = \frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}. \quad (3.3)$$

Thus, the numerator of x is the sum of products of two numbers of $2b + O(1)$ bits with a number of $3b + O(1)$ bits, which generally requires $7b + O(1)$ bits. The denominator is the sum of products of three numbers of $2b + O(1)$ bits, generally requiring $6b + O(1)$ bits. Therefore, the intersection point of three triangles with b-bit integer coordinates generally requires $13b + O(1)$ bits for exact representation. If operations are cascaded, the number of bits required to describe the new objects increases exponentially, leading to space and computation time issues.

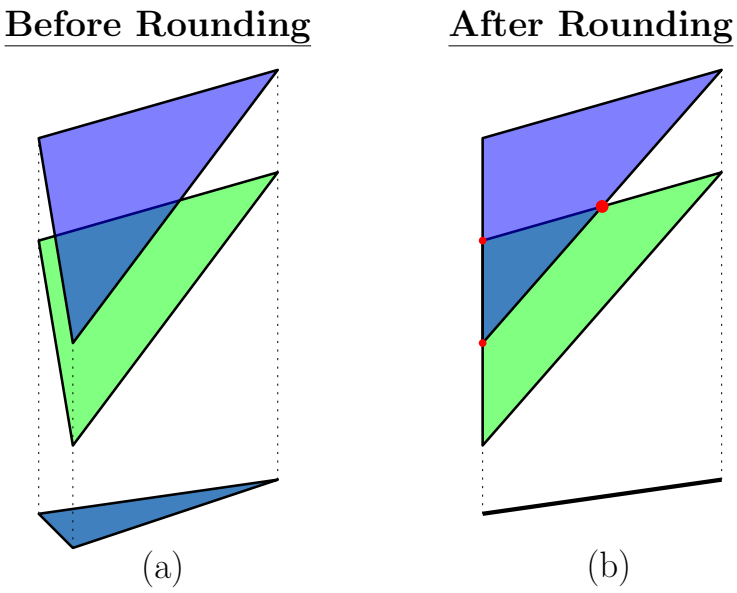


Figure 3.2: Counter-example to PSL algorithm [23]: (a) two triangles, one above the other with equal projection on the floor, (b) After rounding, they round into two properly intersecting vertical triangles degenerate on the floor. The Milenkovic's algorithm [23] and the Project, Subdivide and Lift (PSL) algorithm (Section 3.4) both fail on such input.

3.3 State of the art

In 1990, Milenkovic [23] was the first to provide an algorithm for 3D Snap Rounding (actually for any dimension). His approach consists in projecting the faces onto the floor (xy-plane), computing the arrangement on the floor, adding to the arrangement a line parallel to the y-axis through each vertex on the floor, and then lifting this arrangement back onto the faces in 3D. This algorithm is simple, but Fortune [11] pointed out that it is flawed. Indeed, consider two faces with identical projections on the floor that become vertical in the same plane after rounding; these two faces may intersect after rounding and are not subdivided by Milenkovic's algorithm (see Fig. 3.2).

In 1997, Goodrich et al. [12] proposed a scheme for rounding segments in 3D, using ideas from 2D Snap Rounding. It tags "hot" the voxels containing a vertex or two "close" segments and subdivides all segments by the hot voxels. More formally, for each pair of non-parallel segments (Goodrich et al. assume general position), if their distance is smaller than one, the segment are subdivided by the two points that realize the distance. In spirit, this operation is equivalent to subdividing in 2D the segments by their intersection if it exists. Then, all the voxels containing a vertex in 3D are tagged "hot". Finally, the segments are subdivided by their intersections with

the boundaries of the hot voxels. This algorithm is simple and has a good complexity $O(n^2)$, but again, Fortune [11] pointed out that it is flawed, as described below.

We present a counter-example of the algorithm of Goodrich et al. [12] inspired by that of Fortune [11] but without intersection of the segments on the projection on the three axis-align planes. Refer to Figure 3.3. Consider two nearly parallel segments that round into the same vertical plane and properly intersect at the end of the rounding. Their endpoints are pairwise "close" but located in different voxels. We assume that the segments are resulting from subdivision by boundaries of hot voxels and thus their endpoints are on the boundary of their voxels and are not subdivided by them. The coordinates of the segments AB and CD are $A = (1, \epsilon, 1)$, $B = (6 + \epsilon, 6 + \epsilon, 4)$, $C = (\epsilon, 1, 0.8)$, $D = (6 + \epsilon, 6 + 3\epsilon, 4 + \epsilon)$ where $0 < \epsilon \ll 1$. These coordinates are chosen such that the segments do not cross into the voxels containing the endpoints of the segments, and the minimum distance between the two segments are realized by endpoints (Fig. 3.3). Consequently, these segments do not intersect any hot voxels and are not subdivided by Goodrich's algorithm. The coordinates of AB and CD are also chosen so that they do not intersect in the projection onto any of the three axis-aligned planes (Fig. 3.3(c-e-g)), so including the operation of subdividing edges according to their projections on the three axis-aligned planes, as done in many other rounding algorithms, will not suffice to create a correct algorithm. This example shows that intersections during rounding can be very intricate, making the design of algorithms to prevent them, even for just rounding segments, a significant challenge.

The first correct results were provided by Fortune [10,11] in 1997 and 1999, but he did not solve the target problem. Fortune suggested a high-level rounding scheme for plane-based polyhedra, where faces are defined by their supporting planes, and edges and vertices are intersections of these planes. Fortune's algorithm rounds the coordinates of the planes, which does not generalize to the more common vertex-based polyhedra [10]. Later, Fortune [11] proposed a rounding algorithm that takes a set P of n disjoint triangles in $\mathbb{R}^3$ and maps it to a set Q of triangles with $O(n^4)$ vertices on a discrete grid such that (i) every triangle in P is mapped to a set of triangles in Q at L_{inf}-Hausdorff distance at most $\frac{3}{2}$ from the original face, and (ii) the mapping either preserves or collapses the vertical ordering of the faces. Unfortunately, this rounding scheme is very intricate and uses grid precision that depends on the number n of triangles, with vertex coordinates rounded to integer multiples of approximately $\frac{1}{n}$, making it impractical for most applications.

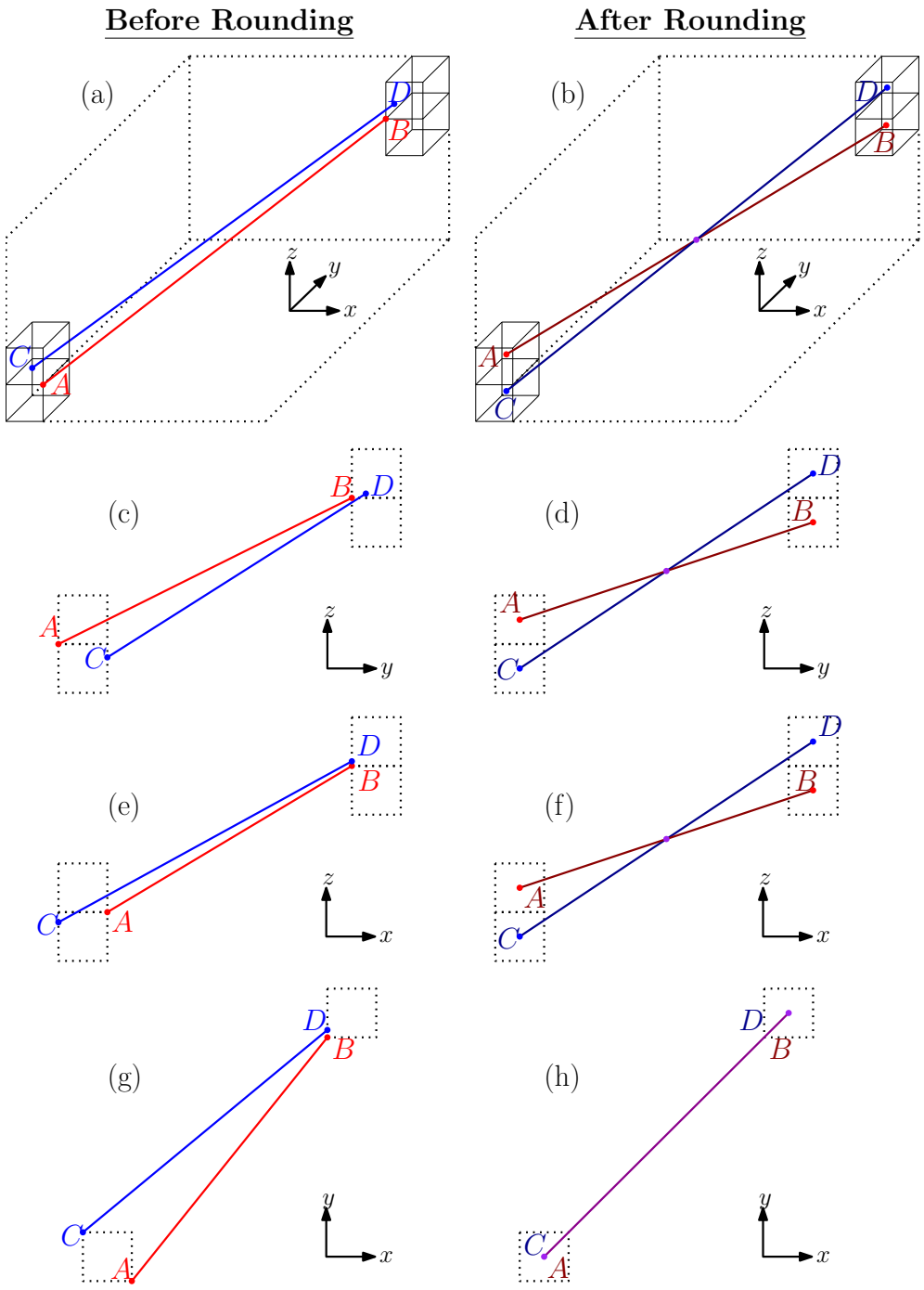


Figure 3.3: Counter-example to Goodrich et al. algorithm [12]: (a) Input segments in 3D. (b) They intersect after rounding. (c-e-g) They do not intersect after projections in the three axis-aligned planes. (d-f-h) They intersect in the projections after the rounding. Segments do not go through any hot voxels and the distance is realized by B and D. $A = (1, \epsilon, 1)$, $B = (6+\epsilon, 6+\epsilon, 4)$, $C = (\epsilon, 1, 0.8)$, $D = (6 + \epsilon, 6 + 3\epsilon, 4 + \epsilon)$ where $0 < \epsilon \ll 1$.

In their 2016 paper on repairing meshes, Zhou et al. [31] provide a databased of rounding issues in computer graphics. Of the 10K models in Thingi10K [32], they repaired 8616 models

with Piecewise-constant Winding Numbers (PWN). Among these models, 3413 models contained self-intersections. Zhou et al. [31] refined these self-intersections and created new vertices that are rounding to double precision floats. According to their experiments, naïve rounding fails about 5% of the time.[18] Due to the lack of a theoretical solution, Zhou et al. [31] propose a heuristic for rounding vertices. Roughly speaking, their approach involves rounding coordinates, computing and refining self-intersections, and repeating this process at most 20 times until no new intersections are created by rounding. More formally, their method involves repeating the following steps while new vertices need rounding: (1) round all vertices to double precision floating-point coordinates, (2) find intersecting triangles (if none, end the loop), (3) round the vertices of these triangles to single precision floating-point coordinates, and (4) compute the self-union of the model to remove self-intersections. In their experimentation, after twenty iterations, only five of the initial 3413 models still contained self-intersections after rounding. In Section 17.2, we reproduced experiments of Zhou et al. [31] and generalize it to all models of Thingi10K, and not only PWN-ones, by remplacing the self-union by subdividing by self-intersections. This heuristic method is currently the state-of-the-art for solving rounding issues in computer graphics, but our experiments show that 9% of Thingi10K models that require non-trivial rounding remain unsolved by this heuristic (see Section 17.2).

In 2019, a new algorithm and implementations were proposed by Milenkovic and Sacks [25]. Their approach consists, after some cleaning as pre-processing, of moving the vertices to increase the minimum distance between pairs of features (the pairs are vertex/face and edge/edge) using several iterations of linear programming. As post-processing, they use gradient descent to minimise the distance between the input position of a vertex and its moved position without reducing the minimum distance between pairs of features. After these operations, they round the vertices and no intersection occurs since the distance between all pairs of features is greater than the rounding drift, thus preserving the topology. The drawback of this approach is that the distance between the input and the output is not bounded, a vertex can drift arbitrarily far. However in practice, Milenkovic and Sacks [25] shows that the drift of the vertices remains small. The tests conducted by Milenkovic and Sacks were mainly based on custom examples and were not evaluated on the more messy models from Thingi10K. As a result, it is difficult to compare the practical effectiveness of their approach with other results.

[18] In their paper, they report a 2.19% failure rate on all models, but only 40% have self-intersections.

Finally, in 2020, Devillers, Lazard and Lenhart provided the first 3D snap rounding algorithm [8] that outputs a set of triangles with integer coordinates such that the Hausdorff distance between the input and the output is at most $\frac{3}{2}$ and the topology is preserved "up to collapse" (see Def. 1). Roughly speaking, this algorithm consists in three operations, each guaranteeing the rounding of one coordinate. One operation subdivides space by vertical parallel slabs and subdivide the model accordingly, ensuring no intersection will occur when rounding the x-coordinates at the end. Another operation collapses faces close to one another along the y-direction, preventing intersection when rounding the y-coordinates at the end. The last operation, similarly to first algorithm of Milenkovic [23] or the following PSL algorithm 3.4, project faces on the xy-plane, subdivide them, and lift the subdivisions back into 3D, ensuring no intersection when rounding the z-coordinates. Even if the proof is very intricate, each operation is quite simple. However the worst-case complexity of the size of the output of this algorithm is really high, $O(n^{13})$, and even if the complexity is better under some assumptions, experiments reveal that this algorithm is impractical [29]. Our algorithm builds on theirs.

3.4 Difficulties of 3D snap rounding

The difficulty of snap rounding faces in 3D is well described by Fortune [11]: Initially, it seems reasonable to round every vertex to the center of the voxel containing it. However, this approach can cause a vertex to traverse a face. To prevent this, it might be necessary to add a vertex on the face beforehand, which requires triangulating it (see Fig. 3.1). Newly formed edges may cross older edges when rounding; to avoid this, new vertices are added to these edges, which in turn requires further triangulating of faces. It is not known whether such schemes terminate, but even if they do, they may terminate after a number of iterations independent of the input size. Milenkovic [25] avoids these subdivisions by moving the two features away from each other without subdividing, but by doing so, the distance between an input vertex and its image can be arbitrarily high.

To better understand the difficulty of the problem, Devillers et al. [8] describe the following algorithm being flawed. Despite its flaws, it is correct on some specific inputs, and we use it as an optimization for our algorithm (see Section 8.2). We refer to it as the PSL algorithm (Project, Subdivide, and Lift). First, (P) project all the input faces onto the floor (xy-plane),

(S) subdivide the projected faces as in the triangulated 2D snap rounding (see Section 2.6), (L) lift this triangulation vertically on all faces, and then round all vertices to the centers of their voxels. For an input of size n, this yields a rounded output of size $\Theta(n^3)$ in the worst case and an L_{inf}-Hausdorff distance of at most $\frac{1}{2}$ between the input faces and their rounded images.

This algorithm guarantees that two faces do not intersect in their interior during the rouding as long as one of them is not vertical at the end of it (Lemma 8). Unfortunately, despite this guarantee, edges may cross after performing the PSL algorithm. Indeed, consider two nearly vertical close triangles whose projections on the horizontal plane are the same triangle that round in 2D to a segment; such triangles in 3D may be rounded into overlapping vertical triangles with crossing edges (see Fig. 3.2). Attempting to post-process these intersections may require creating new vertices, which need to be rounded and may trigger new intersections (the new vertices have no reason to remain in the same vertical plane as their supporting face when rounded). Fortune [11] solved this problem by using a finer grid to round the vertices and avoid vertical rounding of the faces.

Lemma 7. *After the lifting operation of the PSL algorithm, the projection on the floor of any two triangles do not properly intersect at any stage of the rounding.*

Proof. By subdivision and lifting, the projection on the floor of two triangles do not properly intersect at the beginning of the rounding. Due to the subdivision by hot pixels on the floor, a vertex does not properly intersect an edge in projection on the floor during the rounding motion, and therefore, in projection on the floor, they do not properly intersect since they do not at the beginning of the rounding. □

Lemma 8. *After the lifting operation of the PSL algorithm, two triangles that do not become perpendicular to the xy-plane during the rounding do not intersect in their interior during the rounding, unless they become equal.*

Proof. Let A and B be two triangles at the beginning of the rounding, and let A_1, A_2, A_3 and B_1, B_2, B_3 be their vertices. Let $\tilde{A}^t$ and $\tilde{B}^t$, be their image at a moment t of the rounding. Suppose that $\tilde{A}_t$ and $\tilde{B}_t$ are not vertical and intersect in their interiors. Thus, their projection on the floor intersect in their interiors. By Lemma 7, the two triangles are then equal in projection on the floor. Note that for two vertices v and u, since we round to the closest point with

integer coordinates, if $z_v > z_u$, where z_i is the z-coordinate of i before rounding, then $\tilde{z_v}^t \geqslant \tilde{z_u}^t$, where $\tilde{z_i}^t$ is the z-coordinate of i at time t of the rounding. Thus, if the vertex A_1 is above B_1 (they are equal in projection on the floor) before rounding, $\tilde{A}_1^t$ remains above or equal to $\tilde{B}_1^t$. Suppose without loss of generality that triangle A is above triangle B (they do not intersect by assumption), then A_1 (resp. A_2, A_3) is equal or above B_1 (resp. B_2, B_3), thus $\tilde{A}_1^t$ (resp. $\tilde{A}_2^t$, $\tilde{A}_3^t$) remains above or equal to $\tilde{B}_1^t$ (resp. $\tilde{B}_2^t$, $\tilde{B}_3^t$) during rounding. If all their vertices are equal, then $\tilde{A}^t = \tilde{B}^t$. If not, $\tilde{A}^t$ is still above $\tilde{B}^t$ and they do not properly intersect in their interior. □

Chapter 4

Contributions

In this chapter, we summarize our contributions. The first section presents a certified 3D snap rounding algorithm and its implementation. The second section presents optimizations of the previous algorithm that we believe of independant interest. The third section describes an uncertified but highly efficient heuristic for rounding the vertex coordinates of meshes. The last section presents a minor independant contribution to 2D snap rounding.

4.1 A local 3D snap rounding algorithm and its implementation

In Part II, we present the first local 3D snap rounding algorithm. Recall that Devillers et al. [8] presented in 2020 the first and only 3D snap rounding algorithm on a uniform grid. However, a significant drawback of their algorithm is that it is not local, as it subdivides the entire model regardless of where intersections may occur during the rounding. This leads to numerous subdivisions even in simple models, making the algorithm impractical [29].

Building on their work, we developed a local 3D snap rounding algorithm. Our algorithm simulates the rounding process and tags as "critical" all pairs of faces that properly intersect during this simulation. The algorithm then subdivides only the critical faces and those adjacent to them, resulting in a significantly reduced number of subdivisions.

This algorithm is the first that suceeds, in practice, to round naively all faces that do not trigger intersections when rounding and to only subdivide the faces that are close to critical faces. In that sense, this algorithm behaves as one would wish for. However, even though subdivisions are performed only near difficult spots, these subdivisions are numerous and we developped

optimizations to avoid them as much as possible, as discussed in the next section.

With these optimizations, our algorithm is the first 3D snap rounding algorithm that works in practice on non-trivial instances. Indeed, among the 4,524 models of Thingi10K that contain self-intersections, about 90% are easily solved by essentially directly rounding the relevent vertices. Among the 520 remaining models in which proper intersections occur during the rounding motion, our implementation sucessfully solves about 95% and fails on 33 instances. This rate of success is quite similar to the heuristic of Zhou et al. [31] (see Chapter 17). The remaining 33 models on which our algorithm fails in practice are likely due to the fact that our implementation still suffers from instabilities. Indeed, managing the numerous degeneracies that are present in the input and that are created by our algorithm present huge difficulties. While being not mature, our code is a proof of concept and shows that our algorithm is practical and that it can be used to remove self-intersections and round models with topology guarantees.

4.2 Optimizations for the 3D snap rounding algorithm

Based on our experimentations (see Section 16.7), although our algorithm is far more practical than that of Devillers et al. [8], it still lacks efficiency in practice. Indeed, if the number of critical faces increases, the number of subdivisions become numerous too. Therefore, we developped three optimizations for our algorithm (see Chapter 8).

We believe that our first optimization, which locally subdivides edges and faces, is of independent interest for 3D snap rounding. Our second optimization, which subdivides faces that are almost coplanar, revisits already known 3D snap rounding building bricks but it is new in the context of optimization. Our last is a lazy 2D snap rounding optimization, whose interest is mostly practical.

Our first optimization locally subdivides edges and faces that properly intersect during the simulation of rounding. The intuitive idea is very naive: if a vertex properly intersect a triangle during the simulation, we subdivide the triangle so that it does not occur anymore and similarly if two edges properly intersect during the simulation, we subdivide them both. This naive idea is however, not so easy to develop in practice. We designed a subdivision scheme which ensures that two subdivided edges will not intersect during the rounding, and similarly for a face and a vertex. Note that this scheme is only a heuristic because, even though two edges that are subdivided

with this scheme will not intersect when rounded, they may intersect other edges in the scene.

Our second optimization focuses on faces that are almost coplanar. Our experiments show that many meshes present a high number of coplanar intersections during the simulation of the rounding. We designed an optimization specifically for this case: given any number of triangles that become coplanar during the simulation of the rounding, we project them on that plane, subdivide them according to 2D snap rounding (see Section 2.6) and lift back the subdivisions on the input 3D triangles. We subdivide the input triangles according to this optimization and consider the result as input to our main algorithm.

Our third optimization is a lazy 2D snap rounding algorithm. Our algorithm uses a variant of 2D snap rounding that outputs an unrounded triangulation of the input such that no intersections occur during the rounding; this variant inserts hot pixels boundaries to the 2D triangulation, which leads to multiply the number of triangles systematically by a constant factor. We designed a lazy version of this variant that adds hot pixels to the 2D triangulation only if it is needed.

There is no guarantee that these subdivisions manage all rounding issues but their are meant to manage the most simple of them while our certified algorithm manages the difficult ones. Note that those optimizations preserve all distances and topological guarantees of the main algorithm, unlike the heuristic presented in Section 4.3, since they only subdivide the input faces.

4.3 An uncertified but extremely efficient heuristic

In the absence of an efficient certified algorithm that works on "all" cases, we present in Part III a very efficient new heuristic. This heuristic builds on that by Zhou et al. [31]. Given a self-intersecting triangle soup with vertex coordinates represented as double floating-point numbers, our heuristic outputs a simplicial complex that is an intersection-free triangle soup with vertex coordinates also represented as double floating-point numbers.

The heuristic works by first scaling the scene so that it is as large as possible while the integral part of the coordinates are representable as float. Then, the vertices of the pairs of triangles that properly intersect are rounded to the integer grid. We also round to the integer grid all the vertices that lie in the voxels that contain rounded vertices. The self-intersections are then removed by subdividing the triangles accordingly, and newly created vertices are rounded to doubles. This process is repeated at most a fixed number of times as long as the model contains

self-intersections before the scaling back of the model. The distance between a rounded vertex and its original position is bounded by the number of iterations of our heuristic times a constant that depends on the scaling used for defining the grid.

We tested our heuristic on the self-intersecting models of the Thingi10K dataset, and we are the first to produce an intersection-free simplicial complexes for all these models. This heuristic improves on the state-of-the-art results by Zhou et al. [31], which successfully solved about 99% of the self-intersecting models of Thingi10K but only about 90% of the non-trivial models (see Section 17.2).

This heuristic is remarkably simple and effective and we believe that it will become a reference for removing self-intersections in models when guarantees on the topology are not required.

4.4 A floating-point 2D snap rounding algorithm

We also present the first 2D snap rounding algorithm that, given a set of segments in 2D, rounds vertices to the center of their "pixel" when the underlying grid is not uniform. This allow us to round the coordinates directly to floating point numbers instead of rounding on a uniform grid as done by existing algorithms.

Given a set of input segments, the algorithm outputs a simplicial complex with coordinates represented using floating-point numbers (of any precision), ensuring that the Hausdorff distance between an output vertex and its corresponding point on the input is at most one unit of least precision (ulp). In other words the rounding is done with the maximum precision allowed by the floating point number instead of rounding on a uniform grid as done by existing algorithms. Our algorithm also preserves the topology "up to collapse". Essentially, this algorithm subdivides the segments using vertical slabs spaced between two consecutive floating-point numbers, with each slab containing one vertex of the arrangement (a slab has a width of one ulp). The algorithm output complexity is $O(n(n+k))$ vertices where n is the number of input vertices and k the number of intersections of input segments. This simple algorithm is an independent and minor contribution of this thesis and is not related to 3D snap rounding.

Part II

3D mesh rounding algorithm

Chapter 5

Preliminaries and overview of our algorithm

5.1 Introduction

We present the first practical algorithm for rounding a set of interior disjoint polygons into a simplicial complex whose vertices have integer coordinates and which is geometrically close to the input. It is close in the sense that the L_∞ Hausdorff distance between every input face and its rounded image is at most $\frac{3}{2}$. The algorithm actually satisfies the stronger property that, roughly speaking, no point traverses a face during the rounding: formally, there is a continuous motion that moves the input faces to their rounded images so that (i) the L_∞ Hausdorff distance between every input face and its image during the motion never exceeds $\frac{3}{2}$ and (ii) if two points on two faces become equal during the motion, they remain equal through the rest of the motion.

In practical applications, people aim at rounding coordinates to doubles or floats; since our algorithm rounds to integers, we can apply preprocessing and postprocessing scalings so that the final output coordinates are floats or doubles that preserve as many digits as possible in the integral parts of the input coordinates (see Section 5.3 for details).

The main feature of our algorithm is that, in pratice on real-world meshes, our algorithm behaves as one wishes, that is, it rounds naively almost all input faces and it subdivides very few of them. We will see further on that this wish was not fully achieved, which will be the main motivation to define new optimizations we describe after our main algorithm.

This algorithm is implemented and tested, see Part IV and V.

Our main algorithm use the same building blocks as Devillers et al. [8] but we succeed to apply them only in the neighborhoods of faces where trivial rounding induces proper intersections. As a consequence, new subdivision vertices added by our algorithm are only around the problematic faces. However even if we succeed to keep the subdivisions local, the number of subdivisions can be really high even for small inputs (see Section 16.7.2). In consequence, we designed new optimizations to manage "simple" cases. In comparaison, the algorithm by Devillers et al. is believed to create, in practice, $\Theta(n\sqrt{n})$ new vertices in the same time complexity [8,29]. It should be stressed that the crafting of this new algorithm is quite subtle and attention to details is critical for ensuring correction and avoiding cascading subdivisions. Another practical improvment is that we succeed to avoid general position assumptions.

Even though our algorithm is practical, its worst-case complexity is bad: given an input of size $O(n)$, the algorithm outputs a simplicial complex of complexity $O(n^{12})$, in time $O(n^{44})$ and space $O(n^{17})$. In comparaison, the algorithm by Devillers et al. [8] outputs a simplicial complex of complexity $O(n^{13})$, in time and space $O(n^{15})$. None of these bounds is known to be tight in the worst case.

The algorithm described in this section is the one that has been implemented.

5.2 Overview

The algorithm is based on a while loop in which faces are subdivided and that iterates as long as proper intersections occur between faces when simulating the rounding of the vertices to the centers of their voxels. In a favorable scenario where no proper intersections occur, the algorithm does not subdivide any faces and simply rounds the vertices. Otherwise, the loop ends when no proper intersections remain and the vertices are then rounded.

If some pairs of faces properly intersect when simulating the rounding, we work locally by subdividing these faces, called faces in *conflict*, and some faces near by. In a first step, we subdivide all these faces and deform them by performing some local projections along the y-direction, so that no two distinct resulting faces are close to each other along y; we maintain this way the so-called Back-wall invariant. These projections are combined with the creation of so-called *connecting walls* to ensure the connectivity of the resulting deformed faces. This step roughly ensures that any subdivisions of the resulting faces do not properly intersect when

rounding the y-coordinates of their vertices. In the while loop, faces are deformed only in this first step; in the rest of the while loop, faces are only subdivided and triangulated.

In a second step, we subdivide the resulting faces in conflict so that the projections on the xy-plane of any two close-by such faces are either geometrically equal or interior disjoint; this ensures the so-called Floor invariant. This invariant roughly ensures that the *boundaries* of these faces do not properly intersect when rounding the z-coordinates of their vertices.

In a third step, we consider *narrow slabs* that are the regions bounded by the two planes $x = c \pm \frac{1}{2}$ for some integers c. We subdivide each pairs of faces in conflict by the narrow slabs that contain some of their vertices. We also subdivide the close-by faces in conflict in order to maintain the Floor invariant.

We triangulate the faces in the last two steps. In Step 4, we triangulate the faces in the narrow slabs by simulating the 2D-snap rounding described in Section 2.6 of their projections onto the middle plane of the slab, similarly as in [8]. It should be stressed that this process may create some new vertices, called *junction vertices*, on the boundary of these faces, which induce some triangulations of the adjacent faces. Finally, in Step 5, we triangulate the remaining faces by ensuring that any two faces in conflict that are close and that have the same projection on the xy-plane have the same triangulations (without considering the junction vertices).

We repeat all these steps while proper intersections remain when simulating the rounding of the vertices. It should be stressed that when two inner triangulation edges of some faces properly intersect during the simulation of rounding, we never subdivide these edges; instead, we discard the triangulations of these faces and subdivide (or deform) them in the next iteration of the while loop.

When the while loop terminates, we round all vertices to the center of their voxels. No proper intersection occurs during this final rounding of the termination criterion of the while loop.

The key property that ensures the termination of the algorithm is that, in Step 3, each considered narrow slab necessarily contains a vertex of the arrangement of the edges that appear in Step 1, after their projections on the xy-plane. The number of considered narrow slabs can thus be bounded in terms of the size on the input. The termination of the algorithm essentially follows because it is straightforward to bound the number of faces obtained after Steps 1 and 2 in terms of the input size and these faces are only subdivided by narrow slabs and triangulated.

The while loop is detailed in Chapter 6 and the rounding scheme in Chapter 7. Optimizations

are detailed in Chapter 8 and predicates are detailed in Chapter 9. The proof of correctness and termination is done in Chapter 10 and we provide a worst-case complexity analysis in Chapter 11.

5.3 Notation and preliminaries

Intersections. Two polygons, edges, or vertices are said to *properly intersect* if their intersection is non-empty and not a common face, edge or vertex of both.

Floor, back wall and side walls. The coordinates in the Euclidean space $\mathbb{R}^3$ are referred to as x, y, and z and $\vec{\imath}, \vec{\jmath}, \vec{k}$ is the canonical basis. We use several planes parallel to the axes: the xy-plane is called the *floor*, the xz-plane is called the *back wall* and a plane parallel to the yz-plane is called a *side wall*. Projections on the floor, back wall and side walls are always considered orthogonal to the plane of projection.

Distances. Unless specified otherwise, the considered distances are L_∞ distances. The y-distance between two objects (i.e., sets) is the infimum between ∞ and the distances between any two points, one in each set, that coincide in projection on the back wall. Similarly, the y-maximum-distance between two objects is the supremum between $-\infty$ and the distances between any two points, one in each set, that coincide in projection on the back wall. Similarly for x- and z-distances.

Scaling. Our algorithm rounds vertices to points with integer coordinates. However, in most applications, vertices are to be rounded onto a grid of fixed-precision floating-point coordinates. In such contexts, we apply preprocessing and postprocessing scalings such that the integer rounding preserves all digits of the integral parts of the input coordinates and as many as possible digits in their fractional parts.

Let m be the length of the mantissa of the output coordinates (53 bits with double precision). Let M be the maximum absolute value of all input coordinates and let $b = \lfloor \log_2 M \rfloor$. We multiply all input coordinates by 2^{m-b} (in exact arithmetics). We thus obtain coordinates whose absolute values will round to integers with at most m bits. We do the reverse scaling at the end of the algorithm, so this scaling only plays a role when rounding the coordinates (either at the end or

in the termination criterion).[19]

Pixels and voxels. Pixels of $\mathbb{R}^2$ and voxels in $\mathbb{R}^3$ are defined as unit squares and unit cubes centered on vertices of the integer grid. They are defined as cartesian products of intervals that are open on their left sides and closed on their right sides.

Narrow slabs. We define some so-called *narrow slabs* $\mathcal{N}$ as the regions bounded by the two planes $x = c \pm \frac{1}{2}$ for some integers c. We will create some vertices on the boundary walls of the narrow slabs and, to ensure these vertices round to the middle planes of these slabs, we actually define their leftmost planes as $x = c - \frac{1}{2} + \epsilon$ with $\epsilon > 0$ sufficiently small. (There is no need to do the same for their rightmost planes since $c + \frac{1}{2}$ is rounded to c by convention.)

Sets of input and subdivided faces: $\mathcal{F}, \mathcal{F}_C, \mathcal{G}, \mathcal{G}_C$. Let $\mathcal{F}$ be the set of input polygonal faces in 3D that do not properly intersect. Let $\mathcal{F}_C \subset \mathcal{F}$ be the set of input faces in *conflict*, that is those that induce some proper intersections when simulating the rounding in the Termination criterion and in Step 1. Initially, $\mathcal{F}_C$ is empty and it is greedily augmented in the while loop as long as intersections occur in the simulation of rounding.

In the while loop, $\mathcal{G}$ is a set of faces obtained after (possibly) modifying (in Step 1) and subdividing (in Steps 2 to 5) the faces of $\mathcal{F}$. The faces in $\mathcal{G}$ are polygons that are eventually equipped with triangulations. Initially, $\mathcal{G}$ is equal to $\mathcal{F}$ and, at the end of the while loop, the triangulated faces in $\mathcal{G}$ are output after the rounding of their vertices. Each face of $\mathcal{F}$ is naturally mapped to a set of faces in $\mathcal{G}$, called the *image* of F in $\mathcal{G}$. The image of $\mathcal{F}_C$ in $\mathcal{G}$ is denoted $\mathcal{G}_C$. Note that the faces of $\mathcal{F} \setminus \mathcal{F}_C$ are not subdivided by the algorithm but those adjacent to faces in $\mathcal{F}_C$ may be triangulated (in $\mathcal{G}$).

[19] It may be relevant to consider a larger scaling in x than in y and z, which is similar to considering a finer grid in x than in y and z. Indeed, consider the two segments of intersection of a face with the two side walls of a narrow slab. If the grid is very coarse in x (with respect to the coarseness in y and z), these two segments will typically have endpoints in distinct pixels and they thus will be rounded (typically) into distinct segments; this will create some visually unpleasant faces normal to the x-axis and will increase the size of the output. On the other hand, if the grid is very fine in x, the two segments will typically have their endpoints in the same two pixels and they will be rounded to the same segment, avoiding the above drawbacks. Similarly, edges spanning a narrow slab will often be rounded to points if the grid is much finer in x than in y and z.

Chapter 6

Backbone of our algorithm: a while loop

Recall that, before the while loop, $\mathcal{G}$ is initialized to the set of input faces $\mathcal{F}$ and $\mathcal{G}_C$, the image in $\mathcal{G}$ of the input faces in conflict, is empty. We maintain the following invariants, which hold at the beginning by assumption on the input faces and since $\mathcal{G}_C$ is initially empty.

Invariant on $\mathcal{G}$. The faces in $\mathcal{G}$ do not properly intersect.

Back-wall invariant on $\mathcal{G}_C$. The projections on the back wall of any two faces in $\mathcal{G}_C$ at y-distance less than 1 do not intersect in their interiors.

Floor invariant on $\mathcal{G}_C$. Any two faces in $\mathcal{G}_C$ at distance[20] at most 1 and whose projections on the floor intersect in their interiors have geometrically equal boundaries on the floor.

6.1 Termination criterion of the loop and faces in conflict

We iterate the while loop as long as some[21] (triangulated) faces of $\mathcal{G}$ properly intersect during the simulation of the rounding of their vertices (precisely defined in Chapter 7). This is done by iterating the while loop as long as (a) a vertex of $\mathcal{G}$ properly intersects a (triangulated) face of $\mathcal{G}$ during the simulation of rounding or (b) two (boundary or triangulation) edges of $\mathcal{G}$ properly

[20] It is important to consider the distance instead of the z-distance. Indeed, consider in the yz-plane a segment that intersects the left and bottom sides of a pixel and that sweeps over the pixel's center when rounded; a vertex in that pixel near its top-right corner properly intersects the segment when rounding although their y- and z-distances are larger than 1.

[21] Note that a single triangulated face may self intersect during the simulation of rounding.

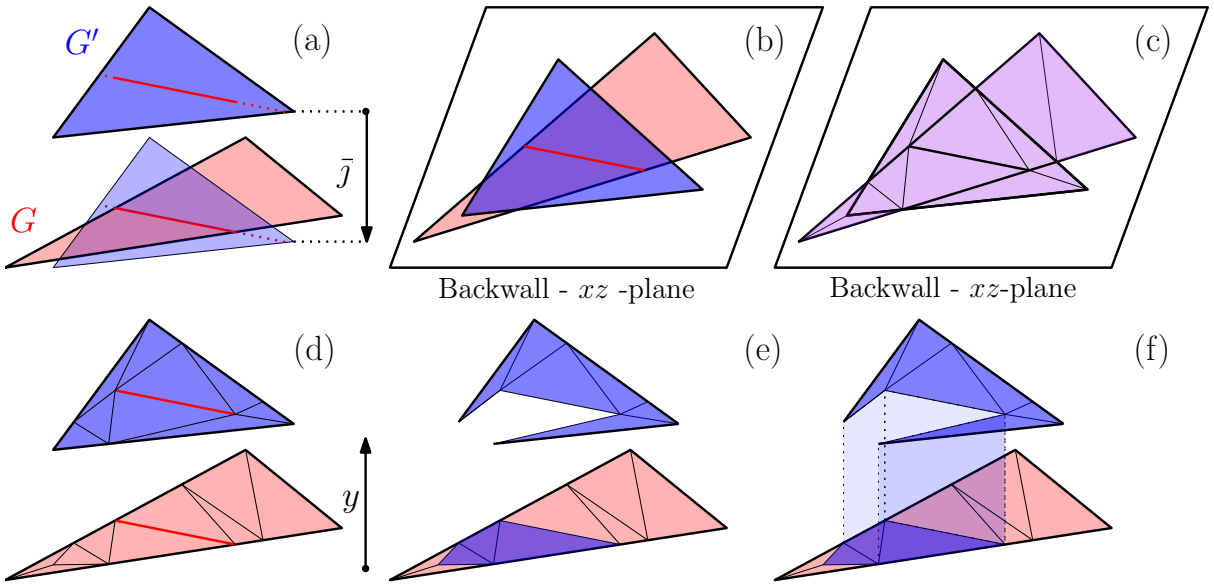


Figure 6.1: Step 1 projection for two faces. (a) Intersect G by the translated copie of G' by $\pm\vec{j}$. (b) Project the faces on the backwall. (c) Compute and triangulate the arrangement on the backwall. (d) Lift this arrangement back on the faces. (e) Project on G the triangles of G' at y-distance less than one from G and (f) create connected walls between these triangles and their projections (somes are not drawn for readibility).

intersect during the simulation of rounding. We will prove in Lemma 19 that these two criteria are equivalent.

We define here $\mathcal{F}_C$, the set of input faces that induce some proper intersections when simulating the rounding in some iteration of the while loop from the first to the current iteration. Let $\mathcal{F}_C^{\text{old}}$ be the set $\mathcal{F}_C$ at the end of the previous iteration of the loop; $\mathcal{F}_C^{\text{old}}$ is empty at the first iteration. Let $\mathcal{F}_C^{\text{new}}$ be the set of new input faces involved in the termination criterion. Precisely, $\mathcal{F}_C^{\text{new}}$ is the set of faces in $\mathcal{F} \setminus \mathcal{F}_C^{\text{old}}$ that support a face of $\mathcal{G}$ involved in the termination criterion, that is, the faces intersected by a vertex in the intersections of type (a) and the faces that are incident or support the edges in the intersections of type (b). We define $\mathcal{F}_C$ as $\mathcal{F}_C^{\text{old}} \cup \mathcal{F}_C^{\text{new}}$. Recall that $\mathcal{G}_C$ is the image of $\mathcal{F}_C$ in $\mathcal{G}$.

6.2 Step 1: y-projection of the faces of $\mathcal{F}_C$

If $\mathcal{F}_C^{\text{new}}$ is empty, we skip this step. Otherwise, we start by reinitializing $\mathcal{G}$ to $\mathcal{F}$.[22] The operation we perform in this step is roughly speaking that, for each pair of faces F, F' in $\mathcal{F}_C$, we project,

[22] We reinitialize the whole set $\mathcal{G}$ to $\mathcal{F}$ for simplicity, in order to avoid that sub-faces are projected more than once in distinct instances of Step 1 during the while loop.

along y on F, the part of F' that consists of all the points that are at distance less than 1 from their projections along y on F (see Figure 6.1). The goal is that, before rounding at the end, any two distinct faces in $\mathcal{G}_C$ are at y-distance at least 1 (whose projections on the back wall intersect in their interior). In the proof of correctness (Lemma 20), we consider that these projections are done simultaneously to avoid intersections, but we define the projections in an iterative way.

Refer to Figure 6.1. (a) For any pair of faces in $\mathcal{G}_C$ that are within y-distance at most one, compute the intersections of the first face with the translated copies of the other face by $\pm\vec{\jmath}$ and subdivide them accordingly.[23] (b) Project all the faces of $\mathcal{G}_C$ onto the backwall and (c) compute their triangulated arrangement. (d) Lift this arrangement back onto the faces of $\mathcal{G}_C$ along the y-direction and subdivide them accordingly. After these steps, any pair of faces in $\mathcal{G}_C$ will either be interior disjoint or equal in their projection onto the backwall. Additionally, for any pair of faces in $\mathcal{G}_C$ at distance less than one, any subfaces of them equal in their projection are at distance less than one.

Now, in lexicographic order of (i, j), we consider every pair of faces G_i and G_j in $\mathcal{G}_C$ that are at y-distance less than 1 and whose projections on the back wall intersect in their interiors (their projections are thus equal). For every such pair, if a face $G \in \mathcal{G} \setminus \mathcal{G}_C$ intersects the interior of their convex hull, we add its supporting face (in $\mathcal{F}$) to $\mathcal{F}_C^{\text{new}}$ and restart Step 1. Otherwise, we project G_j onto G_i. By projecting, we mean that (e) we replace in $\mathcal{G}_C$ the face G_j by a copy of G_i and (f) we create in $\mathcal{G}_C$ *connecting walls* defined as the faces of the convex hull of G_j and G_i, other than these two.[24]

The connecting walls (and the input faces that are parallel to the y-direction) may overlap in 3D, in which case we subdivide them to ensure that the Invariant on $\mathcal{G}$ holds. The connecting walls can also form "winglets" if two adjacent triangles are both projecting in the same direction (see Figure 6.2). These winglets are not problematic for the algorithm and could be accepted. However, they are unnecessary, non-manifold, and "unesthetic". Avoiding them does not affect the complexity of the algorithm and simplifies the output. Therefore, we delete all connected

[23] For simplicity, we describe this operation as an initial subdivision. However, in practice, we compute the segments on G_i and G_j that are at a y-distance one, add their projections to the subsequent backwall arrangement, and perform the subdivision during the lifting of this arrangement.

[24] An alternative to avoid the reinitialization of $\mathcal{G}$ at the beginning of Step 1 is to consider the order in which faces of $\mathcal{F}$ are added to $\mathcal{F}_C$ and to consider the pairs of faces (G_i, G_j) not in the lexicographic order of (i, j) but in the lexicographic order of (i', j') where the i'-th and j'-th faces added to $\mathcal{F}_C$ are those supporting G_i and G_j, respectively. This ensures that, if G_j is projected onto G_i in a Step-1 instance of the while loop, they will not be projected again in a later instance of Step 1.

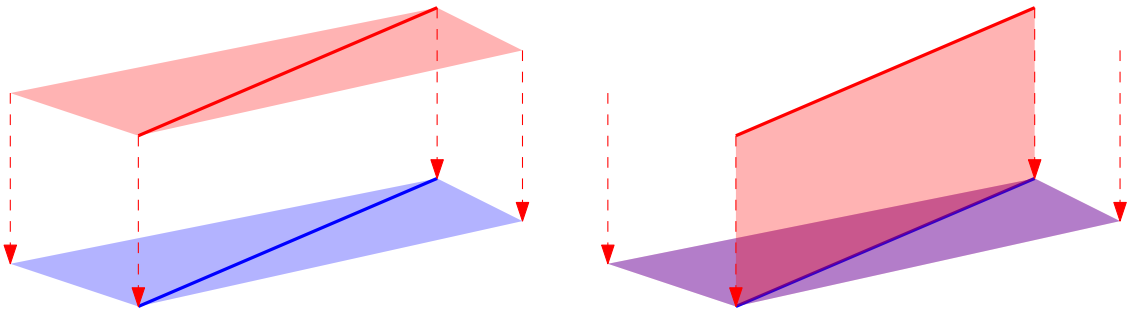


Figure 6.2: The faces adjacent to the red edge are both projected, and the resulting connecting walls form a "winglet". For simplicity and aesthetic purposes, these winglets are deleted.

walls that form winglets[25].

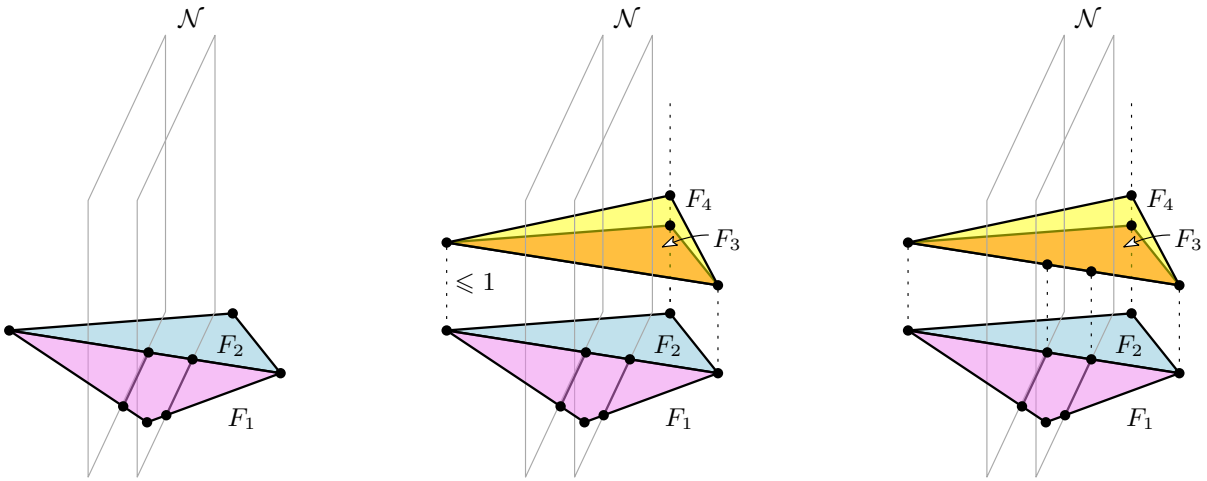
The Back-wall invariant on $\mathcal{G}_C$ holds by construction at the end of this step. It will also hold at the end of the loop iteration because the faces of $\mathcal{G}$ are only subdivided and triangulated in Steps 2 to 5.

Note that this version of Step 1 is the original. The first iteration of this component was more localized, processing pairs of critical faces in lexicographic order. It computed the region within a y-distance of at most one, and then the faces were subdivided accordingly in $\mathcal{G}_C$. Subsequently, the faces in $\mathcal{G}_C$ located at a y-distance of at most one were projected, and the connecting walls were created. Although conceptually very similar, this initial version was difficult to implement (see Section 14.8.3), which motivated modifications to the algorithm.

6.3 Step 2: Subdivision of faces in $\mathcal{G}_C$ by the vertical projections of close-by faces in $\mathcal{G}_C$

The goal of this step is to recover the Floor invariant on $\mathcal{G}_C$ after Step 1. We thus skip this step if Step 1 was skipped. We project all the faces of $\mathcal{G}_C$ on the floor and compute their triangulated arrangement. Then, we lift this arrangement back on the faces of $\mathcal{G}_C$ along the z-direction and subdivide them accordingly.

[25] A connected wall W is considered part of a winglet if one of its edges, not parallel to $\vec{\jmath}$, is not adjacent to any face geometrically different from W.



(a) Subdivision of F_1. (b) Before Rule (iv). (c) After Rule (iv).

Figure 6.3: Step 3: (a) Example of subdivision by a narrow slab $\mathcal{N}$: F_1 is subdivided into 3 sub-faces and the boundary of F_2 is augmented by two subdivision vertices. (b-c) Example where Subdivision rule (iv) applies. Faces $F_1, \ldots, F_4$ are in $\mathcal{G}_C$ and, in projection on the floor, F_2, F_3, F_4 are geometrically equal and they are interior disjoint from F_1.

6.4 Step 3: Subdivision of the faces in $\mathcal{G}_C$ by narrow slabs

We subdivide faces by narrow slabs according to the following rules depending on the proper intersections detected during the simulation of rounding of their vertices in the Termination criterion.[26]

(i) If a vertex v of $\mathcal{G}$ properly intersects a face $G \in \mathcal{G}_C$ during the simulation of rounding, we discard the triangulation of G and subdivide it (in $\mathcal{G}_C \subset \mathcal{G}$) by the boundary walls of the narrow slab that contains v. We also label the vertex v as a *narrow vertex*.

(ii) If two (boundary or triangulation)[27] edges of $\mathcal{G}_C$ properly intersect during the simulation of rounding, we discard the triangulation of their (incident or supporting) faces and we subdivide these faces (in $\mathcal{G}_C \subset \mathcal{G}$) by the boundary walls of the narrow slabs that contain their endpoints.[28]

[26]Note that, in the Termination criterion, if two boundary edges of $\mathcal{G}$ properly intersect during the simulation of rounding and if these two edges properly intersect in projection on the floor, then they are subdivided in Step 2 and it is likely that in practice the new edges will not properly intersect during the rounding any more. So practically, we can perform Steps 3 to 5 only when $\mathcal{F}_C^{\text{new}}$ is empty, that is once the faces in conflicts have already been (possibly) subdivided in a previous iteration.

[27]If Steps 1 and 2 were not skipped in the current loop iteration, all faces in $\mathcal{G}_C$ are triangles. However, otherwise, faces of $\mathcal{G}_C$ have been subdivided and triangulated in the previous loop iteration.

[28]In fact, we only need to consider the narrow slabs that contain the two non-x-extreme endpoints of the two edges, but for clarity we consider all four endpoints.

The intersections of such faces by such narrow slabs are called *narrow faces.*[29] The other faces in $\mathcal{G}$ are called *wide faces.* When a face $G \in \mathcal{G}_C$ is subdivided by a narrow slab $\mathcal{N}$, its edges are naturally subdivided accordingly. If such an edge e bounds another face $G' \in \mathcal{G}$ (that is not subdivided by $\mathcal{N}$), the edge e is subdivided in G' by adding vertices on it (see Figure 6.3(a)). The vertices induced by these subdivisions are called *subdivision vertices.*

We also consider the following (recursive) subdivision rule to ensure that the Floor invariant on $\mathcal{G}_C$ holds.

(iii) When a face $G \in \mathcal{G}_C$ is subdivided by a narrow slab $\mathcal{N}$ in the Subdivision rules (i), (ii) or (iii), we also subdivide by $\mathcal{N}$ all faces G' in $\mathcal{G}_C$ that are at distance at most 1 from G and that are equal to G in projection on the floor. As in Subdivision rules (i-ii), the triangulations of the faces G' are discarded before subdividing them. As above, faces $G' \cap \mathcal{N}$ are called narrow faces and the vertices created in the process are called subdivision vertices.

Further edge subdivisions. Two faces in $\mathcal{G}_C$ that are at distance at most 1 and that are geometrically equal in projection on the floor may have subdivision vertices that do not coincide on the floor; see Figure 6.3(b). We further perform some edge subdivisions to ensure that such subdivision vertices coincide on the floor; see Figure 6.3(c).

(iv) Ensure that two (wide or narrow) faces in $\mathcal{G}_C$ at distance at most 1 and that intersect in their interiors on the floor have consistent subdivision vertices. For each pair of faces G and G' in $\mathcal{G}_C$ at distance at most 1 and that intersect in their interiors on the floor (they are geometrically equal on the floor by (iii)), subdivide their boundary edges so that their subdivision vertices[30] project vertically on one another. As in Subdivision rules (i-iii), the triangulations of the faces are discarded.[31]

[29] If the input face supporting a face $G \in \mathcal{G}_C$ has already been subdivided by a narrow slab $\mathcal{N}$ and if $G \cap \mathcal{N}$ is an edge of G, there is no need to define the (degenerate) narrow face $G \cap \mathcal{N}$, but we nonetheless always do so for the sake of simplicity (of the proofs of Lemmas 23 and 24).

[30] It should be stressed that we only project subdivision vertices but not junction vertices (possibly defined in Step 4 of previous loop iterations). This is critical for the proof of termination in Lemma 25 (see Foonote 44). Indeed, if junction vertices were also projected, the recursion is not known to terminate.

[31] Discarding the triangulation is not actually required but it simplifies the description of Step 5.

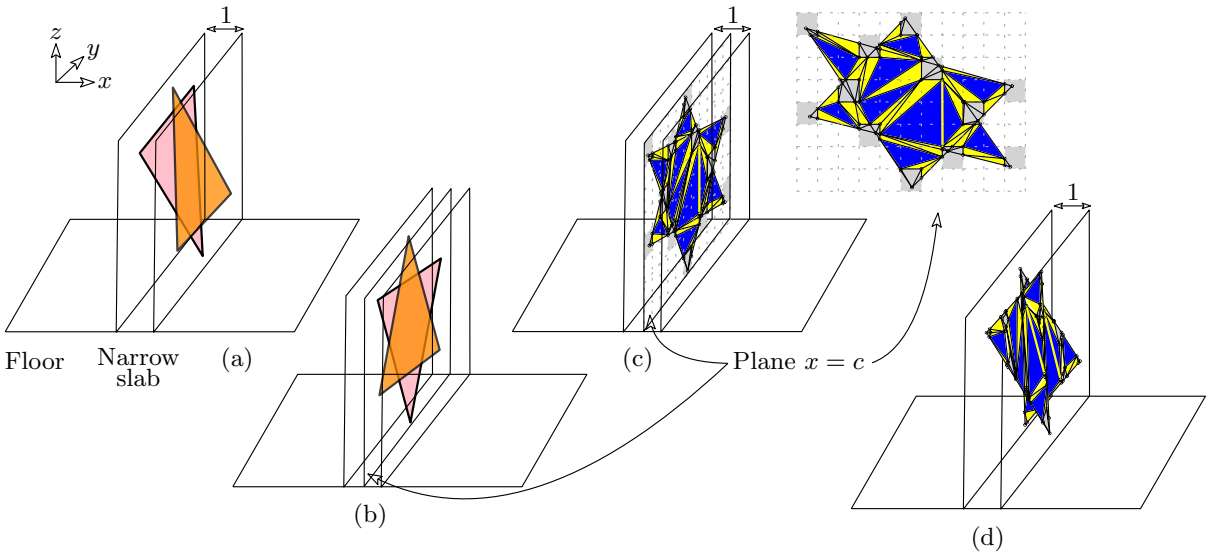


Figure 6.4: Step 4: Triangulation of the narrow faces (Devillers et al. [8]). (a) Narrow faces in a narrow slab. (b) Their projections onto the middle side wall. (c) In that plane, the edges are split as in 2D snap rounding, the boundaries of the hot pixels are added, and the faces are triangulated. (d) The triangulation is lifted back on the 3D faces.

6.5 Step 4: Triangulation of the narrow faces in $\mathcal{G}_C$: 2D-snap simulation in the narrow slabs

Consider all the narrow slabs that are involved in the Subdivision rules (i-ii). For each slab, consider all the narrow faces (in $\mathcal{G}_C$) and narrow vertices within it, and discard their triangulations. We then triangulate these faces by simulating a 2D triangulated snap rounding of their projections onto the side wall, as described by Devillers et al. [8], which we detailed in Section 2.6. Refer to Figure 6.4 for a visual representation.

For each slab, project all the narrow faces and narrow vertices in $\mathcal{G}_C$ along the x-axis onto the side wall. In this side wall, subdivide all the triangles according to the 2D triangulated snap rounding method. Lift this triangulation back along the x-axis onto all the narrow faces within the slab and triangulate them accordingly without subdividing (in $\mathcal{G}_C$). Any new vertices created on the boundary edges of the faces are called *junction vertices*.[32] These junction vertices may thus appear on the boundary of adjacent wide faces, which then need to be re-triangulated accordingly as follows.

[32]When triangulation of a face of $\mathcal{G}_C$ is discarded, its junction vertices are discarded too.

6.6 Step 5: Triangulation of the wide faces

We triangulate the wide faces in $\mathcal{G}_C$ involved in the Subdivision rules (i–iv) as well as the faces in $\mathcal{G} \setminus \mathcal{G}_C$ on which subdivision vertices have been added by these rules. However, we do not consider yet the junction vertices (defined in Step 4) in these triangulations. Furthermore, we ensure that any two faces in $\mathcal{G}_C$ that are at distance at most 1 and that intersect in their interiors on the floor have the same triangulation on the floor. This can be done since, in projection on the floor, the boundaries of the faces are geometrically equal by the Floor invariant on $\mathcal{G}_C$ and they have the same vertices (except for junction vertices) by Subdivision rule (iv). Then, for each junction vertex, we add a new triangulation edge in order to re-triangulate the wide face. If the re-triangulation is not unique, we consider any arbitrarily. We call these new edges *junction-induced edges*. We have the following property.

Step-5 property. Any two wide faces in $\mathcal{G}_C$ at distance at most 1 and that intersect in their interiors on the floor have, on the floor, the same triangulation except for their junction-induced edges (and without considering their junction vertices).

6.7 Recursion

Since we subdivided and triangulated faces in $\mathcal{G}$, new vertices and edges may induce new proper intersections when rounding. We thus recurse while intersections are detected in the Termination criterion of the while loop. Note that the Invariant on $\mathcal{G}$ holds at the end of each iteration since it holds at the end of Step 1 and if two edges or faces are equal, they are subdivided in the same way. The Back-wall invariant on $\mathcal{G}_C$ also holds since it holds at the end of Step 1 and the faces of $\mathcal{G}$ are only subdivided and triangulated in Steps 2 to 5. Finally, the Floor invariant on $\mathcal{G}_C$ holds since it holds right after Subdivision rule (iii) and the interior of the faces in $\mathcal{G}$ are not subdivided afterward (only their boundary edges may be subdivided.)

Chapter 7

Rounding motions and collision detection

A significant aspect of our algorithm lies in the simulation of the rounding and the detection of intersections that arise during this simulation. Thus, the motion of the vertices is an integral part of our algorithm, rather than solely serving as a proof tool, which was the case in previous algorithms. The choice of this motion involves a trade-off between the simplicity of the tests and the frequency of collisions triggered by the motion.

In this chapter, we describe precisely the rounding motion and the detection of intersections during the simulation of the rounding. In the next section, we present the most natural but not so practical rounding motion. In Section 7.2 and 7.3, we present a more practical coordinate-by-coordinate motion and how to detect intersection during this rounding motion. We describe an improvement of the coordinate-by-coordinate motion in Section 7.4 and the detection of proper intersections during this motion is detailed in Section 7.5. Finally, we argue that some proper intersections are ignorable in Section 7.6.

7.1 Standard linear motion

The standard scheme for moving a vertex to the center of its pixel/voxel consists in moving it at constant linear speed (referred to here as the *standard linear motion*). However it is difficult to test collisions with this motion.

Indeed, consider the scenario where we want to test the intersection of two edges (disregarding

degenerate cases) during the standard linear motion. When these two edges intersect at time t, they become coplanar. Potential intersections can therefore be detected by computing the determinant of the four vertices involved. Subsequently, we test if the two segments intersect in 3D space at times corresponding to the roots of this determinant.

The determinant formed by the four points at time t is expressed by the following determinant:

$$D_t = \begin{vmatrix} a_x(1-t)+(\bar{a}_x)t & a_y(1-t)+(\bar{a}_y)t & a_z(1-t)+(\bar{a}_z)t & 1 \\ b_x(1-t)+(\bar{b}_x)t & b_y(1-t)+(\bar{b}_y)t & b_z(1-t)+(\bar{b}_z)t & 1 \\ c_x(1-t)+(\bar{c}_x)t & c_y(1-t)+(\bar{c}_y)t & c_z(1-t)+(\bar{c}_z)t & 1 \\ d_x(1-t)+(\bar{d}_x)t & d_y(1-t)+(\bar{d}_y)t & d_z(1-t)+(\bar{d}_z)t & 1 \end{vmatrix}$$

where $(\bar{a}_x, \bar{a}_y, \bar{a}_z)$ is the center of the voxel containing (a_x, a_y, a_z) and similarly for the others.

This determinant is a polynomial of degree 3 in terms of t. Therefore, the determinant can change sign three times throughout the motion. Consequently, solely checking the orientation at the beginning and end is insufficient to determine if an intersection occurs during the motion. While it is possible to determine the roots of this polynomial, doing so incurs a substantial computational cost, especially considering our aim for certified results. For these reasons, we opt to utilize alternative motions.

7.2 Coordinate-by-coordinate linear motion

Instead of the standard linear motion, a simple motion (refered to as *coordinate-by-coordinate linear motion*) is to separate the motion in three steps, each step rounding one coordinate. The tests are much simpler with this motion. We can simply test the orientation at the beginning and the end of the motion steps to detect a potentiel intersection (see Section 7.3 for more details). It is also much more convenient for proofs. In general, it is more convenient to deal with this motion rather than the standard linear one. The drawback of this motion is that it can trigger a lot of "unnecessary" proper intersections (see Fig. 7.1); we detect proper intersections with this motion that we will not see with the standard linear motion.

More formally, we define the motion cbc_p of a point $p = (x, y, z)$ from $p_0 = (x_0, y_0, z_0)$ to $p_r = (x_r, y_r, z_r)$ as a continuous fonction $cbc : \mathbb{R} \to \mathbb{R}^3$ such that $cbc_p(0) = p_0$ and $cbc_p(3) = p_r$.

We define the *coordinate-by-coordinate linear* motion as follows:

- Step 1: the y- and z-coordinates of $cbc_p(t)$ are invariant for $t \in [0,1]$ and its x-coordinate is equal to $x_r t + x_0(1-t)$ for $t \in [0,1]$.
- Step 2: the x- and z-coordinates of $cbc_p(t)$ are invariant for $t \in [1,2]$ and its y-coordinate is equal to $y_r(t-1) + y_0(2-t)$ for $t \in [1,2]$.
- Step 3: the x- and y-coordinates of $cbc_p(t)$ are invariant for $t \in [2,3]$ and its z-coordinate is equal to $z_r(t-2) + z_0(3-t)$ for $t \in [2,3]$.

It should be stressed that the order x, y, z of the motion is critical for the proofs of correctness presented in Chapter 10 and it may also impact the output of the algorithm since, depending on the order, two features may intersect or not in the Termination criterion.

7.3 Coordinate-by-coordinate intersection test

To explain the simplicity of the intersection test between two edges for the coordinate-by-coordinate motion, we provide a brief overview. We can independently test the intersection at each step. We define the test for the x-coordinate; the test is similar for the other coordinates. The orientation of the four vertices during the motion of x can be expressed as:

$$O_t = sign \begin{vmatrix} a_x(1-t) + (\bar{a}_x)t & a_y & a_z & 1 \\ b_x(1-t) + (\bar{b}_x)t & b_y & b_z & 1 \\ c_x(1-t) + (\bar{c}_x)t & c_y & c_z & 1 \\ d_x(1-t) + (\bar{d}_x)t & d_y & d_z & 1 \end{vmatrix} = sign(P(t)) \text{ with } P \text{ of degree 1.} \quad (7.1)$$

As the determinant is linear in t, it becomes null if and only if the orientation has changed between the start and the end of the motion step.[33] Hence, we only need to test the orientation at these two instances to determine if there is a potential intersection. Furthermore, the projection of the edges on the sidewall remains invariant during the motion. Consequently, we only need to check whether the two edges intersect in the projection on the sidewall (feasible using a 2D orientation test). There is no necessity to compute it at the precise moment of intersection (see

[33] We will not delve here into the case where they are coplanar throughout the entire motion.

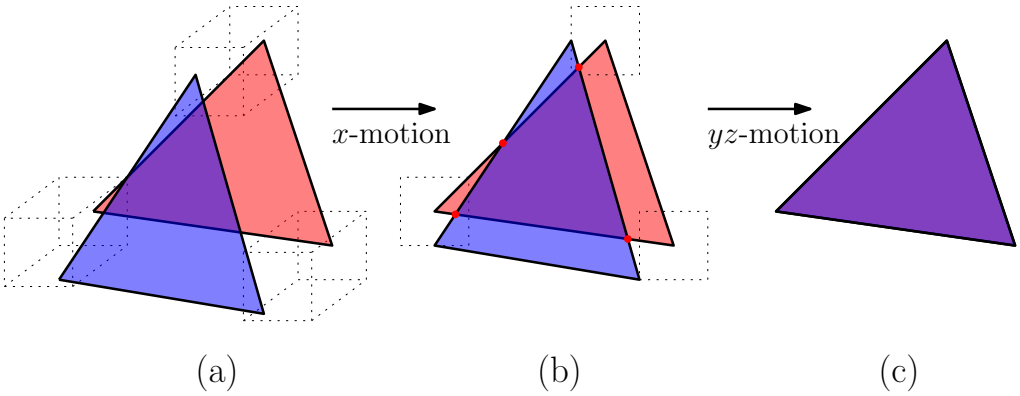


Figure 7.1: (a) Two input triangles in 3D, (b) these triangles after the x-motion, (c) these triangles after the rest of the motion. These triangles properly intersect during the rounding with the coordinate-by-coordinate motion while they do not with the standard linear motion.

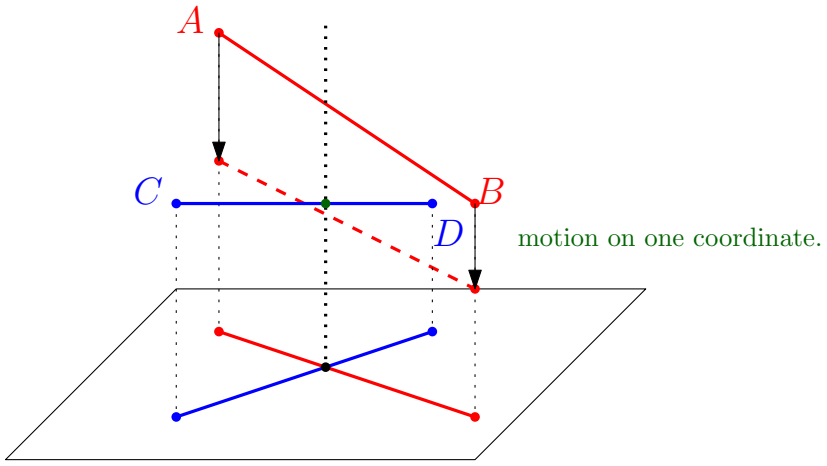


Figure 7.2: Two input edges $[AB]$ and $[CD]$ during a motion step and their projection onto the plane perpendicular to the motion axis. The arrows indicate the movement of the vertices of $[AB]$, and the dashed line represents the rounded version of $[AB]$ after the motion step. The two edges properly intersect during the motion step if they intersect in the projection, and if the orientation of the four vertices changes during the motion step.

Fig. 7.2). This straightforward test can be executed solely with orientation data on input and output coordinates that are rational.

7.4 Trimmed coordinate-by-coordinate linear motion

To enhance the coordinate-by-coordinate linear motion, we introduce a *trimmed coordinate-by-coordinate linear motion*. This motion comprises four steps: each of the first three steps nearly complete the rounding of one coordinate up to an infinitesimal factor and the final step completes the motion with an infinitesimal standard linear motion (Fig 7.3). The tests involved are no more

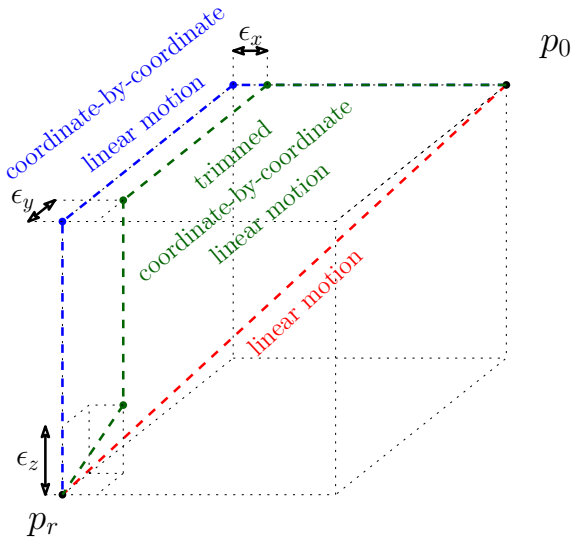


Figure 7.3: The 3 differents motions.

complex than those in the coordinate-by-coordinate linear motion but they result in a reduced number of intersections.

More formally, we define the motion $tcbc_p$ of a point $p = (x, y, z)$ from $p_0 = (x_0, y_0, z_0)$ to $p_r = (x_r, y_r, z_r)$ as a continuous fonction $f : \mathbb{R} \to \mathbb{R}^3$ such that $tcbc_p(0) = p_0$ and $tcbc_p(4) = p_r$. We define the *trimmed coordinate-by-coordinate linear* motion as follow for $0 < \epsilon \ll 1$.

- Step 1: the y- and z-coordinates of $tcbc_p(t)$ are invariant for $t \in [0, 1]$ and its x-coordinate is equal to that of $cbc_p(t)$ for $t \in [0, 1 - \epsilon^4]$ and is invariant for $t \in [1 - \epsilon^4, 1]$.

- Step 2: the x- and z-coordinates of $tcbc_p(t)$ are invariant for $t \in [1, 2]$ and its y-coordinate is equal to that of $cbc_p(t)$ for $t \in [1, 2 - \epsilon^2]$ and is invariant for $t \in [2 - \epsilon^2, 2]$.

- Step 3: the x- and y-coordinates of $tcbc_p(t)$ are invariant for $t \in [2, 3]$ and its z-coordinate is equal to that of $cbc_p(t)$ for $t \in [2, 3 - \epsilon]$ and is invariant for $t \in [2 - \epsilon, 3]$.

- Step 4: We move p to p_r with a standard linear motion for $t \in [3, 4]$.

This motion ensures that for any four non-coplanar points, they remain non-coplanar at the end of Steps $1, 2, 3$ of the rounding process (Corollary 11). Indeed, since the movement is trimmed, if four vertices are about to become coplanar, it halts just before this occurs, and the motion is only completed during Step 4. This reduces the number of intersections during the motion at a minimal cost.

To formalize this, considering a vertex v, with positions v_0, v_1, v_2, v_3, v_4 at times $0, 1, 2, 3$, and 4 respectively. We establish the following lemma:

Lemma 9. *Let* $M = \begin{bmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{bmatrix}$ *be the matrix of 4 vertices* (a, b, c, d). *Let* M_0, M_1, M_2, M_3 *be the matrix* M *at the corresponding time of the motion. Let* $D_\star = \det(M_\star)$. *The determinants of* $D_1, \ldots, D_3$ *satisfy:*

$$D_1 = D_x + \epsilon^4(D_0 - D_x), \tag{7.2}$$

$$D_2 = D_{xy} + \epsilon^2(D_x - D_{xy}) + \epsilon^4(D_y - D_{xy}) + \epsilon^6(D_0 - D_x - D_y + D_{xy}), \tag{7.3}$$

$$\begin{aligned} D_3 = {} & D_{xyz} + \epsilon(D_{xy} - D_{xyz}) + \epsilon^2(D_{xz} - D_{xyz}) + \epsilon^3(D_x - D_{xy} - D_{xz} + D_{xyz}) \\ & + \epsilon^4(D_{yz} - D_{xyz}) + \epsilon^5(D_y - D_{xy} - D_{yz} + D_{xyz}) + \epsilon^6(D_z - D_{xy} - D_{yz} + D_{xyz}) + \\ & \epsilon^7(D_0 - D_x - D_y - D_z + D_{xy} + D_{yz} + D_{xz} - D_{xyz}). \end{aligned} \tag{7.4}$$

where $D_{U \subset \{x,y,z\}}$ *is the determinant of the matrix in which the* U*-coordinates are rounded.*

Proof. The proof is straighforward by entering the equations in MapleSoft [21]. □

Corollary 10. *Let* O_* *be the sign* $(0, +$ *or* $-)$ *of the determinant* D_*. O_* *is known as the orientation of the points that defines* M_*. *The orientations satisfy:*

$$O_1 = \begin{cases} & O_x \quad \text{if } O_x \neq 0, \\ \text{otherwise} & O_0. \end{cases} \tag{7.5}$$

$$O_2 = \begin{cases} & O_{xy} \quad \text{if } O_{xy} \neq 0, \\ \text{otherwise} & O_x \quad \text{if } O_x \neq 0, \\ \text{otherwise} & O_y \quad \text{if } O_y \neq 0, \\ \text{otherwise} & O_0. \end{cases} \tag{7.6}$$

$$O_3 = \begin{cases} & O_{xyz} & \text{if } O_{xyz} \neq 0, \\ \text{otherwise} & O_{xy} & \text{if } O_{xy} \neq 0, \\ \text{otherwise} & O_{xz} & \text{if } O_{xz} \neq 0, \\ \text{otherwise} & O_x & \text{if } O_x \neq 0, \\ \text{otherwise} & O_{yz} & \text{if } O_{yz} \neq 0, \\ \text{otherwise} & O_y & \text{if } O_y \neq 0, \\ \text{otherwise} & O_z & \text{if } O_z \neq 0, \\ \text{otherwise} & O_0. & \end{cases} \tag{7.7}$$

Corollary 11. *$\forall i \in \{1, 2, 3\}$ if the orientation O_i is null, then all orientations O_j for $j < i$ are null.*

7.5 Trimmed coordinate-by-coordinate collision test

The intersection test during the scheme trimmed coordinate-by-coordinate motion resembles that of the coordinate-by-coordinate scheme but utilizes the formulas of Corollary 10. Formulating this accurately is difficult because we aim to detect cases prompted by the presence of tagged vertices (see Chapter 9) and degenerate cases (see Fig. 7.4). The algorithm returns pairs of features, vertex/triangle or edge/edge, that properly intersect during the rounding motion; we say that these features "collide". We prove that two triangles do not properly intersect during the motion if all of their features do not collide (see Lemma 12).

For each pair of triangles: For every pair of features (edge/edge or vertex/triangle) from these two triangles, compute the orientations of the four vertices at each integer time of the motion O_0, O_1, O_2, O_3 using the formulas of Corollary 10. For each consecutive orientation O_i, O_{i+1} where $i \in 0, 1, 2$:

- If $O_{i+1} = 0$ (which implies that $O_i = 0$), return a collision if a vertex properly intersects an edge during this motion step.
- If $O_i \neq O_{i+1} \neq 0$, return a collision between these features if and only if they properly intersect[34] in projection onto the sidewall for $i = 0$, the backwall for $i = 1$, and the floor

[34]If a vertex properly intersects a triangle along an edge, the edges incident to that vertex also properly intersect that edge. Therefore, we can ignore this case when testing for vertex-triangle intersections and only check whether the vertex intersects the interior of the triangle.

for $i = 2$.

Finally, report a collision if the features are not forming a simplicial complex (except for a vertex on a face or a vertex on an edge, see Section 7.6) at the end of the motion ($i = 4$).

Lemma 12. *If two triangles properly intersect during the rounding motion, the collision detection returns a collision between a pair of features from these triangles.*

Proof. Let two triangles A and B that properly intersect during the trimmed coordinate-by-coordinate motion. Since the two triangles do not properly intersect at the beginning of the rounding, either an edge of A properly intersects an edge of B during the rounding, or a vertex of A properly intersects B during the rounding, or a vertex of B properly intersects A during the rounding.

The proof is relatively straightforward. Suppose that a feature of A properly intersects a feature of B during one of the first three motion steps.

Let O_i and O_{i+1} be the orientation of the four vertices of these two features, respectively before and after the motion step. Suppose that O_{i+1} is null, thus O_i is null (Corollary 11), and since the determinant is linear during a motion step, the orientation of these four vertices is null during the entire motion step. Therefore, the features properly intersect if and only if a vertex properly intersects an edge during the rounding.

Now suppose that O_{i+1} is not null. The four vertices are not aligned in the projection on the plane orthogonal to the motion. The features properly intersect if their orientation becomes null, and since the orientation is linear during a motion step, it hence occurs if $O_i \neq O_{i+1}$. Then, if the features properly intersect in 3D, they are equal or properly intersect on the floor. Since the four vertices are not aligned, they are not equal, concluding the proof. □

7.6 Proper intersections that can be post-processed

Our objective is to ensure that the resulting output forms a simplicial complex. However, some proper intersections that may arise can be resolved by subdividing the faces without introducing new vertices, thereby avoiding additional rounding. We can thus disregard intersections of the form of a vertex that lies on an edge or on a face at the end of the rounding process, as illustrated in Figure 7.5. Note that even though these proper intersections can be ignored at the end of the

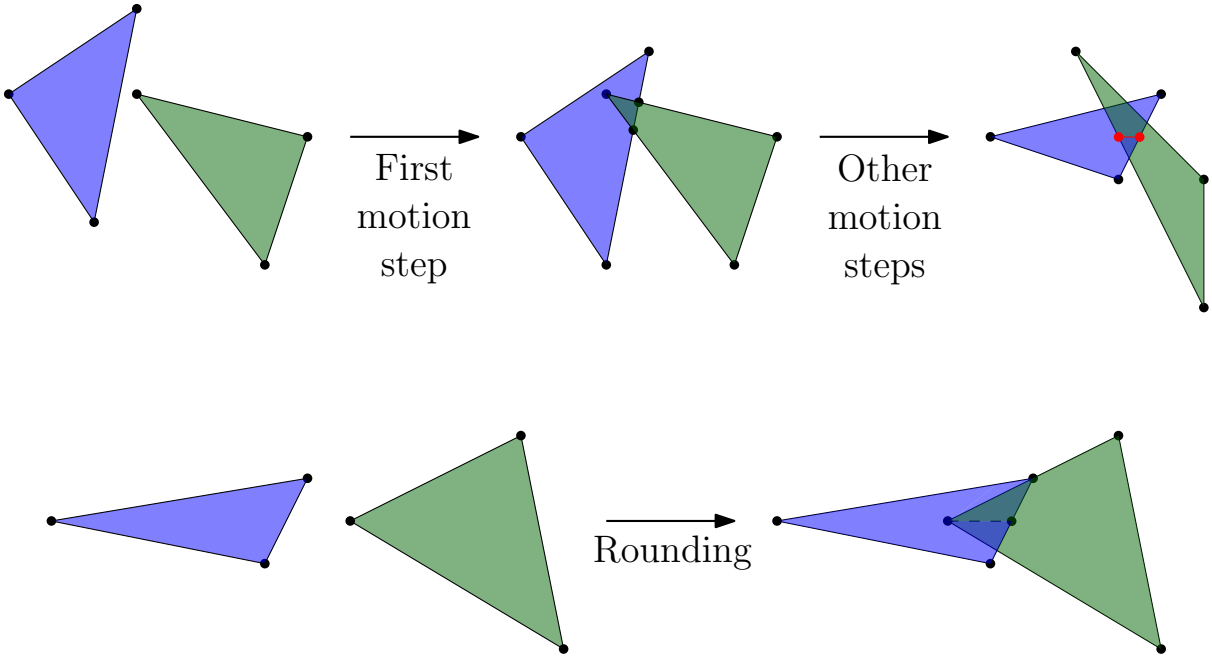


Figure 7.4: Example of degenerate proper intersections. At the top, the green triangle intersects a boundary of the blue triangle during the first step of the motion and intersects it transversally after the second step. At the bottom, the green triangle properly intersects the blue triangle during the rounding. This rounding does not involve the intersection of the interior of a pair of edge/edge.

rounding process, they cannot be ignored in the middle of the rounding if we wish to preserve the topology up to collapse (see Def. 1). Indeed, if a vertex traverses a face, it needs to be detected.

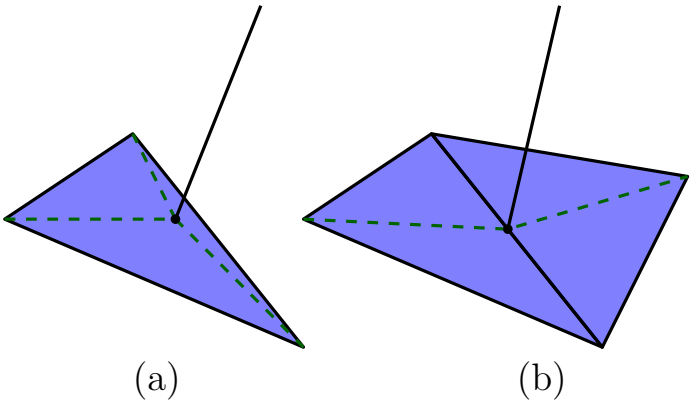


Figure 7.5: (a) A vertex in the interior of a face, (b) a vertex in the interior of an edge. These intersections can be computed and resolved without creating a new vertex.

Chapter 8

Optimizations

Experiments reveal that the described algorithm exhibits low running times but produces large output sizes (see Section 16.7). In this section, we introduce several optimizations aimed at reducing both the running time and the size of the output.

In the first section, we introduce local subdivision schemes that we believe to be of independent interest for 3D snap rounding. In Section 8.2, we explore the application of the PSL algorithm to handle coplanar cases. We discuss in Section 8.3 the Integration of these optimizations in the main algorithm. Finally, in the last section, we present a lazy variant of the triangulated 2D snap rounding algorithm.

8.1 Local subdivisions

Experimental results indicate that our algorithm can lead to excessive subdivisions (see Section 16.7). Although the algorithm guarantees a rounding without proper intersections, it often produces a large number of subdivisions. To mitigate this issue, we introduce a new local subdivision scheme.

Given a pair of features that properly intersect during a step of the rounding motion, this scheme subdivides them such that these features no longer properly intersect in that rounding motion. However, the newly subdivided edges or faces may intersect features that they did not originally intersect during the rounding. We have no proof that applying this scheme recursively terminates and therefore this does not constitute an algorithm on its own. Nonetheless, in practice, this approach improves significantly the efficiency of our algorithm by attempting to

resolve intersections using this scheme and adding the involved triangles to $\mathcal{F}_C$ if intersections persist after a fixed number of iterations.

We discuss the difficulties of subdividing schemes in the next section. In Section 8.1.2, we present our scheme for subdividing pairs of edges and, in Section 8.1.3, our scheme for subdividing triangles that are properly intersected by a vertex during the rounding.

8.1.1 Difficulty of subdividing

Designing an edge subdivision scheme such that the subdivided edges no longer properly intersect during the rounding motion is already a difficult task. As discussed in Section 3.3, recall that the two naive schemes for subdividing pairs of edges are flawed. These schemes subdivide pairs of edges either according to their intersection points in projection on the axis-aligned planes or at the points that realize their closest distances if less than one (see Fig. 3.3).

In the new scheme we present, we consider the coordinate-by-coordinate rounding motion and it should be stressed that the following intuitive scheme for pairs of edges in that context also fails. Namely, compute the time at which the two edges properly intersect, determine the intersection point, project this point back onto the original edges along the axes, and subdivide these edges accordingly. However, as illustrated in Fig. 8.1, the two subdivided edges can still properly intersect during the rounding motion. It is not clear whether repeating this process eventually eliminates all the intersections, but in any case, the number of iterations can be arbitrarely large, even for two edges, as example of Figure 8.1 can be repeated on the subdivided edges.

8.1.2 Edge/edge subdivision scheme

We consider the trimmed coordinate-by-coordinate rounding motion (see Section 7.4). Recall that this motion rounds the x-coordinates of all vertices in a first step, the y-coordinates in a second step and the z-coordinates in a third step.

Consider two edges that properly intersect during a given motion step. We describe a subdivision method that ensures that the subdivided edges no longer properly intersect during that motion step (see Fig. 8.2). The main idea behind the subdivision is that if we ensure that two edges do not properly intersect in their projection onto the orthogonal plane of the motion step (e.g., the yz-plane for the x-motion), they cannot properly intersect during that motion step.

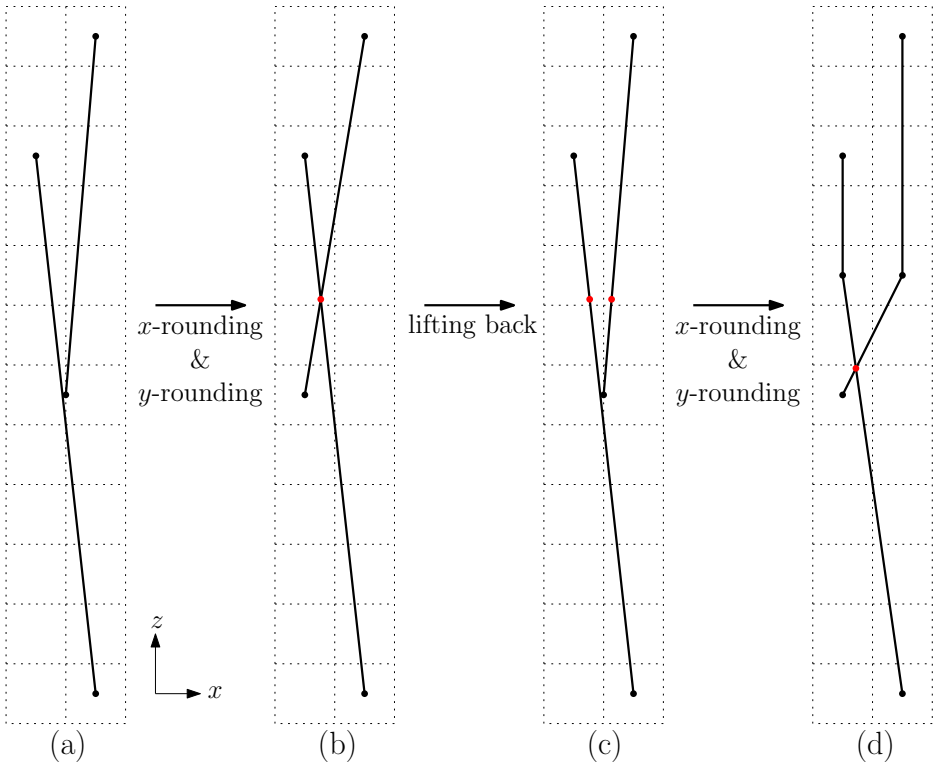


Figure 8.1: A flawed intuitive approach: (a) input edges, (b) intersection point of the edges after the rounding of the x-coordinates and then of the y-coordinates, (c) lifting of this point back onto the original edges along the axes. (d) The subdivided edges still properly intersect during the rounding motion.

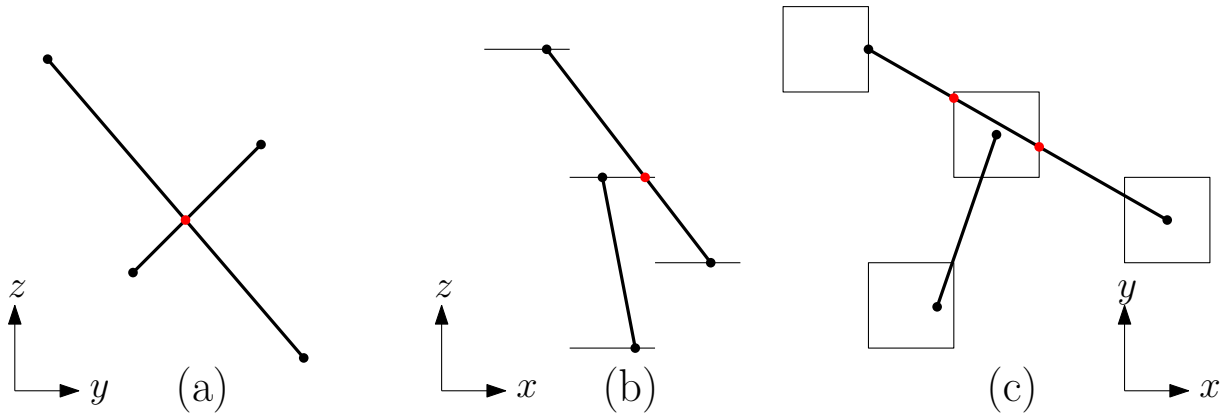
To ensure that, we project the input edges onto the corresponding orthogonal plane, subdivide them using techniques inspired by 2D Snap Rounding, and then lift the subdivisions back in 3D.

More formally, if two edges properly intersect during a motion step (see Fig. 8.2):

- **If they properly intersect during the x-motion:** Project the starting edges onto the sidewall (yz-plane) and subdivide them at their intersection point.[35] Then, lift the subdivisions back to the edges in 3D.

- **Otherwise, if they properly intersect during the y-motion:** Project the starting edges onto the backwall (xz-plane) and subdivide them at their intersection point (if one exists),[35] subdivide them by the 1D pixels along x that contain a vertex of the edges.[36] Then, lift the subdivisions back to the edges in 3D.

- **Otherwise, if they properly intersect during the z-motion:** Project the starting edges onto the floor (xy-plane) and subdivide them at their intersection point (if one exists),[35] subdivide them by the boundaries of the 2D pixels that contain a vertex of the



Subdivisions to prevent proper intersections during the x-motion.

Subdivisions to prevent proper intersections during the y-motion.

Subdivisions to prevent proper intersections during the z-motion.

Figure 8.2: The three types of subdivisions performed on edges to prevent proper intersections during the rounding motion: (a) To prevent proper intersections during the x-motion, edges are subdivided at the intersection point in the projection. (b) To prevent proper intersections during the y-motion, edges are further subdivided by the 1D pixel boundaries of their vertices. (c) To prevent proper intersections during the z-motion, edges are subdivided by the 2D pixel boundaries of their vertices.

edge.[36] Finally, lift the subdivisions back to the edges in 3D.

Lemma 13. *If two edges properly intersect during a motion step, then after the subdivision using the local subdivision scheme, the subdivided edges do not properly intersect during that motion step.*

Proof. Assume that the edges properly intersect during the x-motion step. Consider the projections of the edges onto the sidewall (yz-plane). After subdivision, the subedges do not properly intersect in the projection onto the sidewall (yz-plane), ensuring that they cannot properly intersect during the x-motion step.

Now, assume that the edges properly intersect during the y-motion step. At the beginning of the rounding motion, the subedges do not properly intersect in the projection onto the backwall (xz-plane). After performing the x-motion, the 1D pixel subdivisions ensure that the subedges still do not properly intersect in the projection onto the backwall (xz-plane), preventing a proper

[35] If the two projected edges are collinear but distinct, they are subdivided so that they do not properly intersect. If some input vertices are tagged, the new vertices have the relevant tags, see Chapter 9 for details.

[36] In the implementation, this subdivision is only performed if there is no intersection point between the projections of the edges. Consequently, two rounds of subdivision, rather than one, may be required to satisfy Lemma 13.

intersection during the y-motion step.

Finally, assume that the edges properly intersect during the z-motion step. Similarly as before, at the beginning of the rounding motion, the subedges do not properly intersect in the projection onto the floor (xy-plane). After performing the x- and y-motions, the 2D pixel subdivisions ensure that the subedges still do not properly intersect in the projection onto the floor (xy-plane), preventing a proper intersection during the z-motion step. □

8.1.3 Vertex/triangle subdivisions

As before, we consider the trimmed coordinate-by-coordinate rounding motion (see Section 7.4).

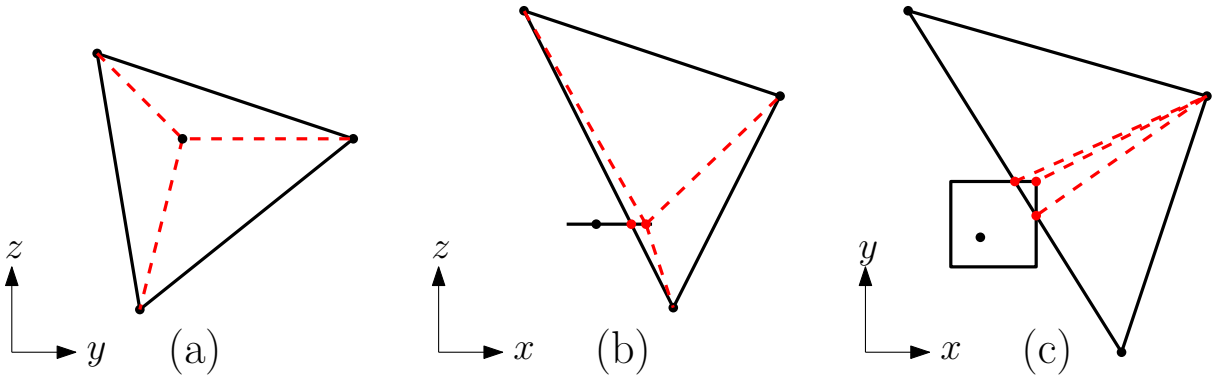
The vertex/triangle subdivision follows the same approach as the edge/edge subdivision (see Fig. 8.3). In the case where a vertex properly intersects a triangle on its boundaries, the intersection is already handled by the previous edge/edge subdivisions. Therefore, we only consider the scenario where a vertex intersects the interior of a triangle during the motion. We use tools from 2D snap rounding to ensure that the vertex does not lie inside the subdivided triangles in 2D during the motion. This scheme is performed at most three times per vertex/triangle pair, once for each motion step (see Lemma 15).

Consider a vertex that intersects the interior of a triangle during the rounding motion (see Fig. 8.3).

- **If the vertex intersects the interior of the triangle during the x-motion:** Consider the projection of the vertex and the triangle onto the sidewall (yz-plane). Subdivide the triangle in 2D at the point where the vertex is located (the projected vertex must be inside the projected triangle in this case).[37] Then, lift the subdivision back in 3D.

- **Otherwise, if the vertex intersects the interior of the triangle during the y-motion:** Project the features onto the backwall (xz-plane) and subdivide the triangle in 2D at the point where the vertex is located if it lies inside the triangle.[37] Additionally, subdivide the triangle by the 1D pixel along x that contains the vertex,[38] and lift the subdivision back in 3D.

- **Otherwise, if the vertex intersects the interior of the triangle during the z-motion:** Project the features onto the floor (xy-plane), and subdivide the triangle in 2D



Subdivisions to prevent proper intersections during the x-motion.

Subdivisions to prevent proper intersections during the y-motion.

Subdivisions to prevent proper intersections during the z-motion.

Figure 8.3: The three types of subdivisions performed on triangles to prevent proper intersections with a vertex during the rounding motion: (a) To prevent proper intersections during the x-motion, triangles are subdivided at the intersection point with the vertex in the projection. (b) To prevent proper intersections during the y-motion, triangles are further subdivided by the 1D pixel boundary of the vertex. (c) To prevent proper intersections during the z-motion, triangles are subdivided by the 2D pixel boundary of the vertex.

at the point where the vertex is located if it lies inside the triangle.[37] Further, subdivide the triangle by the boundaries of the 2D pixel that contains the vertex,[38] and lift the subdivision back in 3D.

Lemma 14. *If a vertex intersects the interior of a triangle during a motion step, after the triangle is subdivided by the local subdivision scheme, the vertex does not properly intersect the subtriangles during the same motion step.*

Proof. Suppose a vertex intersects the interior of a triangle during x-motion. Consider the vertex and a subtriangle after the subdivision in the projection on the sidewall (yz-plane). Due to the subdivision, the projection of the vertex does not lie within the interior of the projection of this subtriangle, ensuring that the vertex does not intersect the interior of the subtriangle during the x-motion step.

Now, suppose a vertex intersects the interior of a triangle during y-motion. Consider the vertex and a subtriangle after the subdivision in the projection on the backwall (xz-plane). The

[37] If the triangle is degenerate, vertex v properly intersects at least one edge of the triangle during the motion, and this case is already handled by edge/edge subdivisions. If vertex v is tagged, any new vertex resulting from subdivision inside the triangle inherits the same tags.

[38] In the implementation, this subdivision is performed only if the vertex is not inside the triangle. Therefore, two rounds of subdivision instead of one may be necessary to satisfy Lemma 14.

vertex projection does not lie within the interior of the projection of this subtriangle before the x-motion, and it does not cross an edge of the projected subtriangle during the x-motion, thanks to the 1D pixel subdivisions. Therefore, the vertex will still not be inside the projected subtriangle during the y-motion step, and thus it does not intersect the interior of the subtriangle during the y-motion step.

Finally, suppose a vertex intersects the interior of a triangle during z-motion. Simirlarly, consider the vertex and a subtriangle after the subdivision in the projection on the floor (xy-plane). The vertex projection does not lie within the interior of the projection of this subtriangle before the x- and y-motions, and it does not cross an edge of the projected subtriangle during the x- and y-motions due to the 2D pixel subdivisions. Consequently, the vertex will still not be inside the projected subtriangle during the z-motion step and thus does not intersect the interior of the subtriangle during the z-motion step. □

Lemma 15. *If a triangle is subdivided by the local subdivision scheme to avoid proper intersection with a vertex v during a motion step, then vertex v does not properly intersect the triangle during this motion step, even with further subdivisions.*

Proof. We go back to the proof of Lemma 14.

Suppose a vertex intersects the interior of a triangle during the x-motion. Recall that after the subdivision, the vertex no longer lies within the interior of the projection of any subtriangle. This remains true even if the subtriangles themselves are further subdivided, ensuring that the vertex does not intersect the interior of the subtriangles during the x-motion step.

Similarly, if a vertex intersects the interior of a triangle during the y-motion. After the subdivision, the 1D pixel containing the vertex does not lie within the interior of the projection of any subtriangle. As before, this remains valid even after additional subdivision of the subtriangles, preventing the vertex from intersecting their interiors during the y-motion step.

Finally, consider a vertex intersecting the interior of a triangle during the z-motion. After the subdivision, the boundary of the 2D pixel containing the vertex no longer lie within the interior of the projection of any subtriangle. Once again, this holds even if the subtriangles are further subdivided, ensuring the vertex does not intersect the interior of the subtriangles during the z-motion step. □

Corollary 16. *The local subdivision scheme for one vertex and one input triangle can be applied*

at most three times.

8.2 Coplanar subdivision scheme

The local subdivision scheme significantly reduces the number of subdivisions performed in the main part of our algorithm, but it becomes ineffective when triangles become coplanar at the end of the rounding motion (see Fig. 8.4). Faces that become coplanar in one of the three orthogonal planes present a specific case that can be well handled by a PSL algorithm, which works in this case although it is flawed in the general case as discussed in Devillers et al. [8] and in Section 3.4.

Recall that the PSL algorithm (P) projects all the input faces onto the floor (xy-plane), (S) subdivides them as described in 2D Triangulated Snap Rounding, and (L) lifts back this triangulation along the z-direction onto all faces, subdividing them accordingly.[39] We restate Lemma 7 on the PSL algorithm.

Lemma 7. *After the lifting operation of the PSL algorithm, the projection on the floor of any two triangles do not properly intersect at any stage of the rounding.*

Using this lemma, we can easily prove that faces that become parallel to the floor during the rounding do not properly intersect (Lemma 17). By permuting the coordinates, the PSL algorithm is ideal for managing faces that become coplanar with any axis-aligned plane during the rounding. Additionally, we apply it to subdivide pairs of faces that become coplanar on non-axis-aligned planes, though there is no formal proof that it entirely prevents intersections in such cases.

Lemma 17. *Let A and B be two input triangles that are parallel to the xy-plane after the rounding. The subdivisions of A and B by the PSL algorithm will not properly intersect during the rounding.*

Proof. Consider a pair of subedges after the subdivision by the PSL algorithm. Due to the subdivision (Lemma 7), they do not properly intersect on the floor during the rounding motion. Thus, if the edges are not, or do not become, equal on the floor during the rounding motion, they do not properly intersect.

[39] We can use the lazy variant from Section 8.4.

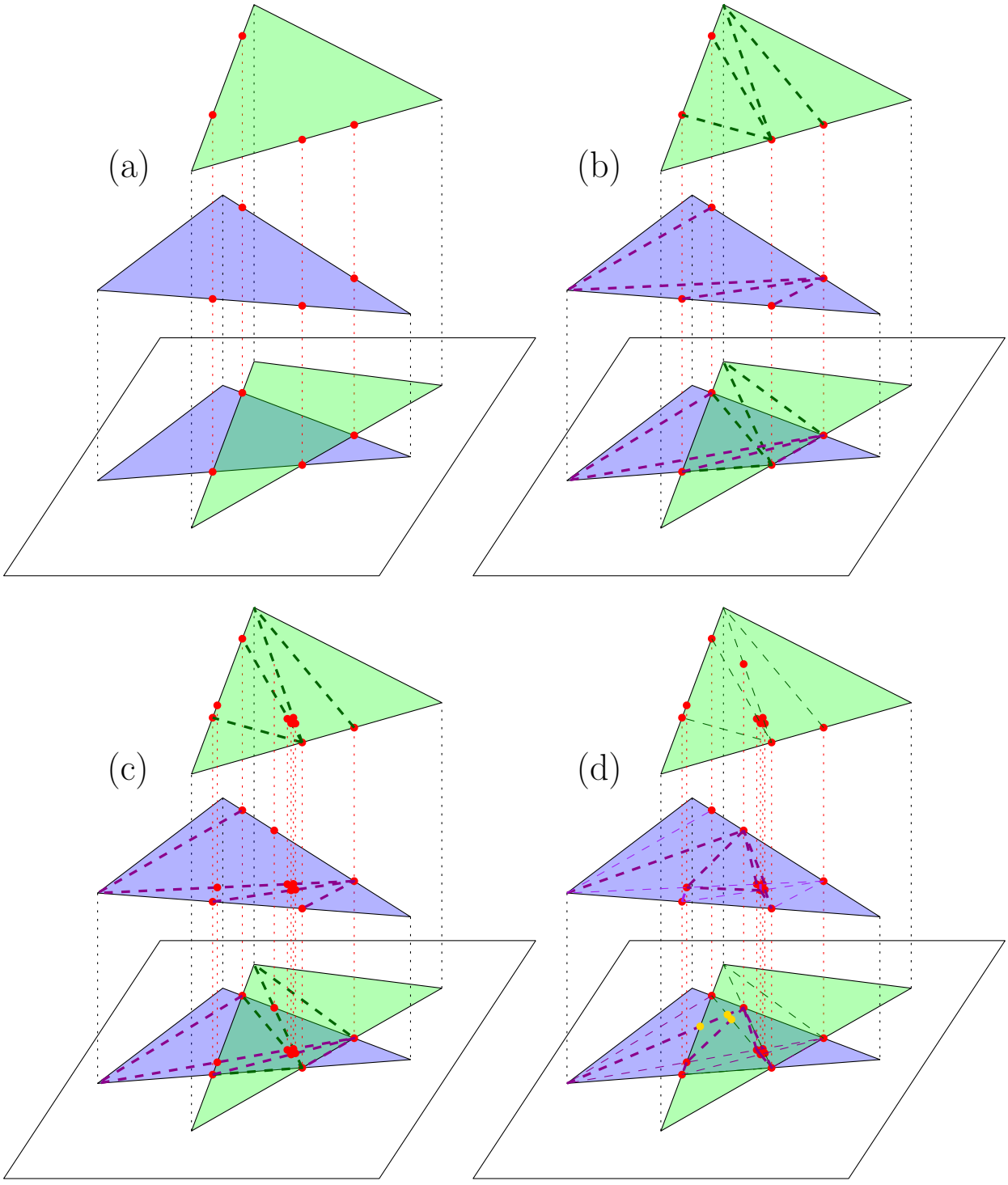


Figure 8.4: Pathological example for edge/edge subdivision: two faces that become coplanar during the rounding motion and where the local subdivision scheme fails to subdivide them efficiently to prevent intersections. (a) The edges of two faces are subdivided by the local subdivision scheme. (b) The faces are triangulated before simulating the rounding. (c) New edges are subdivided by the local subdivision scheme. (d) The faces are triangulated again, but intersections still occur during the simulation of the rounding.

If the edges are equal on the floor at the beginning of the rounding motion, by assumption, they do not properly intersect at the beginning of the rounding, and since they remain equal on the floor, they do not properly intersect during the rounding.

If the edges become equal on the floor during the rounding motion, due to the trimmed coordinate-by-coordinate motion (see Section 7.4), they do not properly intersect during the first three steps of the rounding motion (by Corollary 11) and thus do not properly intersect during these steps. During the last step of the motion, the two edges become coplanar to the floor by assumption, and since they are equal on the floor, they are equal in 3D.

The proof is similar for a vertex and a subtriangle after the subdivision by the PSL algorithm. Due to the subdivision (Lemma 7), they do not properly intersect on the floor during the rounding motion. Thus, if the triangle is not degenerated to a segment on the floor and the projection of the vertex is not on this segment, they do not properly intersect.

If they satisfy these properties at the beginning of the rounding motion, by assumption, they do not properly intersect at the beginning of the rounding, and since they remain equal on the floor, they do not properly intersect during the rounding.

If they satisfy these properties during the rounding motion but not at the beginning, due to the trimmed coordinate-by-coordinate motion (see Section 7.4), they do not properly intersect during the first three steps of the rounding motion (by Corollary 11) and thus do not properly intersect during these steps. During the last step of the motion, the triangle and the vertex become coplanar to the floor by assumption, and since they do not properly intersect on the floor, they do not in 3D. □

8.3 Integration in the main algorithm

We have designed two optimization schemes for our algorithm, which we need to integrate into the main structure. Recall that the main structure of the algorithm is as follows: while proper intersections occur during the simulation of the rounding, add the faces involved in proper intersections to the set of critical faces $\mathcal{F}_C$. If $\mathcal{F}_C$ increases, reset the model and perform Step 1 and Step 2 on $\mathcal{F}_C$. Otherwise, perform Step 3 to Step 5 on $\mathcal{F}_C$.

We propose a new structure that integrates these optimizations, in which we do not immediately add the faces involved in proper intersections to $\mathcal{F}_C$.

First, we integrate the coplanar subdivision scheme into the algorithm. We use this as a preprocessing step, where we first compute the rounded support planes for all input faces and cluster them based on these rounded planes. We then apply the PSL algorithm to clusters associated with one axis-aligned planes. The output from this step serves as input for the subsequent stages of the algorithm.

Even though Lemma 17 applies only when faces become coplanar with axis-aligned planes, we can also use the PSL algorithm, as a heuristic, on clusters associated with non-axis-aligned planes to help prevent intersections.

Then, we integrate the local subdivision scheme. The algorithm attempts to resolve proper intersections using this scheme and a face is added to $\mathcal{F}_C$ only if the number of subdivisions performed on it exceeds a fixed constant. Note that when an intersection occurs between a face G in $\mathcal{G}_C$ and one not in $\mathcal{G}_C$, we apply the local subdivision scheme. However, to preserve the floor invariant (see Section 6), we also subdivide all faces in $\mathcal{G}_C$ that share the same projection on the floor as G.

8.4 Lazy 2D triangulated snap rounding

The 2D Triangulated Snap Rounding algorithm takes a set of 2D segments as input and tags as "hot" each pixel that contains a vertex of the arrangement. It then computes the triangulated arrangement of the input and adds the boundaries of all these hot pixels. This process significantly increases the number of vertices in the arrangement, and consequently the number of triangles in the output, by approximately eleven times.[40] Although most of the triangles created will degenerate during the rounding process (and thus will not appear in the final rounded output), they contribute to the computational load during the simulation of the rounding. To address this issue, we propose a lazy variant of the 2D Triangulated Snap Rounding algorithm.

Roughly speaking, this lazy algorithm first computes the triangulated arrangement and only adds the boundaries of a "hot" pixel if it prevents an intersection during the motion.

Formally, given a set of 2D triangles as input, the algorithm computes the 2D arrangement of these triangles and triangulates it. The 2D rounding is then simulated using the trimmed

[40] If the input is a triangulation, each vertex has, on average, six incident edges. For each vertex, the algorithm adds the four corners of the pixel containing it, as well as the intersections of its six adjacent edges with the pixel boundaries.

coordinate-by-coordinate motion (see Section 7.4). For each vertex that properly intersects a segment during this motion, the boundaries of the pixel containing this vertex are added to the arrangement. This process continues as long as proper intersections occur during the simulation.

In the worst case, the boundaries of all hot pixels are added, producing an output identical to that of the original triangulated snap rounding. However, in most cases, only a few pixel boundaries are introduced over a limited number of iterations, resulting in a significant acceleration of the algorithm. Note that the rounded outputs of both the lazy and non-lazy versions of the 2D snap rounding may be not identical. Furthermore, even when they are identical, the detection of proper intersections can differ during the rounding simulation, potentially changing the final output of the main algorithm.

Chapter 9

Vertices on voxel boundaries

In our algorithm, some operations generate vertices with half-integer coordinates. These coordinates are usually rounded down, although some are rounded up. We describe in this chapter how such vertices, referred to as "tagged vertices," are managed.

In the first section, we present the tagged vertices and the issues they present. In the second section, we introduce the symbolic perturbation considered for these vertices and its impact on the algorithm.

9.1 Tagged vertices

During the algorithm, new vertices are created by subdividing edges or faces along the boundaries of narrow slabs or 2D pixels (typically lifted in 3D afterward). These points are positioned on voxel boundaries with coordinates having decimal values of 0.5, which are rounded down by default. To ensure the proofs hold, it is essential that vertices created by these subdivisions remain within the pixel or slab from which they originated (see Fig. 9.1). To achieve this, the coordinates of the vertices are tagged with boolean values. When a coordinate has a decimal value of 0.5, and the corresponding boolean is true, the value is rounded up instead of down. This method allows for up to eight geometrically identical vertices to be rounded into different voxels (see Fig. 9.2). These vertices are referred to as "tagged vertices," and the features containing them are called "tagged segments," "tagged triangles," etc. Using tagged vertices results in features that are geometrically degenerate at the beginning of the rounding.

Tagged vertices are not a new concept. In the 3D snap rounding algorithm by Devillers et

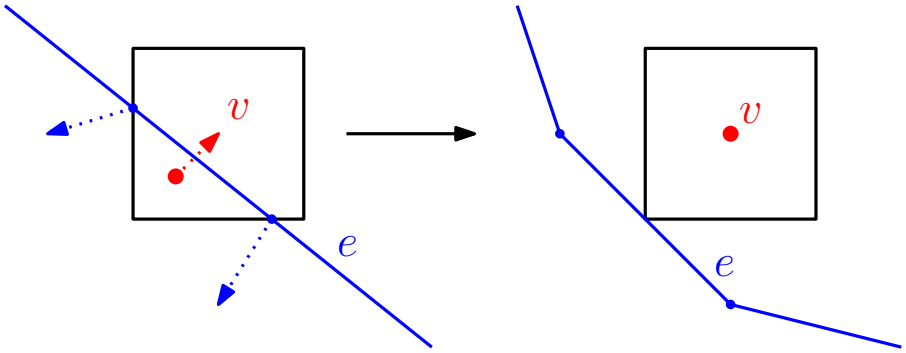


Figure 9.1: On the left, an edge e is subdivided by the boundary of a 2D pixel that contains a vertex v. On the right, their rounded counterparts if we were using the naive rounding, where coordinates are rounded down for half-integer values. The vertex would cross a subedge of e during the rounding motion.

al. [8], tagged vertices existed for slab boundaries. However, they were limited to these boundaries, and only a few operations involved them. In algorithms for 2D snap rounding (see Chapter 2), the issue is addressed by either directly outputting the rounded version or by subdividing edges at any point inside the pixel, rather than at boundaries. Even though such subdivisions make proper intersections possible during the rounding motion (see Fig. 9.3), the resulting output is identical to that of subdivisions at pixel boundaries, so the tagged vertices were a minor theoretical detail, neither developed in papers nor implemented in practice.

With the simulation of the rounding becoming a crucial aspect of our algorithm, it is now necessary to perform subdivisions at pixel/voxel boundaries, as intersections can occur during simulation if subdivisions are made at arbitrary points within a voxel, as shown in Figure 9.3. Thus, we now manage tagged vertices directly, which is a serious burden in our algorithm.

- Through the subdivision of an edge by the walls of a narrow slab during Step 3 (see Section 6.4).
- Through the subdivision of an edge or face by a 2D pixel during the triangulated 2D snap rounding (see Section 2.6). The triangulated 2D snap rounding is used by the PSL algorithm, which itself is used by Step 4 and in the coplanar optimization (see Sections 6.5 and 8.2).
- Through the subdivision of an edge or face by a 1D or 2D pixel during the local optimization (see Section 8.1).

Once created, tagged vertices may be involved in further operations such as intersection tests

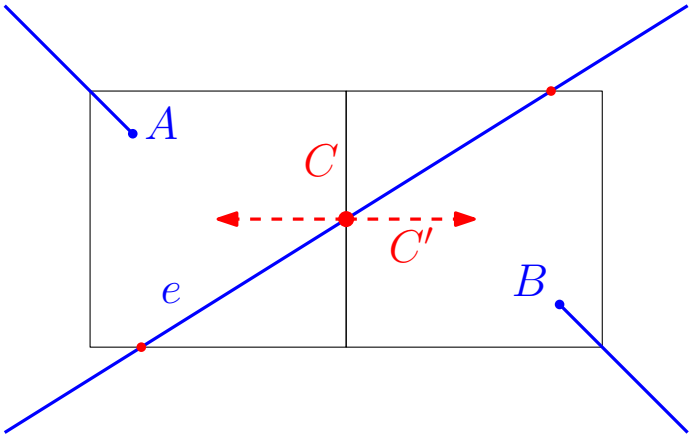


Figure 9.2: An example of triangulated 2D snap rounding (see Section 2.6), where vertices A and B define two adjacent "hot" pixels. The subdivision of the edge e by these two hot pixels results in two vertices, C and C', with equal coordinates but rounded into different voxels.

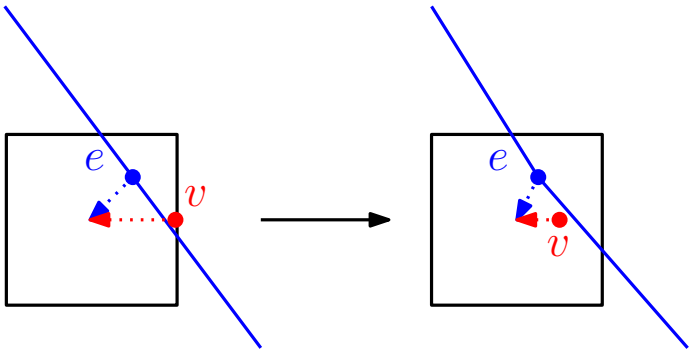


Figure 9.3: On the left, an edge e is subdivided inside a 2D pixel that contains a vertex v. On the right, their counterparts after half of the first step of the coordinate-by-coordinate motion. The vertex crosses a subedge of e during the first step of the rounding motion.

during the rounding motion, subdivision of an edge by a slab, or during the local optimization, etc. Additionally, the lazy nature of our algorithm complicates managing tagged vertices. When tagged vertices are created by the subdivision of an edge or face by a slab or pixel, it cannot be assumed that other edges or faces are similarly subdivided by the same slab or pixel.

9.2 Shifting tagged vertices

Many existing implementations, for instance for computing triangulations, do not accept input vertices with identical geometric coordinates. Our goal is to redefine the predicates so that geometric operations can be performed using tagged vertices.

One initial idea is to assume that each tagged vertex is perturbed by a symbolic distance ϵ in the direction of its tags. However, this symbolically modifies the geometry of the features and

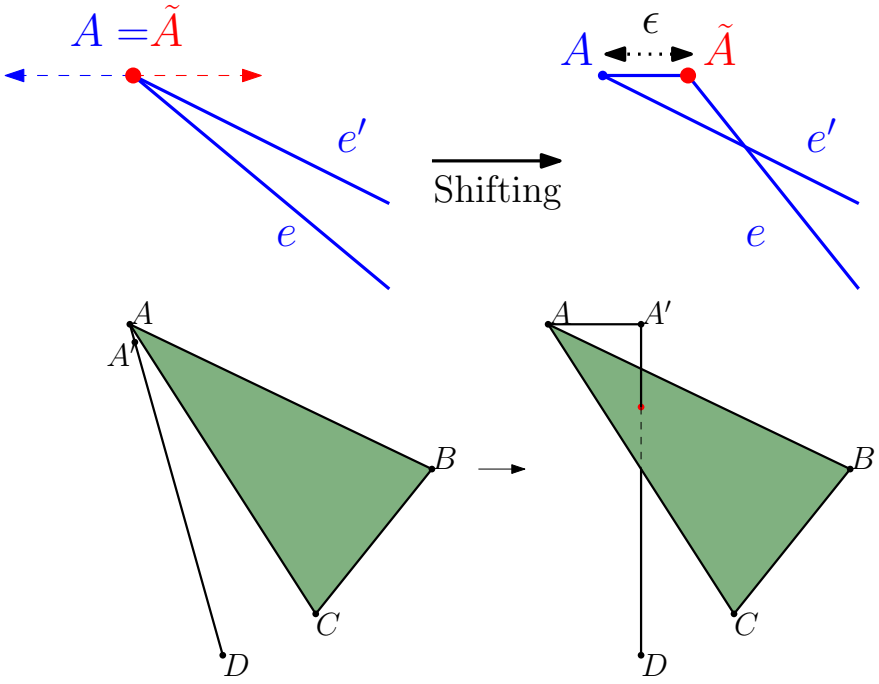


Figure 9.4: A and A' are geometrically identical but have different tags. If we apply perturbation in the direction of the tags, a proper intersection is created between e and e' at the top, and between ABC and $A'D$ at the bottom.

may lead to the creation of proper intersections before the rounding motion (see Fig. 9.4). We assume that the model is intersection-free before the rounding and thus we do not consider such shifts.[41]

To prevent the creation of proper intersections, we aim to build a shifting of the tagged vertices such that the geometry of the features that support them remains unchanged, even symbolically. When tagged vertices are created by the subdivision of an edge, we assume that they are perturbed by a symbolic distance along the line supporting that edge. If tagged vertices are created by subdivision in the middle of a face by a pixel corner in 2D projection, we assume that the vertex is perturbed symbolically toward the point of the face (or its supporting plane), whose 2D projection is the center of that pixel.

This shifting introduces some unusual side effects. For example, two vertices with equal coordinates and tags may not be equal if they are supported by two different edges (see Fig. 9.5). Two segments with endpoints that have equal coordinates may proper intersect when considering the shift, or, conversely, two segments that properly intersect without the shift may no longer do

[41] This problem also arises if we consider symbolic perturbation in the direction of the vertex's rounding value.

so (see Fig. 9.7). The subdivision of an edge by a narrow slab, which typically creates vertices with a tag for the x-coordinate, may also generate tags on the y- and z-coordinates (see Fig. 9.6). While these effects are not inherently problematic, understanding all the implications is a burden on the development of the algorithm and the implementation.

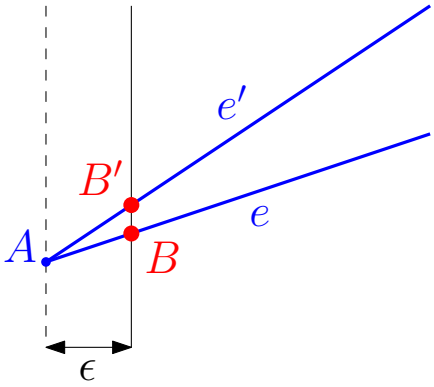


Figure 9.5: A vertex A with a half-integer x-coordinate is incident to edges e and e'. Both edges are subdivided by the wall of a narrow slab at points B and B'. Although vertices B and B' have identical coordinates and tags, they are distinct due to the symbolic perturbation along the edges.

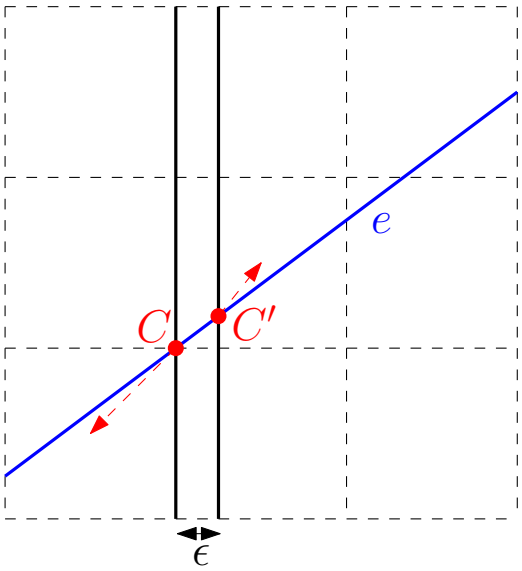


Figure 9.6: An edge e is subdivided by two adjacent narrow slabs at C and C'. e crosses the narrow slab at a half-integer y-coordinate, so the resulting subdivision vertices round to different y-coordinates.

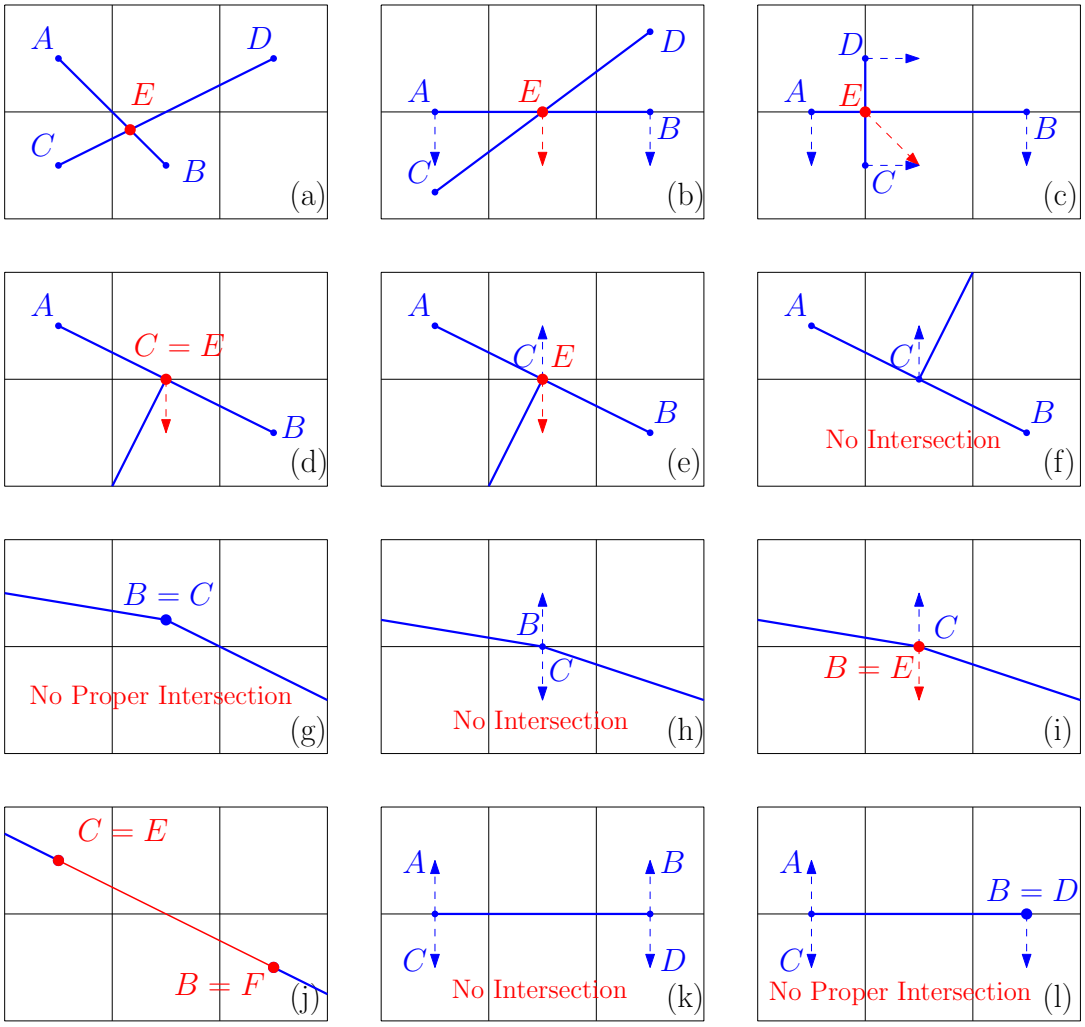


Figure 9.7: Several intersection cases between segments with tagged vertices. The input segments $[AB]$ and $[CD]$ are blue. The proper intersection point E or intersection segment $[EF]$ are shown in red, if they exist. Dashed arrows point to the rounded vertices. (a)-(c) depict intersections along the two edges. (d)-(f) depict cases where an endpoint is geometrically on the other edge. (g)-(i) depict cases of two segments with geometrically identical endpoints. (j)-(l) depict cases of two geometrically collinear segments.

Chapter 10

Proof of correctness and termination

We prove in this chapter that our algorithm, defined in Chapter 6, is correct and terminates.

In the following, we refer to a feature of a (triangulated) face to as one of its vertices, edges or triangles of its triangulation. We say that two (triangulated) faces properly intersect during the rounding if two features of their triangulations properly intersect during the rounding.

Theorem 18. *Given a set $\mathcal{F}$ of polygonal faces in 3D that do not properly intersect, the algorithm terminates and outputs a simplicial complex $\mathcal{F}'$ whose vertices have integer coordinates and a mapping σ that maps every face F of $\mathcal{F}$ onto a set of faces (or edges or vertices) of $\mathcal{F}'$ such that there exists a continuous motion that moves every face F into $\sigma(F)$ such that (i) the L_∞ Hausdorff distance between F and its image during the motion never exceeds $\frac{3}{2}$ and (ii) if two points on two faces become equal during the motion, they remain equal through the rest of the motion.*

The proof of Theorem 18 is organized as follows. We first prove in Lemma 19 that if the algorithm terminates, it outputs a simplicial complex. We then prove in Lemma 20 that Step 1 has the desired properties for Theorem 18. If the algorithm terminates, Lemmas 19 and 20 directly yield Theorem 18 because, after the last iteration of Step 1 in the algorithm's while loop, faces are only subdivided and triangulated (in Steps 2 to 5) and their vertices are then rounded (see the proof of Theorem 18 at the end of the section for details). We prove the termination of the algorithm in Lemma 25. We do so by first proving in Lemmas 23 and 24 that a *new* narrow face or vertex is created every times Subdivision rules (i) and (ii) are applied; we then bound the number of narrow faces and vertices in the proof of Lemma 25.

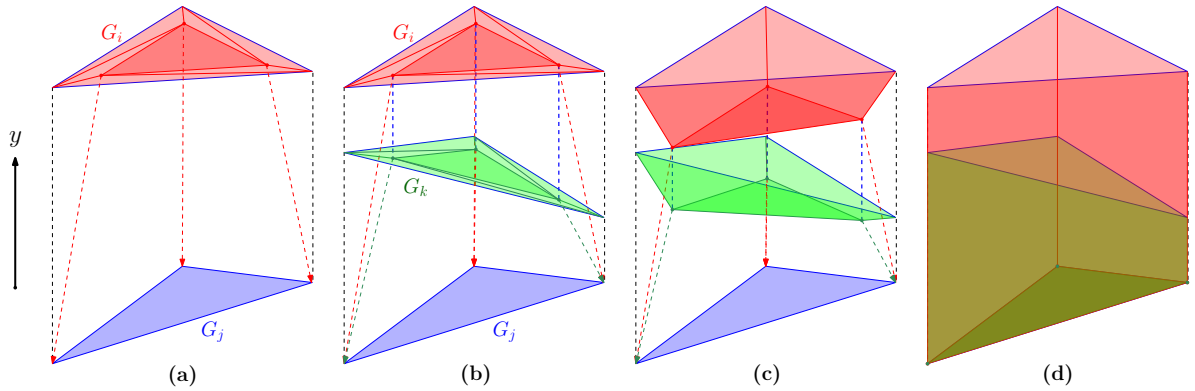


Figure 10.1: For the proof of Lemma 20: Motion for the Step 1 projection (some edges of the triangulations are missing for readability).

Lemma 19. *If the algorithm terminates, the triangulated faces at the end of the while loop (Chapter 6) do not properly intersect during the rounding (Chapter 7).*

Proof. By the Invariant on $\mathcal{G}$, the faces of $\mathcal{G}$ do not properly intersect before the rounding. If two vertices become equal during the rounding (of a coordinate), they remain equal until the end of the rounding (of all coordinates). Hence, if two features properly intersect during the rounding, then a vertex properly intersects a face or two edges properly intersect. By the Termination criterion, such intersections cannot occur when the while loop terminates, which concludes the proof. □

We prove in the following lemma that the projection defined in Step 1 has the desired properties for Theorem 18. This lemma is very similar to one by Devillers et al. [8, Lemma 2] but our proof is remarkably simpler; the main reason is that, in the projection of Step 1, we always define connecting walls between the boundary of the face we project from and the boundary of the face we project to, while in the similar projection in Devillers et al., connecting walls are not defined when they are incident to boundaries on the input faces.

Lemma 20. *Given a set $\mathcal{F}$ of polygonal faces that do not properly intersect, Step 1 outputs a set of faces $\mathcal{G}$ such that there exists a continuous motion that maps every face $F \in \mathcal{F}$ onto a set of faces in $\mathcal{G}$ such that (i) the L_∞ Hausdorff distance between F and its image during the motion never exceeds 1 and (ii) no two distinct points on two faces become equal during the motion except possibly at the end of the motion.*

Proof. Let $\mathcal{G}_j$ be the set of faces in $\mathcal{G}_C$ that project on G_j in Step 1. The faces in $\mathcal{G} \setminus \cup_j \mathcal{G}_j$ are invariant in Step 1 and we define the motion that projects a face $G_i \in \mathcal{G}_j$ onto G_j as follows (see Figure 10.1). Recall that G_i and G_j have the same (non-degenerate) projection on the back wall and that their y-distance is less than 1. For every $G_i \in \mathcal{G}_j$, we consider a triangulation of G_i that contains a shrunk copy of G_i by a constant factor and so that any two faces in $\mathcal{G}_j$ have the same triangulation on the back wall. Each vertex of the shrunk copy of G_i moves at constant speed to its corresponding vertex in G_j. All the other points of G_i move according to their barycentric coordinates in the triangulation. The boundary of G_i remains invariant and, except at the end of the motion, the relative interior of G_i remains strictly inside the convex hull of G_i and G_j, which is a section of the cylinder with basis G_j and axis the y-axis. This property implies the lemma in a rather straightforward way, as shown below.

The definition of the motion trivially implies the first property (i) of the lemma. Furthermore, during the motion, the triangulations of all the faces in $\mathcal{G}_j$ remain equal in projection on the back wall and none of their projected triangles degenerate to a segment except at the end of the motion. Since no two vertices become equal during the motion except possibly at the end, property (ii) holds for pairs of faces in $\mathcal{G}_j^+ = \mathcal{G}_j \cup G_j$. Furthermore, a face $G \in \mathcal{G}$ that intersects the interior of $CH(\mathcal{G}_j^+)$ is necessarily in $\mathcal{G}_j^+$. Indeed, G is in $\mathcal{G}_C$ because otherwise, Step 1 would have been restarted with $\mathcal{F}_C$ augmented with the face of $\mathcal{F}$ that supports G; thus G is either projected on (or equal to) G_j, that is G is in $\mathcal{G}_j^+$, or it is projected onto some other face G_k with $k < j$ (because of the lexicographical order in which the projections are performed). But in the latter case, there is a face G_u in $\mathcal{G}_j^+$ whose y-distance to G_k is less that 1, and since $k < j$, G_u should have been projected onto G_k instead of G_j. □

Lemmas 21 and 22 are straightforward properties that are useful for the proofs of Lemmas 23 and 24. In Lemma 21, the narrow vertices (see Subdivision rule (i)) are considered as degenerated narrow faces.

Lemma 21. *Two narrow faces do not properly intersect during the simulation of rounding in the Termination criterion (Section 6.1).*

Proof. In each narrow slab, the narrow faces are subdivided (in Step 4) so that their projections on the side wall do not properly intersect. Thus, when rounding all coordinates in x, no proper

intersections occur. Then, when rounding the coordinates in y, then in z, the vertices and edges do not properly intersect by property of the 2D snap rounding [15, Thm. 1].[42] □

Lemma 22. *Let T be a triangle in a (closed) narrow slab $\mathcal{N}$ and T' be a triangle whose relative interior is in the (open) complement of $\mathcal{N}$. A point of T and a point of $T' \setminus \mathcal{N}$ cannot coincide during the rounding of their vertices.*

Proof. The x-coordinates of any two vertices in $\mathcal{N}$ and in its (open) complement, $\mathcal{W}$, can never coincide during the rounding. This extends to $p \in T$ and $p' \in T'$ as long as p' belongs to the open set $\mathcal{W}$, since p and p' are affine combinations of the vertices of their triangles. □

We prove in the following Lemmas 23 and 24 that a new narrow face or vertex is created every times Subdivision rules (i) and (ii) are applied, which is instrumental for the proof of termination in Lemma 25. In these lemmas, a face is called a *new* narrow face if it is not geometrically and combinatorially equal to a narrow face that already exists. The index of the original face is considered in the combinatorial information.

Lemma 23. *In Subdivision rule (i), at least one new narrow face is created (or labelled) or a non-narrow vertex is labelled narrow. Furthermore, if no new narrow face is created (or labelled), the new narrow vertex is an input vertex or one created in Steps 1 to 3.*

Proof. Let v be a vertex that properly intersects a (triangulated) face G in $\mathcal{G}_C$ during the simulation of rounding when Subdivision rule (i) is applied. Let $\mathcal{N}$ be the narrow slab that contains v. Assume for a contradiction that v is already labelled as a narrow vertex and that $G \cap \mathcal{N}$ already exists as a narrow face. The latter implies that G lies in $\mathcal{N}$ or that $G \cap \mathcal{N}$ is reduced to one of its edges or vertices (or a straight polyline). It follows that the triangulation of G can be decomposed into the part $G \cap \mathcal{N}$ that lies in $\mathcal{N}$ and a set of triangles whose relative interiors lie in the (open) complement of $\mathcal{N}$. The narrow face $G \cap \mathcal{N}$ cannot properly intersect the narrow vertex v during the rounding, by Lemma 21, and the same holds for the other triangles by Lemma 22, which is a contradiction and proves the first claim of the lemma.

We now prove the second claim of the lemma. No vertices are created in Step 5. If v is a vertex created in Step 4, then v is a vertex of (the triangulation of) a narrow face. Thus, if $G \cap \mathcal{N}$

[42] Note that the proof of Lemma 21 is very similar to one by Devillers et al. [8, Lemma 5] but we cannot directly use their result because, although we perform the same subdivisions of the narrow faces in each narrow slab, we use a different rounding scheme.

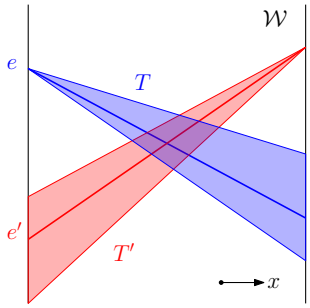


Figure 10.2: For the proof of Lemma 24.

is already defined as a narrow face, then, as above by Lemmas 21 and 22, v cannot properly intersect G during the rounding when Subdivision rule (i) is applied. This is a contradiction, which concludes the proof. □

Lemma 24. *In Subdivision rule (ii), at least one new narrow face is created (or labelled).*

Proof. Let e and e' be two edges that properly intersect during the simulation of rounding when Subdivision rule (ii) is applied, and let G and G' be two faces in $\mathcal{G}_C$ that support e and e' respectively. Let $\mathcal{N}_i$, $i \in \{1, \ldots, 4\}$, be the narrow slabs that contain the endpoints of e and e'. Assume for a contradiction that the intersections of G and G' with $\mathcal{N}_i$, $i \in \{1, \ldots, 4\}$, are already defined as narrow faces (in $\mathcal{G}$). As in the proof of Lemma 23, this implies that the triangulations of G and G' can be decomposed into parts that lie in $\mathcal{N}_i$, $i \in \{1, \ldots, 4\}$, and a set of triangles whose relative interiors lie in the (open) complement of $\cup_i \mathcal{N}_i$.

If e is contained in a narrow slab $\mathcal{N}_i$, then e and e' do not properly intersect during the rounding, since the same holds for $G \cap \mathcal{N}_i$ and $G' \cap \mathcal{N}_i$ by Lemma 21, and for $G \cap \mathcal{N}_i$ and $G' \setminus \mathcal{N}_i$ by Lemma 22.

We can thus assume that e does not lie in the narrow slabs $\mathcal{N}_i$, $i \in \{1, \ldots, 4\}$, and thus that it spans a (wide) slab $\mathcal{W}$ that is a connected component of the complement of $\cup_i \mathcal{N}_i$. Similarly, we can assume that e' spans a slab $\mathcal{W}'$ and that $\mathcal{W} = \mathcal{W}'$ since, otherwise, e and e' cannot intersect during the rounding except possibly at a common endpoint.

We first observe that e and e' properly intersect in projection on the floor or in projection on the back wall. Indeed, otherwise, they would not properly intersect during the rounding since they span $\mathcal{W}$.

Consider the case where e and e' do not properly intersect in projection on the floor. They

thus properly intersect in projection on the back wall. They furthermore intersect transversally in projection on the back wall since they span $\mathcal{W}$ (see Figure 10.2) and their y-distance is less than 1 since otherwise they would not properly intersect in 3D during the simulation of rounding. If neither G nor G' is parallel to the y-axis, their projections on the back wall intersect in their interiors, contradicting the Back-wall invariant on $\mathcal{G}_C$. Thus, say, G is parallel to the y-axis, which yields a contradiction since G' should have been subdivided in Step 1 so that their back-wall projections would not properly intersect.

Thus e and e' properly intersect in projection on the floor. They furthermore intersect transversally in projection on the floor since they span $\mathcal{W}$ (see Figure 10.2). Let $\mathcal{T}$ be the triangulation of G without considering its junction vertices and junction-induced edges, and let T be a triangle of $\mathcal{T}$ that contains e.[43] Define similarly triangle T' that contains e' in the triangulation $\mathcal{T}'$ of G'. As argued at the beginning of the proof, T and T' are contained in $\mathcal{W}$ (see Figure 10.2).

If neither T nor T' is parallel to the z-axis, their projections on the floor intersect in their interiors and are distinct (see Figure 10.2), which contradicts the Step-5 property (T and T' are at distance at most one since e and e' intersect in 3D during the simulation of rounding). Thus, say, T (and thus G) is parallel to the z-axis, which yields a contradiction since G' should have been subdivided in Step 2 so that their floor projections would not properly intersect. □

Lemma 25. *The algorithm terminates in polynomial time.*

Proof. At each iteration of the while loop, Subdivision rules (i) or (ii) are applied (in Step 3), hence, by Lemmas 23 and 24, a new narrow face or narrow vertex is created (or labelled).

Let α be the number of faces obtained after the subdivisions of Steps 1 and 2 and let β be the number of narrow slabs. The number of narrow faces is trivially in $O(\alpha\beta)$. The number of vertices labelled narrow in Subdivision rule (i) when no new narrow face is created (or labelled) is also in $O(\alpha\beta)$. Indeed, by Lemma 23, such vertices are either input vertices or vertices created in Steps 1-3. There are $O(\alpha\beta)$ such vertices since those created in Step 3 are intersection points between the $O(\alpha)$ edges at the end of Step 2 and the β narrow slabs.[44]

Both α and β are at most polynomial in the input size: This is straightforward for α. For β,

[43] Note that if e is a junction-induced edge in the triangulation of G, e is strictly inside T (except for its endpoints).

[44] Note that it is critical here that junction vertices are not lifted in Step 3, Subdivision rule (iv).

observe that each narrow slab contains (at least) a vertex of $\mathcal{G}_C$ (by Subdivision rules (i) and (ii)). The number of input vertices and those defined in Steps 1 and 2 is in $O(\alpha)$ and the vertices created in Steps 3 and 4 are defined in narrow slabs that already exist.

Finally, each iteration of the loop is clearly also polynomial in the input size. □

Proof of Theorem 18. Lemma 25 proves that the algorithm terminates and Lemma 19 that it outputs a simplicial complex.

It remains to prove that there exists a continuous motion such that (i) the Hausdorff distance between every face and its image never exceeds $\frac{3}{2}$ during the motion and (ii) if two points on two faces become equal during the motion, they remain equal through the rest of the motion. Lemma 20 implies these properties (for the motion defined in the proof of Lemma 20 for the projection of Step 1 and in Chapter 7 for the final rounding). Indeed, in every non-trivial iteration of Step 1 in the algorithm's while loop, we reinitialize $\mathcal{G}$ to $\mathcal{F}$, so in other words, we discard all previous subdivisions and modifications performed on the faces. Furthermore, after the last non-trivial iteration of Step 1, faces are only subdivided and triangulated (in Steps 2 to 5) and their vertices are then rounded (Chapter 7). Hence, the Hausdorff distance is at most 1 during Step 1 (by Lemma 20) and at most $\frac{1}{2}$ during the final rounding. In addition, Property (ii) holds for the Step 1 motion by Lemma 20 and it also trivially holds for the final rounding motion. □

Chapter 11

Worst-case complexity

In this chapter, we study the worst-case complexity of the algorithm presented in Chapter 6. In the first section, we provide an upper bound of the complexity in the worst case, while we provide a lower bound in the second section.

11.1 Upper-bound complexity

Theorem 26. *Given a set of polygons of total complexity $O(n)$, the algorithm outputs a simplicial complex of complexity $O(n^{12})$, in time $O(n^{45})$ and space $O(n^{17})$.*

Before proving Theorem 26, we bound the number of loop iterations. This proof follows that of Lemma 25 but with a more careful complexity analysis.

Lemma 27. *The algorithm terminates with at most $O(n^{11})$ loop iterations, where n is the number of features (vertices, edges, faces) of the input.*

Proof. At each iteration of the while loop (Chapter 6), either a new input face is added to $\mathcal{F}_C$ when evaluating the Termination criterion or Subdivision rules (i) or (ii) are applied (in Step 3).[45]

The number of times a new face can be added to $\mathcal{F}_C$ is at most the number of input faces, that is $O(n)$. By Lemmas 23 and 24, every time Subdivision rule (i) or (ii) is applied, a new narrow face is created (or labelled) or a new vertex is labelled narrow.

The number of narrow faces is at most the number of faces obtained after the subdivisions of Step 1 and Step 2 times the number of narrow slabs.[46] It is also bounded by the number of faces

[45] If the optimization (on the output) mentioned in Footnote 26 is not applied, Subdivision rules (i) or (ii) are necessarily applied at each iteration.

[46] Actually three times the number of narrow slabs according to Footnote 29.

(after Step 2) plus the number of intersection points between the lines supporting the edges of these faces and the boundary walls of narrow slabs.

We first study the complexity of the subdivided faces after the subdivisions of Step 1 and Step 2. Consider in the back wall the arrangement $\mathcal{A}$ of the $O(n^2)$ lines that support the projections of the $O(n)$ input edges and the $O(n^2)$ intersections between an input face and another one translated by $\pm\vec{j}$. At the end of Step 1, in the worst case, each of the $O(n)$ input faces is subdivided by the projection (on the face) of these $O(n^2)$ lines, which define $N_3 = O(n^3)$ lines in total. In addition, the connecting walls are also bounded by the $N_4 = O(n^4)$ lines that are parallel to the y-axis and incident in the back wall to the $O(n^4)$ vertices of $\mathcal{A}$. In Step 2, in the worst case, we lift the above $N_3 = O(n^3)$ lines and $N_4 = O(n^4)$ y-parallel lines onto the $O(n)$ planes supporting the input faces, which defines $M_5 = O(n^5)$ lines that are parallel to the yz-plane and $M_4 = O(n^4)$ other lines. The complexity of the subdivided faces at the end of Step 2 is thus $O((N_3^2 + N_3N_4) \cdot n) = O(n^8)$ and there are at most $M_4 = O(n^4)$ supporting lines that are not parallel to the yz-plane.

Furthermore, there are at most $O(n^6)$ narrow slabs. Indeed, each narrow slab contains (at least) a vertex of $\mathcal{G}_C$ (by Subdivision rules (i) and (ii)). There a four kinds of vertices: the $O(n)$ input vertices, the $O(n^8)$ vertices defined in Steps 1 and 2, the subdivision vertices (defined in Step 3) and vertices induced by the 2D-snap simulation in the narrow slabs (in Step 4). The latter two are defined in narrow slabs that already exist and the $O(n^8)$ vertices at the end of Step 2 only define $O(n^6)$ narrow slabs because their x-coordinates are those of the vertices of the arrangement of the above $N_3 = O(n^3)$ lines projected on the floor plus those of the above $N_4 = O(n^4)$ y-parallel lines.

The number of narrow faces is thus at most $O(n^8 + n^4 \cdot n^6) = O(n^{10})$. We now count the number of vertices labelled narrow in Subdivision rule (i) when no new narrow face is created (or labelled). By Lemma 23, such vertices are either input vertices or vertices created in Steps 1 to 3. As mentioned above, there are at most $O(n^8)$ vertices created in Steps 1 and 2. The number of (subdivision) vertices created in Step 3 (in Subdivision rules (i) to (iv)) is at most the $M_4 = O(n^4)$ above lines times the $O(n^6)$ narrow slab boundary walls, which is again $O(n^{10})$.

Since $\mathcal{G}_C$ is reinitialized in Step 1 every time a new face is added to $\mathcal{F}_C$, the total number of loop iterations is at most n times $O(n^{10})$, which concludes the proof. □

Proof of Theorem 26. First, we consider one iteration of the while loop. We have shown in the proof of Lemma 27 that the complexity of $\mathcal{G}$ at the end of Step 2 is $O(n^8)$ and that the edges are supported by $M_5 = O(n^5)$ lines that are parallel to the yz-plane and $M_4 = O(n^4)$ other lines.

In Step 3, the subdivision of the faces by the boundary walls of the narrow slabs defines edges that are supported by $O(n)$ lines in each narrow slab (since there are $O(n)$ input faces). We furthermore define $O(n^4)$ subdivision vertices per slab defined as the intersection of the above $M_4 = O(n^4)$ lines and the slab boundary. When defining the hot pixels in the narrow slabs (in Step 4), we consider non-triangulated faces, thus the hot pixels are defined by the intersections induced (after projection on the side wall) by the above $M_4 + M_5 = O(n^5)$ lines for all slabs and the $O(n)$ lines per slab. Among the M_5 lines parallel to the yz-plane, let m_i denote the number of lines that lie in the i-th slab. The number of hot pixels is thus $O(N_5^2)$ in total plus $O(M_4 \cdot n + m_i \cdot n + n^2)$ for the i-th slab. As argued in the proof of Lemma 27, there are $O(n^6)$ narrow slabs, thus the total number of hot pixels is in $O(N_5^2 + M_4 \cdot n \cdot n^6 + n \cdot \sum m_i + n^2 \cdot n^6) = O(n^{11})$.

Each hot pixel belongs to a narrow slab and it can thus define a junction vertice on each of the above $O(n)$ Step-3 lines in the slab as well as each of the above $M_5 = O(n^5)$ Step-2 lines. The $O(n^{11})$ hot pixels may thus define $O(n^{16})$ junction vertices in total. The complexity of the arrangement on *each* of the $O(n)$ input faces is thus $O(n^{16})$ before the rounding and $O(n^{11})$ after the rounding.

The size of the output is thus in $O(n^{12})$. Furthermore, all the arrangements and triangulations performed by the algorithm can be done in time and space complexities that match their worse sizes. Thus, the time and space complexity of one iteration of the loop, not counting the Termination criterion, is in $O(n^{17})$.

The complexity of evaluating the Termination criterion in each loop iteration is $O(n^{34})$ since the number of intersections that have to be tested is the square of the number of features (vertices, edges, triangles) at the end of the previous loop iteration. This concludes the proof since the number of loop iterations is $O(n^{11})$ by Lemma 27. □

11.2 Lower-bound complexity

It is unclear whether the worst-case complexity of $O(n^{12})$ subdivisions can be achieved. However, we present an example that proves the algorithm induces in the worst case at least $\Omega(n^4)$

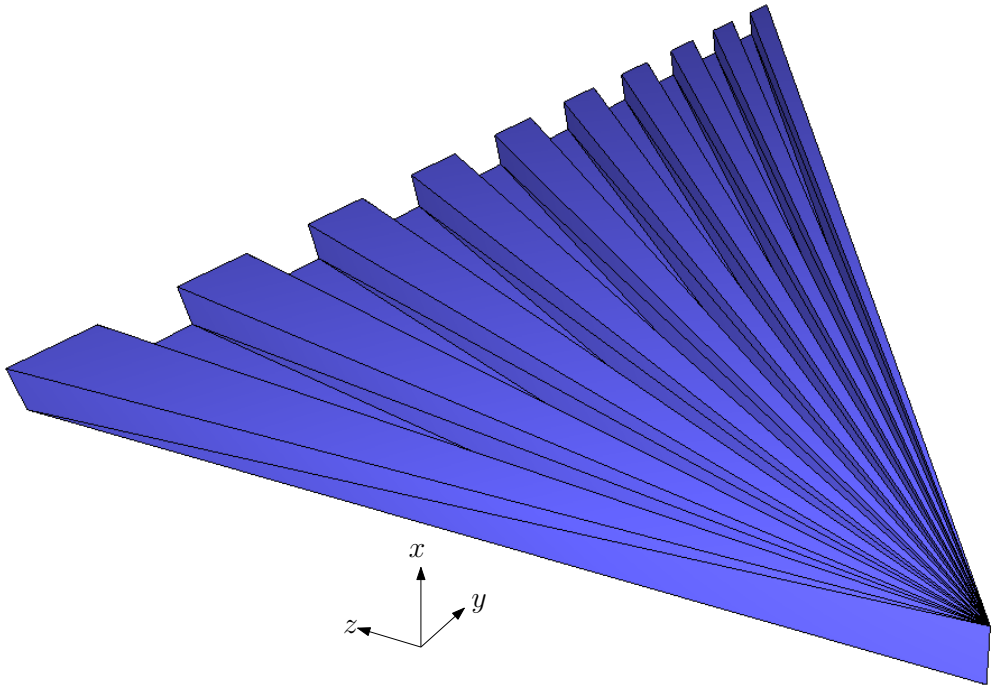


Figure 11.1: The ribbed fan in 3D.

subdivisions in the absence of optimizations, establishing Theorem 28.

Theorem 28. *Given a set of polygons of total complexity $\Theta(n)$, in worst-case, the algorithm outputs a simplicial complex of complexity $\Omega(n^4)$.*

In Section 16.7.2, we explore experiments using a "fan", a pathological example inspired by structures observed in Thingi10K. The example presented in this chapter is an adaptation of this fan, with faces orthogonal to the axis to enhance the readability of the proof.

This example, a ribbed fan (see Fig. 11.1), consists of n rectangular faces orthogonal to the sidewall, all sharing a common edge. To form a connecting mesh, additional walls parallel to the sidewall are introduced. However, as these walls are degenerate in projection on the backwall and the floor, they do not affect the first two steps of the algorithm. For clarity, we omit them in the following.

Formally, the i^{th} rectangle[47] is formed by the common edge with endpoints $v = (0, 0, 0)$ and $v' = (1, 0, 0)$, while the opposite edge has endpoints $v_i = (0, 2^i - 1, M)$ and $v'_i = (1, 2^i - 1, M)$ with $1 \ll M$. These faces are degenerate in lines on the sidewall (see Fig. 11.2).

We add a triangle such that, when rounded, it intersects the corner of the fan, causing all

[47]In the algorithm, the rectangle would be triangulated, but for simplicity, we ignore the diagonals here.

fan's faces to be added to $\mathcal{F}_C$. For simplicity, this triangle is ignored in the following. We then execute our algorithm on the ribbed fan. See Figures 11.2 to 11.11 for a visualization of the algorithm's execution.

We now apply the first step of the algorithm (see Section 6.2). To recap, during Step 1, we project all the faces in $\mathcal{F}_C$ onto the backwall, and for each pair of faces, we add the projection of the lines at a y-distance one between them before lifting the triangulated arrangement on the backwall onto the 3D edges. Subsequently, for all pairs of faces in $\mathcal{G}_C$ in lexicographic order, we project the first face onto the other if they are at a y-distance at most one.

In our ribbed fan, all faces have identical projections onto the backwall, and since they share a common edge, they are all at y-distance less than one (See Fig. 11.3). We will show in Lemma 31 that $\Omega(n^2)$ lines are projected onto the backwall. The faces in $\mathcal{G}_C$ are then projected onto one another. We assume that the faces in $\mathcal{G}_C$ are indexed such that subfaces of face i are projected onto those of face j for $i < j$ (see Fig. 11.4 to 11.9). By Lemma 32, $\Omega(n^2)$ projections are made, resulting in $\Omega(n^3)$ lines in 3D after these projections.

Next, we apply the second step of the algorithm. During Step 2, we project all the faces in $\mathcal{G}_C$ onto the floor (see Fig. 11.10), compute the triangulated arrangement on the floor, and then lift this arrangement back into 3D (see Fig. 11.11). The lines in 3D are lifted onto almost all faces (Lemma 33), resulting in $\Omega(n^4)$ lines in 3D after the first two steps of the algorithm. The faces are at a y-distance at least one according to the Back-wall invariant on $\mathcal{G}_C$, and M can be chosen large enough such that the rounded z-coordinates of all lines are distinct, thus establishing Theorem 28.

We analyzed only the first half of the algorithm here (Steps 1 and 2); even though the rest of the algorithm should increase the lower bound complexity, its analysis is difficult and we did not develop this example further.

We first establish two technical Lemmas 29 and 30. We postpone their proofs to the end of the section. Then, we establish and prove all the lemmas mentioned above.

Lemma 29. *Let $i, i', j, j' \in \mathbb{N}$ with $(i, j) \neq (i', j')$. Then:*

$$\frac{1}{|2^i - 2^j|} \neq \frac{1}{|2^{i'} - 2^{j'}|}.$$

Lemma 30. *Let $i, i', j, j', k, k' \in \mathbb{N}$ with $(i, j, k) \neq (i', j', k')$ and $k < j < i$ and $k' < j' < i'$.*

Then:

$$\frac{2^i - 1}{|2^j - 2^k|} \neq \frac{2^{i'} - 1}{|2^{j'} - 2^{k'}|}.$$

Lemma 31. *For the ribbed fan, $\Omega(n^2)$ distinct lines are added to the arrangement on the backwall during Step 1 of the algorithm.*

Proof. Consider faces i and j where $i \neq j$. The line at y-distance between face i and j are parallel to the x-axis at coordinate $z_{ij} = \frac{M}{|2^i - 2^j|}$ (see Fig. 11.3). By Lemma 29, all coordinates z_{ij} are distinct. Thus, the $\binom{n}{2}$ lines in the projection on the backwall are distinct, completing the proof. □

Lemma 32. *Face i of the ribbed fan is projected onto $n - i - 1$ other faces and $\Omega(i^2)$ lines are lifted onto it during Step 1 of the algorithm.*

Proof. Continuing from Lemma 31, for each pair (i, j), the line $z = \frac{M}{|2^i - 2^j|}$ appears in the projection onto the backwall.

Now consider face i. The inequality $\frac{1}{|2^j - 2^i|} > \frac{1}{|2^{j'} - 2^i|}$ holds if $j' > j > i$. Hence, when face i is projected onto face n, a portion of face i remains at a y-distance at most one from face $n - 1$, and it is subsequently projected onto this face. The same holds for $n - 2, n - 3, \ldots, i + 1$.

The last projection of face i is done at $z = \frac{M}{|2^{i+1} - 2^i|} = \frac{M}{2^i}$. Then, $\frac{1}{2^i} < \frac{1}{|2^j - 2^k|}$ for all $k, j < i$. Thus, the line defined by faces j and k is lifted onto face i, concluding the proof. □

Lemma 33. *$\Omega(i^3)$ lines are lifted onto face i of the ribbed fan during Step 2 of the algorithm.*

Proof. Continuing from Lemma 32, consider face i. For all $j, k \leqslant i$ where $j \neq k$, the line at y-distance one between face j and face k is lifted onto face i. The z-coordinate of this lifting line is $\frac{M}{|2^j - 2^k|}$, and then the y-coordinate of the line is $y_{ijk} = \frac{2^i - 1}{|2^j - 2^k|}$. On the projection onto the floor, the y-coordinate of the unprojected portion of face i ranges from $\frac{2^i - 1}{|2^{i+1} - 2^i|} < 1$ to $2^i - 1$. By Lemma 30, all coordinates $y_{i'jk} < 2^i$ are distinct for $k < j < i' \leqslant i$. Therefore, $\binom{i}{3}$ distinct lines are lifted onto face i during Step 2, concluding the proof. □

Proof of Lemma 29. We prove this by contraposition. Suppose:

$$\frac{1}{|2^i - 2^j|} = \frac{1}{|2^{i'} - 2^{j'}|}. \tag{11.1}$$

Assuming $j < i$ and $j' < i'$, this implies:

$$2^i - 2^j = 2^{i'} - 2^{j'}. \tag{11.2}$$

Assume without loss of generality that j is the smallest and divide by 2^j, we get:

$$2^{i-j} - 1 = 2^{i'-j} - 2^{j'-j}. \tag{11.3}$$

Since the left-hand side is odd, the right-hand side must also be odd, implying $2^{j'-j} = 1$. Thus, $j = j'$. It follows that $i = i'$, concluding the proof. □

Proof of Lemma 30. We prove this by contraposition. Suppose:

$$\frac{2^i - 1}{|2^j - 2^k|} = \frac{2^{i'} - 1}{|2^{j'} - 2^{k'}|}. \tag{11.4}$$

This implies:

$$(2^i - 1)(2^{j'} - 2^{k'}) = (2^{i'} - 1)(2^j - 2^k). \tag{11.5}$$

Assume without loss of generality that $k \leqslant k'$. Dividing both sides by 2^k, we get:

$$(2^i - 1)(2^{j'-k} - 2^{k'-k}) = (2^{i'} - 1)(2^{j-k} - 1). \tag{11.6}$$

Since the right-hand side is odd, the left-hand side must also be odd, implying $2^{k'-k} = 1$. Thus, $k = k'$. We now have:

$$(2^i - 1)(2^{J'} - 1) = (2^{i'} - 1)(2^J - 1). \tag{11.7}$$

where $J = j - k$ and $J' = j' - k$. Expanding both sides:

$$2^{iJ'} - 2^i - 2^{J'} = 2^{i'J} - 2^{i'} - 2^J. \tag{11.8}$$

Assume without loss of generality that $J \leqslant J'$, divide by 2^J:

$$2^{iJ'-J} - 2^{i-J} - 2^{J'-J} = 2^{i'J-J} - 2^{i'-J} - 1. \tag{11.9}$$

Since the right-hand side is odd, the left-hand side must also be odd, implying $J = J'$, and hence

$j = j'$. It follows that $i = i'$, concluding the proof. $\square$

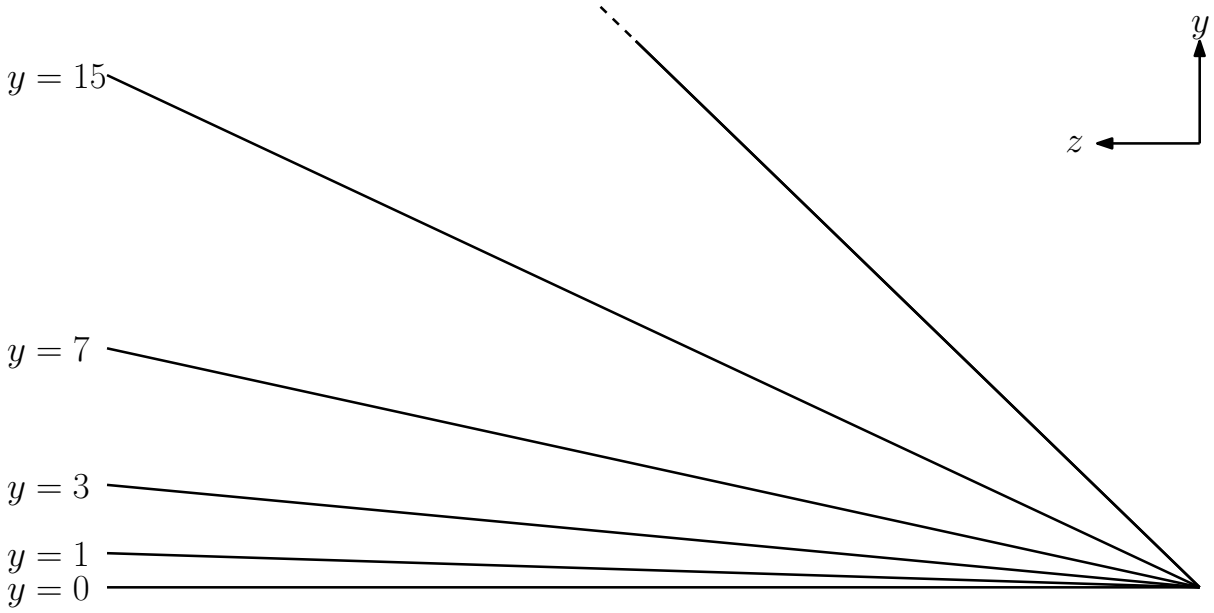


Figure 11.2: The ribbed fan in the projection on the sidewall. The faces are degenerated into edges. Segments and lines parallel to the x-axis are degenerated into vertices.

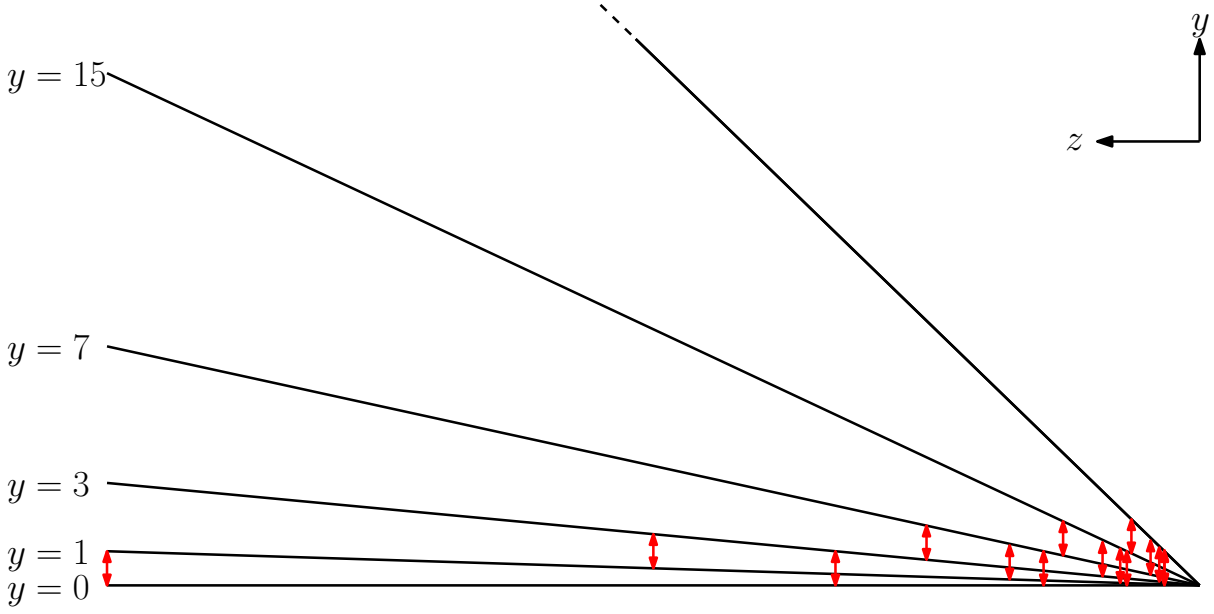


Figure 11.3: The ribbed fan in the projection on the sidewall. Double arrows have length one and show the points at y-distance one between two edges.

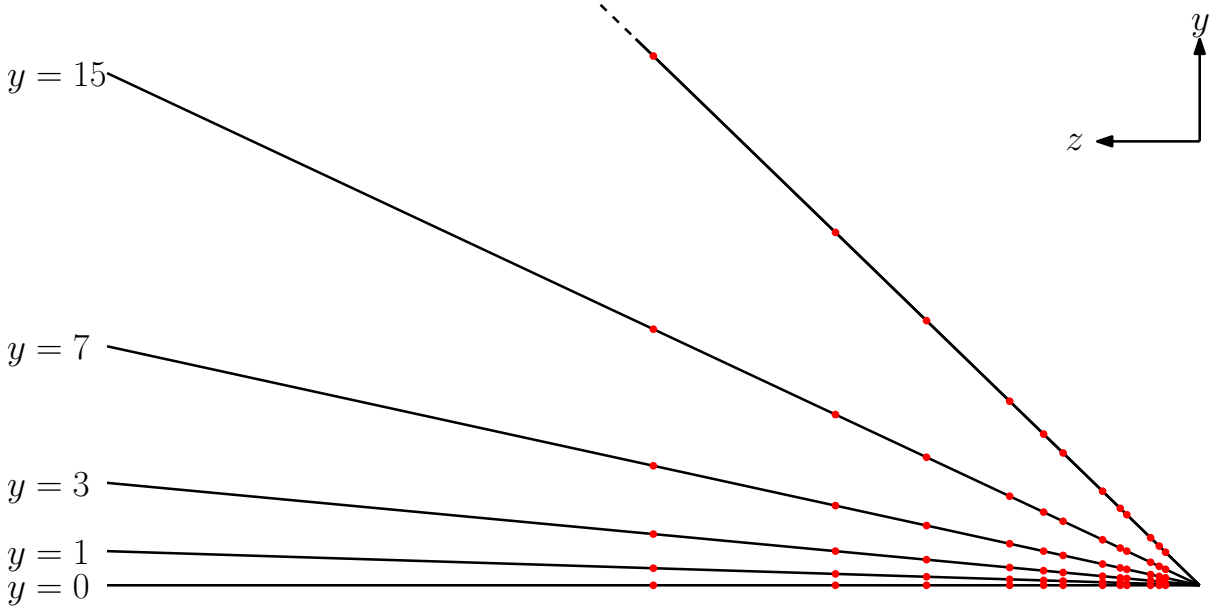


Figure 11.4: The ribbed fan after lifting of the lines at y-distance one along y-axis. The red vertices are the resulting new vertices of $\mathcal{G}_C$.

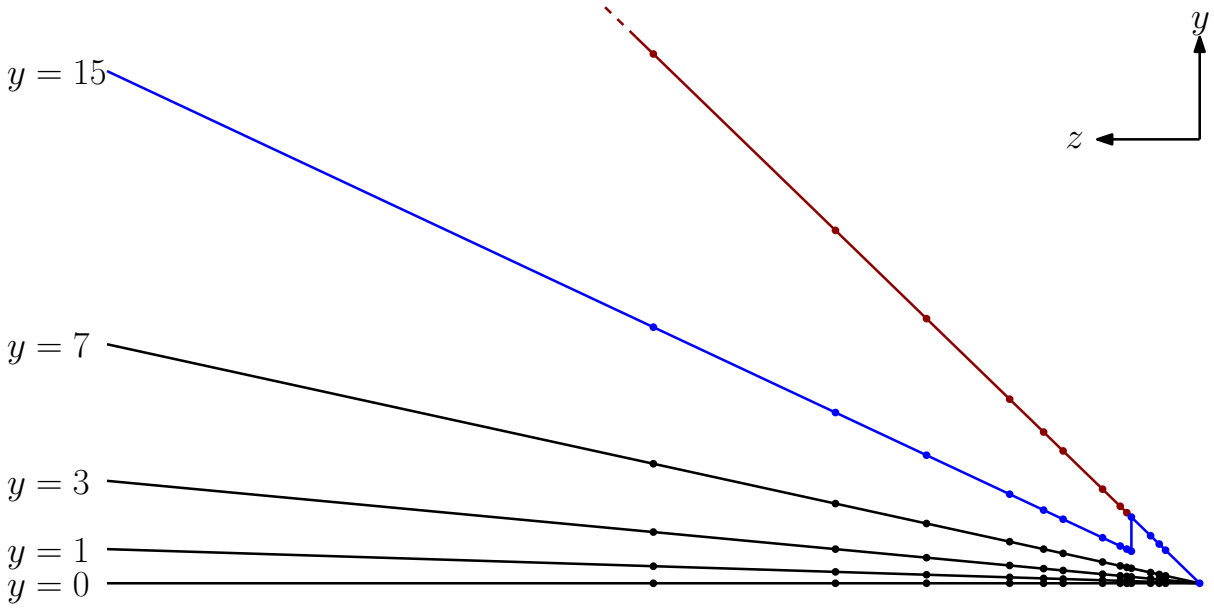


Figure 11.5: Projection of the second face of the ribbed fan during Step 1 of the algorithm. The edges (in blue) of the second face are projected on the edges of the first face above if they are at y-distance at most one. Connecting walls are added to maintain the connectivity of the face.

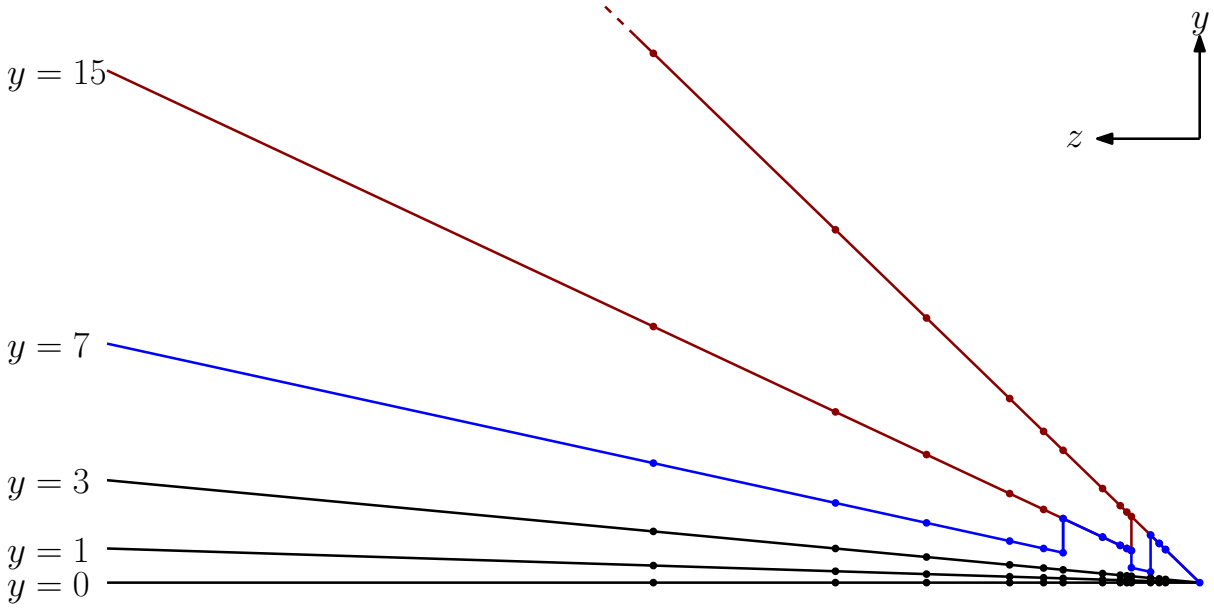


Figure 11.6: Projection of the third face of the ribbed fan during Step 1 of the algorithm. The edges (in blue) of the third face are projected on the edges of the previous faces above if they are at y-distance at most one. Connecting walls are added to maintain the connectivity of the face.

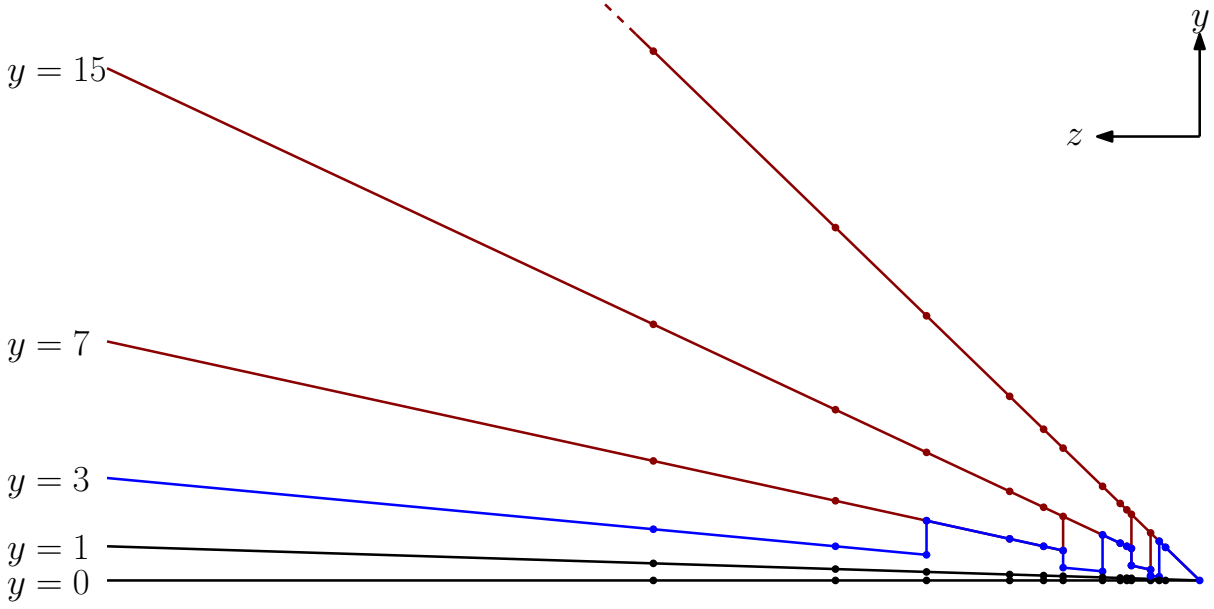


Figure 11.7: Projection of the fourth face of the ribbed fan during Step 1 of the algorithm. The edges (in blue) of the fourth face are projected on the edges of the previous faces above if they are at y-distance at most one. Connecting walls are added to maintain the connectivity of the face.

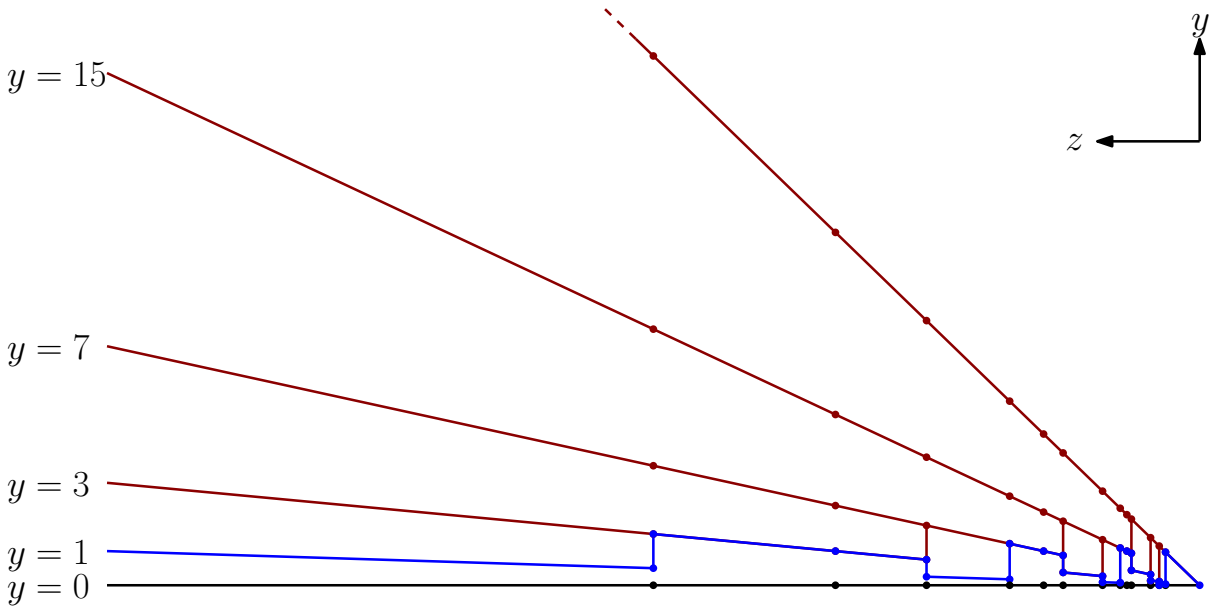


Figure 11.8: Projection of the fifth face of the ribbed fan during Step 1 of the algorithm. The edges (in blue) of the fifth face are projected on the edges of the previous faces above if they are at y-distance at most one. Connecting walls are added to maintain the connectivity of the face.

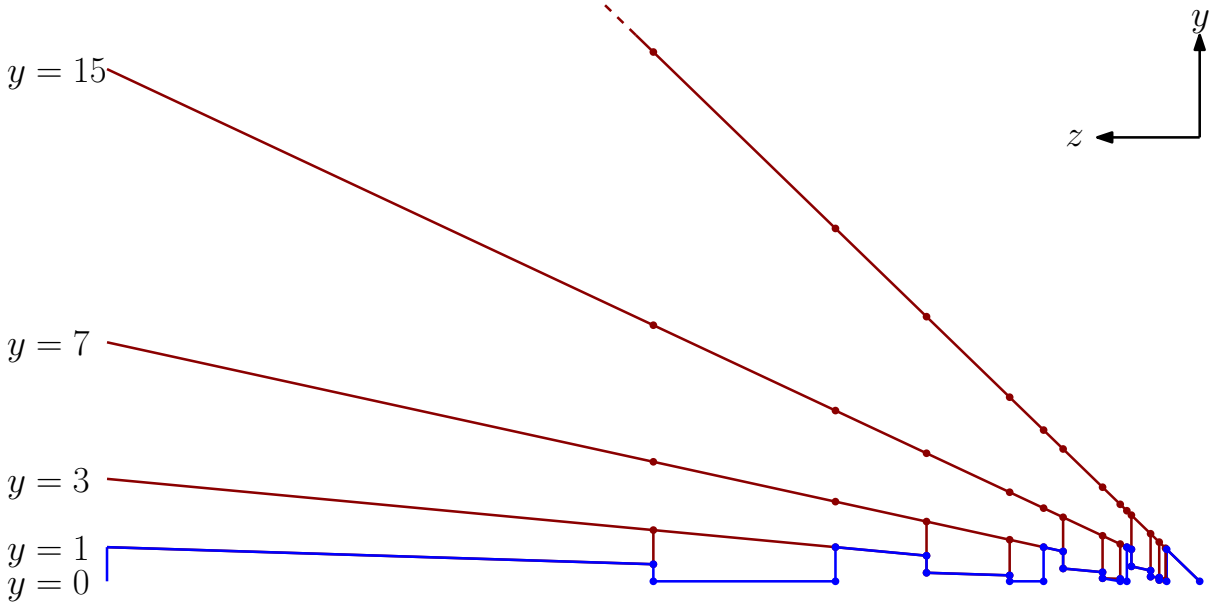


Figure 11.9: Projection of the sixth face of the ribbed fan during Step 1 of the algorithm. The edges (in blue) of the sixth face are projected on the edges of the previous faces above if they are at y-distance at most one. Connecting walls are added to maintain the connectivity of the face.

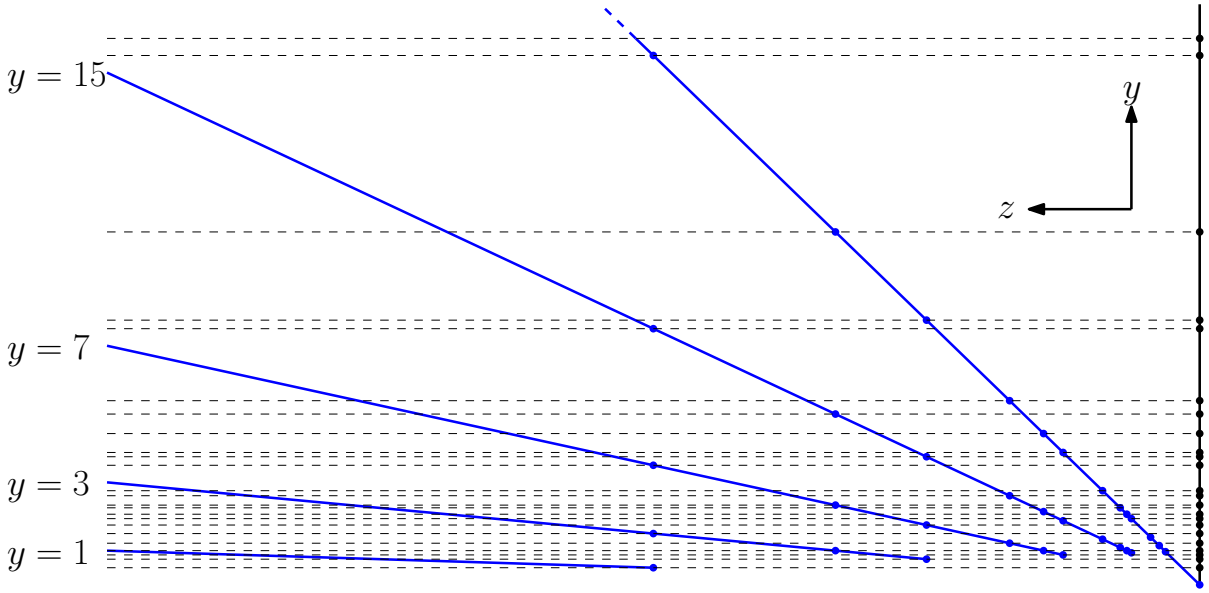


Figure 11.10: Subdivision of the faces of the ribbed fan during Step 2 of the algorithm. All vertices are projected onto the xy-plane (the y-axis on the figure). Some edges are not drawn for readability.

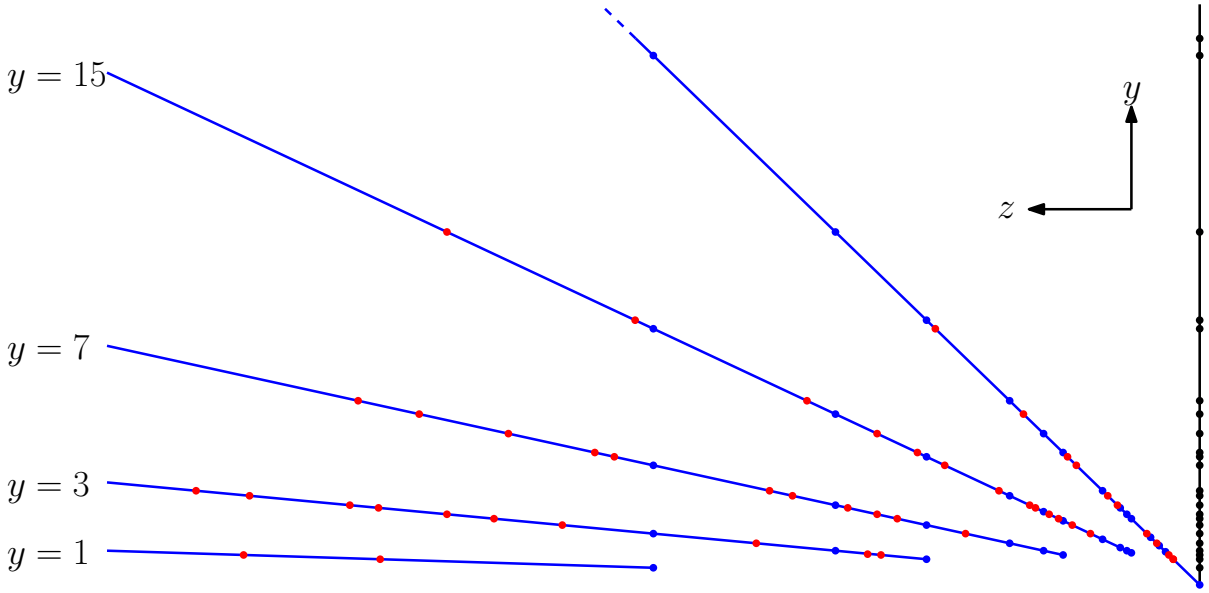


Figure 11.11: Subdivision of the faces of the ribbed fan during Step 2 of the algorithm. All vertices in the xy-plane (the y-axis on the figure) are lifted along the z-axis onto the faces. The vertices of $\mathcal{G}_C$ created are in red. Some edges are not drawn for readability.

Part III

3D mesh rounding heuristics

Chapter 12

State of the art

In this brief part, we present the current heuristics for rounding mesh vertices. This chapter offers an overview of existing methods, while our new heuristic will be introduced in the following chapter. The goal of these heuristics is to remove self-intersections in 3D models while maintaining vertex coordinates as double-precision values.

In the first section, we introduce naive methods for rounding, followed by Section 12.2, that present the heuristic by Zhou et al. [31], which inspired our approach. Experimental results for these heuristics are presented in Chapter 17.

12.1 Naive and iterative naive heuristic

A straightforward heuristic for removing self-intersections and rounding vertices of a mesh is to subdivide the model at its self-intersections and naively round all newly created vertices. However, rounding these newly created vertices may create new intersections between triangles of the mesh. On the self-intersecting models of Thingi10K, about 10% still present self-intersections after this naive operation (see Section 17.1). In the following, we refer to these (527) models as the *non-trivial models*.

On such non-trivial models, we can repeat this process until the model is no longer self-intersecting or after a fixed number of iterations. While very simple, this heuristic successfully resolves only half of the non-trivial models in Thingi10K (see Section 17.1), highlighting the need for more efficient methods.

12.2 Zhou et al.'s heuristic

The heuristic proposed in 2016 by Zhou et al. [31] is the state of the art in computer graphics for removing self-intersections. Their approach amonts to detecting triangles that are properly intersecting, rounding their vertices to floats, removing self-intersections by computing a self-union of the model, and then rounding new vertices to doubles. Repeating this heuristic while the model contains self-intersections or at most 20 times yields good results. According to Zhou et al. [31], all PWN models (see below) of Thingi10K are repaired except five.

The main originality of this heuristic is to round vertices of triangles that properly intersect on "coarse" float coordinates before computing their intersections. We believe that Zhou et al. heuristic works for two reasons: first, rounding vertices on coarse coordinates can collapse "needle" faces and other small features of the model. Indeed self-intersections are often created by triangles with incredibly tiny edges. The second reason is that the new points created by the self-union are computed from coordinates whose precision is less than half that of the input; these features are thus likely to be farther apart.

The heuristic of Zhou et al. suffers from the main limitation that due to the reliance on self-union, their approach is limited to models with Piecewise-Constant-Winding number (PWN). Roughly speaking, a PWN model is a model without boundaries. Indeed self-union cannot be defined on objects without interior and exterior. However, their approach can be easily generalized to any triangle soup without any assumption on the triangles, by replacing the self-union by a subdivision of the model along the self-intersections. This approach preserves triangles in the model's interior (if the interior is defined), which is not the case in Zhou et al.'s method.

Our experiments show good but not perfect results for this variant (see Section 17.2). Indeed, it fails on about 10% of the non-trivial models. On all these models, some vertices of intersecting triangles have coordinates close to zero. Floats are really precise close to zero and our interpretation is that rounding coordinates to float in this case is mostly useless, as tiny features like "needles" still exist and the new points created by the refinement are likely to be very close to some others. To confirm our intuition, we shifted some of the failing models such that all coordinates are larger than one, before applying the heuristic and the models were successfully rounded. In the next chapter, we present a new heuristic based on this intuition.

Chapter 13

Our heuristic

As said before, while we successfully produced a practical algorithm for 3D Snap Rounding, it is not as efficient as we initially hoped for. We present here a new heuristic for rounding vertices of a model. This heuristic ensures that the distance between the input and the output is bounded by a constant times the number of iterations but it does not guarantee that the output is intersection free and does not ensure to preserve the topology (up to collapse). However, this heuristic is incredibly efficient and is the first which successfully solves self-intersections and round all the models of Thingi10K.

As discussed in the previous chapter, our heuristic builds upon and improves the method proposed by Zhou et al. [31]. Instead of rounding certain vertices to floating-point values, as they did, we round them to an integer grid that is scaled to be as large as possible while ensuring the integral part of the coordinates remains representable as a float. However, rounding vertices to this grid can unintentionally introduce intersections with nearby triangles. To minimize this, we also round the vertices located within the same voxels as the rounded vertices to the integer grid.

The complete pipeline of our heuristic is as follows:

(i) Apply a scaling to the scene such that the largest absolute value of each coordinate lies within $[2^{22}, 2^{23})$, (this is the largest scaling so that the integral part of the coordinates are representable as float).[48]

(ii) Compute $\mathcal{V}$, the set of all the vertices of the pairs of triangles that properly intersect.

[48] In our implementation, this scaling is not applied to the whole model but applied only to individual coordinates just before rounding and is reversed immediately after.

(iii) Add to $\mathcal{V}$ all vertices that are in the voxels containing the vertices of $\mathcal{V}$.

(iv) Round all coordinates of the vertices in $\mathcal{V}$ to the nearest integer.

(v) Remove self-intersections by subdiving the pairs of triangles that properly intersect.

(vi) Round the coordinates of the new vertices created in Step (v) to double precision.

(vii) Repeat steps (ii) to (vi) until there are no remaining intersections in the model or at most a fixed number of times.

(viii) Perform the reverse scaling as the one done in Step (i).[48]

Our experiments show incredible results (see Chapter 17). Indeed, all models of Thingi10K except one are rounded without self-intersection. The exception is widely known to be very problematic due to its extremely large number of intersections. We also successfully solved this model by scaling on $[2^{15}, 2^{16})$ in Step (i). The distance between a vertex and its rounded image is bounded by $\frac{1}{2}$ times the number of iterations before the scaling back. This algorithm is very simple and efficient but it does not guarantee an intersection-free output if the fixed number of iterations is reached and it may not preserve the input topology.

It is worth noting that the choice of scaling is somewhat arbitrary. It was chosen to enable a direct comparison with Zhou et al. heuristic, but any scaling could be used as long as the values remain storable in double precision. Our experiments tend to show that a coarser scaling is more efficient at the cost of an increased distance between input and output. In our opinon, using a scaling that corresponds to the significant digits relevant to the model is the most appropriate. For example, the scaling considered above (in (i)) provides a precision of approximately 100nm for an object with a width of one meter.

Note that we subdivide intersecting triangles to be able to apply our algorithm to any triangle soup. However, it may be preferable and probably more efficient to use self-union, as done by Zhou et al. [31], when the input satisfies the required criteria.

As mentioned before, the experiments on this heuristic are presented in Chapter 17.

Part IV

Implementation

Chapter 14

Summary of the implementation

This part describes the implementation of our algorithm and its optimizations, detailed in Part II. This implementation contains about 20 C++ files and 8 000 lines of code excluding blank and commentary. The implementation is highly technical due to the presence of numerous degenerate cases, some that appear in the input model and some that are created during the algorithm. These degenerate cases have resulted in many bugs which have gradually been resolved. However, some bugs still persist in our implementation. As a result, the implementation is neither mature nor fully stable. Nevertheless, our implementation successfully resolved about 90% of the non-trivial models of Thingi10K and we were able to evaluate both the implementation and the algorithm (see Part V). This chapter provides overviews of each component of our implementation, the associated difficulties and our solutions to address them.

The next section introduces CGAL [28], the geometric library that plays a central role in our implementation, and the specific packages we used. Section 14.2 describes CGAL's number representation system and the associated implementation techniques. Section 14.3 focuses on the data structure while Section 14.4 focuses on the degeneracies. Section 14.5 to Section 14.8 provide overviews of each component and the associated difficulties.

14.1 CGAL

Our implementation relies heavily on the Computational Geometry Algorithms Library (CGAL) [28] for nearly all components: number representation, geometric operations between segments and/or triangles, triangulations, bounding box trees, and more.

In addition to the exact representation of numbers detailed in the next section, we used various basic geometric operations (e.g., intersections of segments), 2D constrained Delaunay triangulations to triangulate the faces and to compute triangulated arrangement of segments in 2D. Bounding box trees were also used to determine candidates for intersection tests.

We used CGAL's PolygonMeshProcessing package for preprocessing and postprocessing, but did not use this package for our data structures (see Section 14.3).

14.2 Number representation

In theory, our implementation can accept any exact number representation of a field that provides a ceil function. However, the implementation was tested exclusively with the Exact Predicates Exact Constructions Kernel (EPECK) from CGAL, for which we implemented a ceil function. Our algorithm generates new vertices that require exact representation before being rounded.

In EPECK, coordinates and other values are represented using lazy exact numbers [27]. More precisely, each value is represented by an interval of floating-point numbers that encloses the true value, and operations are carried out on these intervals. A Directed Acyclic Graph (DAG) is also used to record the history of these operations. When evaluating a predicate (which is basically determining whether a value is positive, negative, or zero), a "filter" is used first. The filters use only the intervals to answer a predicate (basically, by looking at the signs of the boundaries of the intervals). If a filter cannot answer a predicate (i.e., if an interval contains zero), referred to as a filter failure, the operations are recomputed using an exact representation and the predicate is evaluated again.[49] Generally, most predicates are resolved by the filters, minimizing the need for costly exact computations.

While CGAL hides these subtleties to the users, it is crucial to keep them in mind when designing an efficient implementation. Indeed, cascaded constructions lead to a deep DAG, which increases the likelihood of filter failures. Similarly, creating a vertex on a face and subsequently testing if the vertex lies on the face almost always triggers a filter failure. Our implementation attempts to avoid these inefficiencies. For example, when our algorithm subdivides edges, the new edges store not only their endpoints but also the vertices of the input edge that support them. This allows us to compute the new point coordinates of subsequent subdivisions using the

[49] In EPECK, the exact representation uses multi-precision rationals.

input edge's vertices, avoiding cascaded constructions. Additionally, when determining whether two edges are collinear, we check if they are supported by the same input edge, thereby avoiding filter failures during geometric tests.

14.3 Data structure

Our algorithm maintains a hierarchy of faces (see Section 5.3). Specifically, triangles are supported by faces in the set $\mathcal{G}$, which represent the faces obtained after the first three subdivision steps. These faces, in turn, are subfaces of those in the set $\mathcal{F}$, corresponding to the original input faces.

Our algorithm frequently resets $\mathcal{G}$ (to $\mathcal{F}$) or discards triangulations of faces in $\mathcal{G}$. These operations were difficult to implement using CGAL's Surface Mesh data structure, instead we developed a custom mesh data structure that mirrors this hierarchy. In our code, faces in $\mathcal{F}$ are referred to as **faces**, those in $\mathcal{G}$ as **gaces**, and triangles are simply represented as triplets of vertex indices.

The model is represented by three arrays: one for the vertices, one for the edges, and one for the faces. Vertices store both their original coordinates and their rounded coordinates. Faces store the indices of their three input vertices and edges, along with an array of the gaces they support. Each gace stores the indices of its incident edges and the triangulation of the gace. Additionally, each gace stores its supporting plane in the form of the indices of three vertices.

The stored edges correspond to those of the gaces. They naturally store the indices of the vertices at their extremities and those of their incident gaces. As mentioned in the previous section, they also store the indices of the vertices of the input edges that support them (if such edges exist).

If present, edges and gaces also store their above and below counterparts according to the floor invariant (see Chapter 6). When an edge or gace is subdivided, the corresponding edges and gaces with equal projections onto the floor are automatically subdivided as well to preserve the floor invariant.

Although this structure is more complex than standard mesh data structures, this part of the implementation is primarily combinatorial and stable.

14.4 Degeneracies

As previously mentioned, degeneracies hindered the progress of the implementation. In addition to the degeneracies already present in input models, our algorithm naturally introduces numerous new degeneracies. As described in Chapter 9, our algorithm also produces degeneracies involving vertices with identical coordinates that are rounded to different pixels or voxels. These are referred to as *tagged vertices*, where tags correspond to the pixel or voxel to which the coordinates are rounded. The following is a list of the degeneracies produced by the algorithm.

- Steps 1, 2, 4, and the coplanar optimization subdivide faces in the projection onto axis-aligned planes, leading to coplanar faces and also to faces with identical projections on some axis-aligned planes.
- Step 1 additionally produces connecting walls that form faces perpendicular to the backwall (xz-plane).
- Step 4 and the coplanar optimization make use of 2D triangulated snap rounding, which introduces tagged vertices.
- Step 3 subdivides faces using vertical slabs with half-integer x-coordinates. The created vertices are thus tagged and the created edges are parallel to the sidewall (the yz-plane). When the faces are subdivided by two adjacent vertical slabs, geometrically identical edges with distinct tags are created (on the sidewall common to the two slabs).
- Local optimizations subdivide edges and gaces by the boundaries of 1D or 2D pixels, again producing tagged vertices.
- During the rounding simulation, coordinates are rounded one at a time, leading to vertices with identical x-coordinates up to a symbolic factor (see Section 7.4) while the y-coordinates are being rounded (and similarly during the rounding of z-coordinates).

Due to the numerous degenerate cases arising from these operations, the implementation is complex and remains somewhat unstable, as noted earlier. The code is designed to first identify all possible degenerate cases and then invoke the appropriate function for each.

As mentioned in Chapter 9, tagged vertices are considered to be shifted by an infinitesimal factor. However, CGAL does not support number types with infinitesimal perturbation. As a

result, some geometric operations are extended, such as the intersection of two segments, to ensure that any vertices resulting from these operations are correctly tagged. For instance, if a 2D vertical edge has both vertices tagged to round to the pixel on the right, any new vertex created on this edge will also be tagged to round to the pixel on the right.

For other geometric operations, such as triangulation, we temporarily shift the coordinates by a small but non-infinitesimal value. This shift is calculated to be small enough to avoid altering the topology of the structure, and it is discarded afterward to preserve the original geometry.

14.5 Intersection tests

For the rounding simulation, we implemented the intersection test for two features (edge/edge or vertex/face), as described in Chapter 7. As mentioned in the previous section, our test must handle degenerate cases. Experiments showed that many detected intersections were degenerate, and do not result in topological changes or proper intersections in the output.

We explored several methods to reduce the number of detected intersections. The first approach was to consider trimmed coordinate-by-coordinate motion instead of the standard coordinate-by-coordinate motion (see Section 7.4). Another method involved deliberately ignoring some degenerate proper intersections, while a third approach introduced perturbations to the triangles before rounding. However, these two methods contributed to the instability of the code rather than mitigating it, and were ultimately discarded.

The final method we developed to reduce the number of degenerate intersections involved testing intersections between two features only if they are incident to triangles that properly intersect at the end of the rounding process. This approach successfully avoided many often-degenerate cases where triangles properly intersect during the motion but not at the end, thus enhancing the stability of our implementation.

However, a triangle could completely pass through another during rounding without intersecting at the end, thereby the test misses a potential topological change. Even though such scenario is extremely rare, they might theoretically occur. It thus forfeits the topological guarantee of our algorithm but we believe that this approach allows us to better evaluate the true potential of our algorithm by avoiding some of the more significant implementation issues.

14.6 Triangulation of the gaces

Before rounding, we need to triangulate the gaces. We aimed to use constrained Delaunay triangulation for this; however, as mentioned before, CGAL's implementation does not allow the insertion of two geometrically identical vertices. To work around this, before inserting a tagged vertex into the Delaunay triangulation, we shift it slightly along the input edge supporting it (see Chapter 9). If the tagged vertex is not supported by an input edge (as is the case for the corners of hot pixels), we shift it toward the center of its voxel. The shift is computed to be small enough to avoid changing the arrangement's topology. This shift is only applied for computing the triangulations and it is discarded afterward.

14.7 Step 3 of the algorithm

Recall that the third step of the algorithm involves subdividing certain faces using well chosen narrow slabs while maintaining the floor invariant (see Section 6.4). Subdividing a gace using a narrow slab is straightforward. As mentioned earlier, to maintain the floor arrangement, edges and faces store references to their counterparts above and below. Thanks to this data structure, whenever a face or edge is subdivided, the corresponding subdivisions for the above and below features are triggered automatically, resulting in a robust implementation.

14.8 PSL-based components

Recall that the PSL algorithm, presented in Section 3.4, consists of three main steps: (P) project all input faces onto the floor (the xy-plane); (S) subdivide them as described in 2D Triangulated Snap Rounding; and (L) lift this triangulation back along the z-direction onto all faces, subdividing them accordingly.

We use variants of this PSL algorithm in several the building blocks of our algorithm, namely Steps 1, 2, 4 and in the coplanar optimization, as detailed in the three following subsections. Even though we use different variants of this algorithm in these building blocks, significant factorization between these variants is possible, leading to a more robust codebase even if some differences prevent the use of a single implementation for all these operations.

The PSL algorithm requires triangulated 2D snap rounding, which we implement using

the lazy variant described in Section 8.4. The implementation uses Constrained Delaunay Triangulation (CDT) to perform the subdivision. We aimed to add the boundaries of the hot pixels to the arrangement and like the triangulation of the faces, we shift all pixel corners by a small value to the center of their corresponding voxels. The shifting value is computed so that the topology is not changed during the shift. This shift is used to compute the triangulation and it is discarded afterward.

14.8.1 Step 4 of the algorithm and coplanar optimization

Step 4 (Section 6.5) and the coplanar optimization (Section 8.2) both directly use the PSL algorithm presents in Section 3.4. They differ in their use of the PSL algorithm only in the stage of the face hierarchy at which subdivisions occur. Specifically, coplanar optimization subdivides gaces, while Step 4 only provides a triangulation for them.[50]

14.8.2 Step 2 of the algorithm

Step 2 (Section 6.3) is a variant of the PSL algorithm. As a reminder, it projects all input faces onto the floor, computes their triangulated arrangement, and lifts this triangulation back along the z-direction, subdividing them accordingly.

In our implementation, Step 2 also builds the data structure to maintain the floor invariant. Each edge and gace in 3D stores the indices of their above and below counterparts as mentionned in Section 14.7. To build this structure, for each edge and triangle in the triangulated arrangement on the floor, we simply sort the corresponding features above it, based on their z-coordinates. After this sorting, constructing the data structure becomes straightforward.

14.8.3 Step 1 of the algorithm

As mentioned in Section 6.2, the initial version of the first step focused on creating a more localized variant, which aimed to reduce the number of face subdivisions. The approach processed pairs of critical faces in lexicographic order. First, it calculated the region at a y-distance of at most one by computing the intersection of two faces on the backwall, bounded by the projection of the four lines at a y-distance one of their supporting planes. The faces would then be subdivided accordingly without triangulating them. Next, we projected the face at a y-distance of one and

[50] Some junction vertices may be added to the gaces during these triangulations (see Section 6.5).

created the connecting walls. After projecting a face onto another, the remaining unprojected parts of the faces formed a set of polygons with holes, which would be used for further projections onto other faces. However, due to the complexities in handling these polygonal sets with holes and managing degenerate cases, we were unable to develop a fully functional implementation.

As a result, we developed the current version of the first step (see Section 6.2), which also uses a variant of the PSL algorithm. Namely, it projects all critical faces onto the backwall and computes their triangulated arrangement. Additionally, for any pair of faces in $\mathcal{G}_C$ that are within a y-distance of at most one, we add to the triangulation the projection of lines at y-distance one of their supporting planes, truncated by the projection of the faces. Finally, the 2D triangulation is lifted back into 3D along the y-axis on the critical faces.

Being a variant of the PSL algorithm already implemented for the previous components, this operation was straightforward to implement. After completing these subdivisions, the subsequent steps are relatively simple: for each triangle in the triangulated arrangement on the backwall, we sort by their y-coordinate the gaces above that project on the triangle. In that order, we project each triangle onto the previously non-projected one if the y-distance between them is at most one, before creating the connecting walls. While this approach is more stable and significantly simpler, it has the drawback of potentially producing more subdivisions.

In summary, modifying the first step allowed us to avoid polygonal sets with holes and iterative subdivisions on the backwall, resolving most implementation issues at the cost of performing additional subdivisions.

Part V

Experimentations

Chapter 15

Experiments environment

In this part, we describe and analyze the results of our experiments on the implementation of both our algorithm and heuristics. The first chapter outlines the experimental environment, while the second and third chapters detail the experiments related to our algorithm and heuristic, respectively.

15.1 Thingi10K

Most of our experiments were conducted on a dataset of 10,000 meshes named Thingi10K [32]. This dataset was created using objects from the model-sharing website for 3D printing called Thingiverse (`https://www.thingiverse.com`). This dataset is significant because it reflects the variety, complexity, and (lack of) quality of 3D models. Since our work falls within the broader subject of mesh repair, datasets of low-quality meshes are relevant. Three models in Thingi10K are unreadable[51] and were excluded from our tests. Of the remaining models, 4,524 exhibit self-intersections and were the primary focus of our experiments. Most models are in *stl* format, with a few exceptions (obj, off, ...).

15.2 Grid'5000

Grid'5000 [4] is a large network of computation nodes for scientific experimentation in France. Our experiments were run on the nodes named "gros" from the default queue resources at the

[51] These three models are referenced as 49 911, 81 313, and 77 942.

Nancy site. The CPUs on these nodes are Intel Xeon Gold 5220 with 18 cores. Our code always used only one core and did not use parallelization.

Chapter 16

Experiments on our certified algorithm

In this chapter, we perform experimentations on our implementation of the algorithm described in Part II. Despite difficulties to manage the degenerate cases, we succeeded in solving 94% of the Thingi10K models, demonstrating the algorithm's practicality. The sections are structured as follows: we first present naive rounding in Section 16.1, followed by the results of our certified algorithm in Section 16.2, and results with relaxed certification constraints in Section 16.3. Sections 16.4 to 16.6 provide details on the number of faces subdivided, the execution time and output size. In Section 16.7, we validate our optimization designs. Finally, we conclude and give a summary of these experiments in Section 16.8.

16.1 Non-straightforward models

We began by investigating how many models exhibit proper intersections during vertex rounding. For each self-intersecting model, we resolved the intersections by creating new vertices with exact coordinates.[52] The modified models were then scaled and rounded so that the largest absolute coordinate fit within the range $[2^{51}, 2^{52})$ (this is the largest scaling so that the integral part of the coordinates are representable as double). On the 4,524 self-intersecting models, 520 presented proper intersections after rounding, which we refer to as the "non-straightforward" models for further experimentations in this chapter.

[52]The size of input in the experiments is the one after these subdivisions.

16.2 Certified algorithm

We tested our algorithm with its optimizations (see Part II) on the 520 non-straightforward models. Of these, 187 did not terminate within the one-hour time limit. Among them, one model crashed during the intersection resolution (see Section 17.3), 26 models crashed within our code, 46 entered infinite loops, and 37 ended with more intersections than they started with. In 77 cases, it was hard to determine whether the algorithm would eventually finish. At least half of the failures can be attributed to code shortcomings, motivating a change in our intersection test, which we detail in the next section.

16.3 Certified algorithm with modified intersection test

To address the observed shortcomings, we modified the intersection test as described in Section 14.5, checking only triangles that intersect properly at the end of the rounding. This variant sacrifices topological guarantees but helps evaluate the algorithm's core performance. With this modification, 52 models (10%) failed to finish within a one-hour time limit and 33 (about 6%) models failed within a ten-hour time limit. Three failures still occurred due to crashes, five in loops, and ten in excessive intersections and in 14 cases, it is hard to answer.[53]

16.4 Number of faces subdivided

We argue that, unlike the algorithm of Devillers et al. [8], our algorithm remains local, subdividing only the problematic faces and their immediate neighborhood. To support this claim, we compare the number of faces involved in proper intersections during the initial simulation of the rounding with the total number of faces subdivided (see Fig. 16.1).[54] In the worst case, our algorithm subdivides at most twice the number of initially intersecting faces.[55]

Next, we compare the number of subdivided faces to the total number of faces in the original model (see Fig. 16.2). As expected, the number of subdivided faces varies greatly depending on the model's complexity. However, in all cases, the number of subdivided faces remains below

[53] The last model crashed during the intersection resolution (see Section 17.3). Many of these models can likely be addressed in a more mature implementation.

[54] For models where the algorithm does not terminate, we record the number of subdivided faces at the point where the algorithm stops.

[55] If the edges of these faces are subdivided, the adjacent faces are triangulated as well.

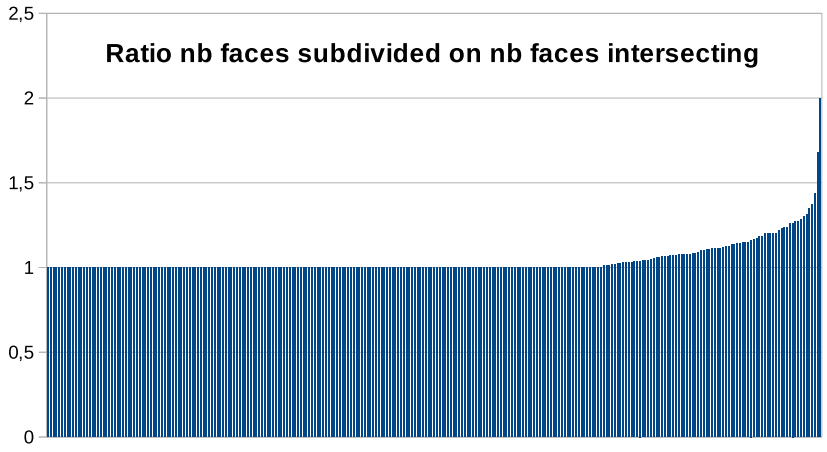


Figure 16.1: Comparison between the number of subdivided faces and the number of faces involved in proper intersections during the first simulation of the rounding. The y-axis shows the ratio of subdivided faces to intersecting faces, while the x-axis represents each of the 520 non-straightforward Thingi10K models, ordered by their corresponding y-values (one pixel per model).

10% of the total faces in the model, and below 1% for the majority of cases.

16.5 Running time

We analyze the running time of our algorithm, beginning with the 4 004 straightforward models (see Fig. 16.3). In these cases, the algorithm first addresses self-intersections as a preprocessing step before performing the rounding simulation. Consequently, we anticipate that computing the intersections of the bounding boxes between features will be the algorithm's bottleneck, leading to an expected running time of approximately $O(n \log n)$. Our results show that the running times are effectively near-linear: the modified intersection test takes about 3 seconds for 1 000 vertices, while the standard intersection test takes around 8 seconds for the same number of vertices. There is an acceleration ratio of about 3 between the two intersection tests.

We next analyze the running times for the 520 non-straightforward models from Thingi10K that successfully terminate (see Fig. 16.3). Subsequent rounding simulations are performed solely on modified features and their neighborhoods. As the subdivisions remain limited to a small number of faces (see Section 16.4), we expect the running time to remain near-linear. Indeed, our observations confirm this expectation: the running time is approximately 7 seconds for 1 000 vertices with the modified intersection test and about 22 seconds for the standard intersection

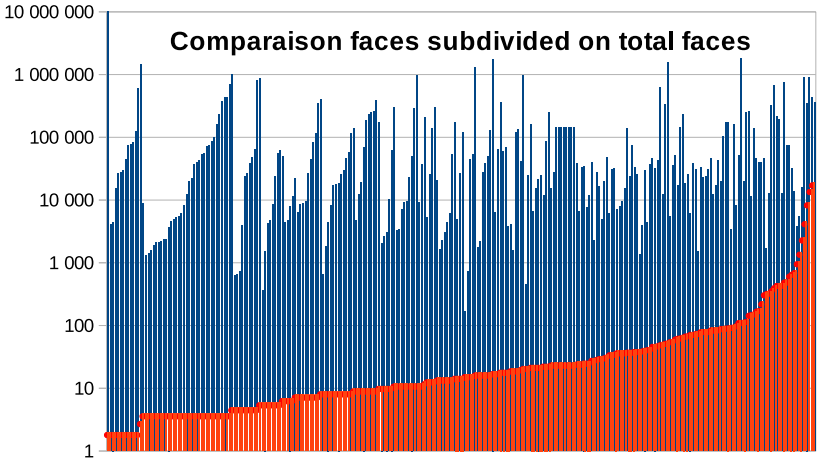


Figure 16.2: Comparison of the number of subdivided faces to the total number of faces. The blue y-axis represents the total number of faces, while the red y-axis indicates the number of faces that have been subdivided. The x-axis displays each of the 520 non-straightforward models from Thingi10K, arranged in order of their corresponding y-values (one pixel per model).

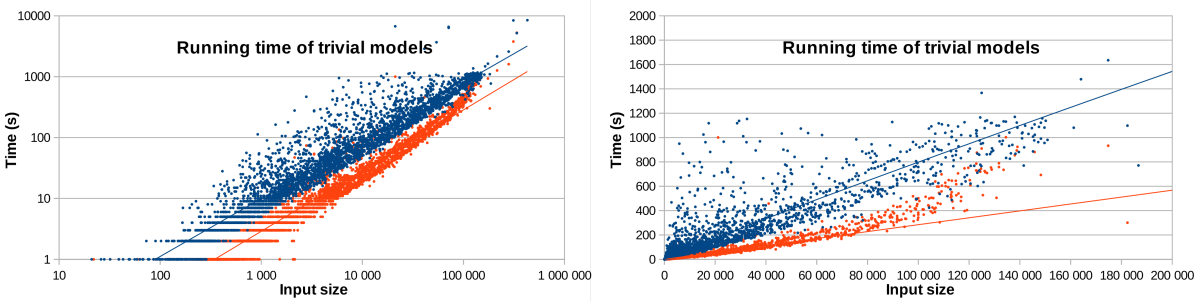


Figure 16.3: Running times (in seconds) for the 4 004 straightforward models of Thingi10K as a function of the number of input vertices. The running times with the modified intersection test are displayed in red, while those with the standard intersection test are shown in blue.

test. Once again, the acceleration ratio between the two intersection tests is about 3.

While these running times may seem substantial for a mesh processing algorithm (notably, compare to our heuristics tested in the next chapter, which takes only a few seconds for 10 000 vertices), they are still within an order of magnitude of what is typical for mesh processing algorithms. Furthermore, it is important to note that our algorithm is significantly complex, and reducing its running time was not our primary focus.

16.6 Size of the output

We investigate the output size of our algorithm. Some vertices may collapse during the rounding motion, and since the number of subdivided faces remains low (see Section 16.4), we expect the

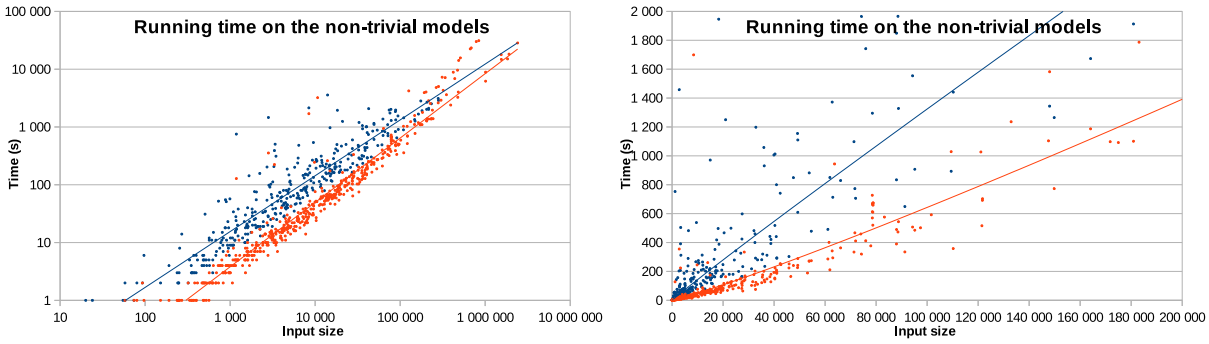


Figure 16.4: Running time (in seconds) for the 487 terminating non-straightforward models of Thingi10K as a function of the number of input vertices. The running times with the modified intersection test are displayed in red, while those with the standard intersection test are shown in blue. The figure on the left uses logarithmic axes, while the one on the right uses linear axes.

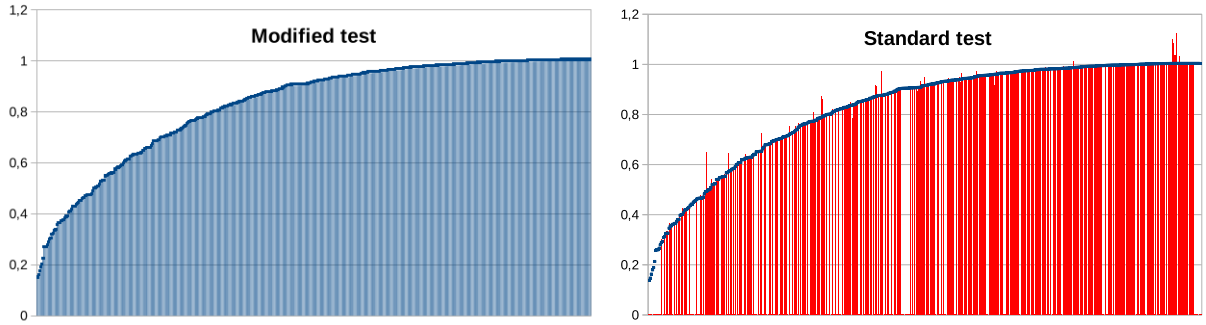


Figure 16.5: Output size versus input size. The y-axis represents the ratio of the number of vertices in the output to the number of vertices in the input. The x-axis corresponds to one pixel for each of the 487 terminating non-straightforward models from Thingi10K, ordered by their y-values based on our modified intersection test. Blank columns indicate models that did not terminate with the standard intersection test.

output size to be approximately equal to that of the input. Figure 16.5 illustrates the size of the output in comparison to the input size. In practice, the output size is almost always smaller than the input size, indicating that our algorithm does not oversubdivide the models.

To validate that the modified intersection test does not significantly alter the output of our algorithm, we compare the outputs of 333 models for which the standard intersection test terminates with those obtained using the modified intersection test. Approximately half of the models yield strictly identical outputs regardless of the intersection test used, and only 24 models exhibit a difference greater than 2% in the number of vertices between the two intersection tests (see Fig. 16.5).

16.7 Validation of optimization designs

16.7.1 Certified algorithm without the optimizations

To support our decision to develop the optimizations described in Chapter 8, we conducted an experiment similar to that in Section 16.3 with the modified intersection test. Recall that only 52 models did not finished with the optimizations. This time, we executed only our algorithm without any optimizations.

Out of the 520 non-straightforward models, 374 did not finish within one-hour limit. Among the 146 that finished, 31 do not initially present proper intersections with the modified test. In 61 cases, $\mathcal{F}_C$ remain smaller than 10 while for 54 models, the number of critical faces ranged between 10 and 279. In these last models, the critical faces are almost all coplanar to the floor (xy-plane) or the backwall (xz-plane) at the end of the rounding, except for one model that has 26 critical faces.

This suggests that, without the optimizations, our algorithm struggles to handle models with a high number of intersections between non-coplanar faces during the rounding process.

16.7.2 The fan: a pathological example

To further justify our decision to develop the optimizations described in Chapter 8, we present an example in this section that highlights how the number of subdivisions performed by the algorithm can skyrocket, even with a small number of faces.

In the models of Thingi10K, vertices with a high degree that form the corners of a cube are a very common structure (see Fig. 16.6). When subjected to random noise, these structures are prone to exhibit a high number of faces that are nearly coplanar with the sidewall (the xy-plane), which is pathological for the first two steps of our algorithm. To demonstrate this, we conduct experiments on a "fan" (see Fig. 16.7) that resembles this type of structure. In Section 11.2, an alternate version of this fan is used to prove a lower bound on the complexity of our algorithm.

All pairs of faces in this fan properly intersect in projection onto the floor and the backwall (see Fig. 16.8), resulting in numerous subdivisions during the first two steps of the algorithm (see Fig. 16.9). As a result, The fan in a version consisting of only four faces, contains approximately 7 000 vertices before the final rounding (see Fig. 16.10).

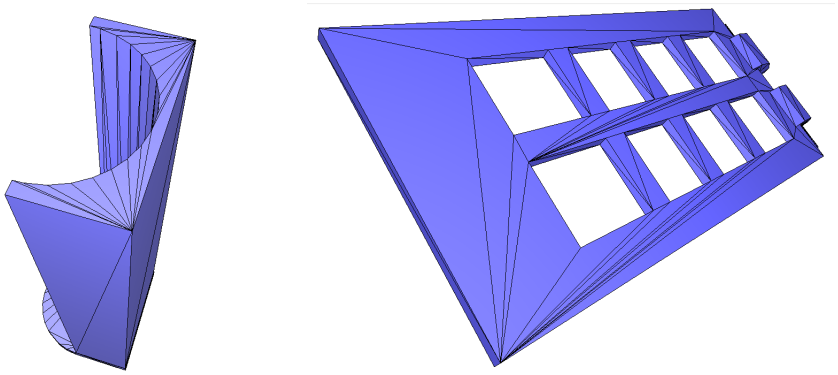


Figure 16.6: High-degree cube corners in the first two models of Thingi10K.

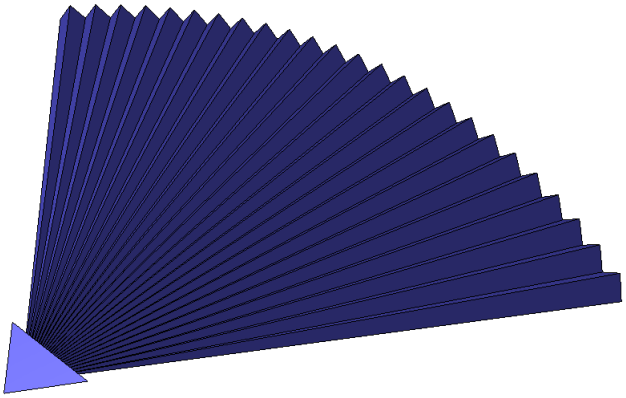


Figure 16.7: The pathological fan in 3D.

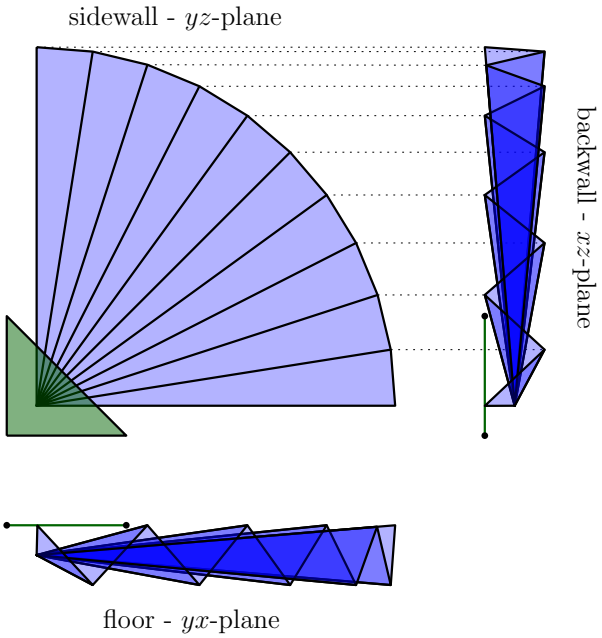


Figure 16.8: The fan projected on each axis-aligned plane.

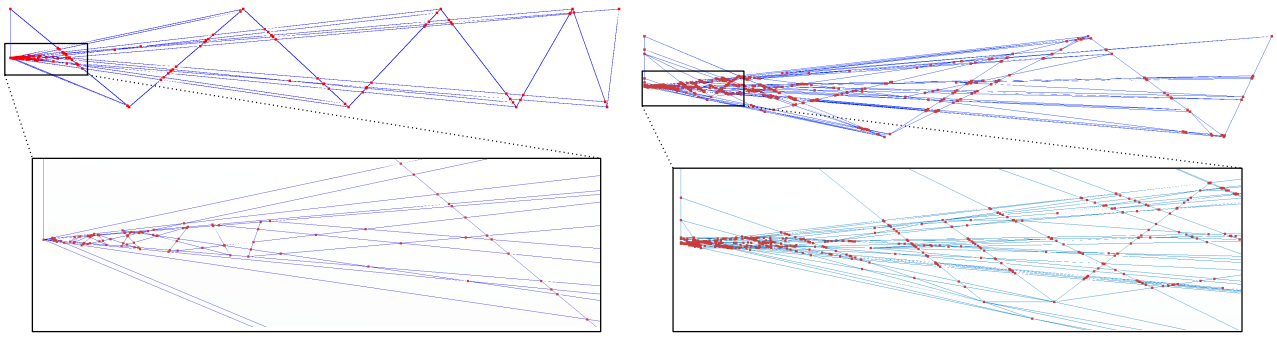


Figure 16.9: Arrangement on the backwall for the fan of eight faces on the left, and the floor arrangement at the end of Step 2 of the fan of four faces on the right. Triangulation edges are not shown.

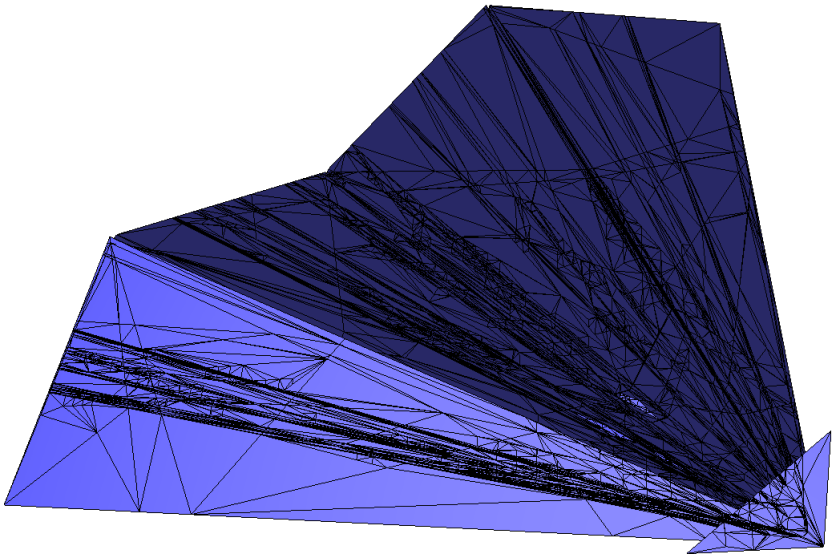


Figure 16.10: The fan of four faces at the end of the algorithm, just before the rounding process.

16.8 Conclusion

The instability of our implementation prevents us from confidently claiming that our algorithm is both effictive and scalable. Nonetheless, our results surpass those obtained with the heuristic proposed by Zhou et al. (see Chapter 17), which was considered the state of the art for rounding vertices in computer graphics. We contend that our experiments demonstrate the practicality of our algorithm in effectively rounding real models such as those from Thingi10K, including difficult ones.

Chapter 17

Experiments on heuristics

In this chapter, we present experiments on naive rounding, iterative naive rounding, Zhou et al.'s heuristic, and our new heuristic. These heuristics are presented in Part III.

The goal of these experiments is to evaluate the performance of these heuristics when removing self-intersection in 3D meshes while preserving the vertex coordinate as doubles. We mostly test these heuristics on the dataset Thingi10K [32]. All experiments involve resolving the self-intersections of these models using CGAL's exact representation and then testing various rounding processes on the coordinates. We consider an experiment failed or unfinished for a model if the last rounded version still presents self-intersections, or if the implementation does not terminate in less than the time limit.

To remove the self-intersections of a model, we subdivide its faces along the intersections, using an exact arithmetic for the new coordinates using CGAL library [28].

The experiments environnement is exactly the same as for the experiments on the main algorithm; see Chapter 15 for details.

We use exact arithmetic for the predicates to ensure correct results in the intersection tests; this ensures that when the heuristic terminates within the 20 iterations and the time limit, the output is certified to be intersection free. We also use exact arithmetic in the constructions of the new vertices to control the maximum distance between the input and the output. One can use exact predicates and inexact constructions to achieve a certified output with potentially faster runtime, although less control on the input-output distance.

Sections 17.1 to 17.3 present the experiments on these heuristics performed on Thingi10K. Sections 17.4 and 17.5 detail respectively the output size and running time of each heuristic in these

experiments. We experiment alternative scalings for our heuristic in Section 17.6. Section 17.7 presents experiments on a difficult artificial example. Section 17.8 presents experiments to validate the design of both our heuristic and that of Zhou et al. We conclude and give a summary of these experiments in Section 17.9.

17.1 Naive rounding and iterative naive rounding

First, we test the naive snap rounding. The naive snap rounding consists in removing self-intersections and rounding the newly created vertices to doubles. Out of the 4 524 models in Thingi10K that contain self-intersections, the naive rounding fails on 527 models ($\approx 12\%$).[56] We consider these 527 models as the non-trivial models of Thingi10K and almost all the results of the next sections are presented considering these models rather than on all the self-intersecting ones.

If we repeat these operations (as describe in section 12.1) at most 20 times within an hour limit, 284 of the 527 models ($\approx 55\%$) still exhibit self-intersections.

We performed up to 20 iterations for comparison with Zhou et al. heuristics [31], but it seems rather useless to iterate a high number of times. Indeed, only 32 models were successfully completed after more than five iterations, and only 13 after more than ten. Among the 284 unfinished models, 197 models failed due to time limit (not due to the maximum number of iterations) and 147 models performed fewer than 10 iterations within an hour.[57] For these 284 unfinished models, further iterations seemed futile for most, 237 had more intersections in the last iteration than in the first, likely meaning they would not finish for any reasonable number of iterations.

In conclusion, the naive rounding fails quite often, that is in about 12% of the self-intersecting models of Thingi10K, and the iterative naive rounding solves only about half of those where the naive rounding fails.

[56] We have here 527 models on which the naive rounding fails, although in Section 16.1, we had 520 models on which the rounding failed. This discrepancy comes from the fact that we round here to doubles and test if there are intersections after rounding, while in Section 16.1, we rounded to a uniform grid and tested intersections during the whole rounding motion.

[57] Some are large models, but for many, the number of intersections skyrockets with iterations, thus increasing computation time.

17.2 Zhou et al. heuristic

Recall that Zhou et al. heuristic can only be applied to PWN models (see Section 12.2). Of the 10 000 models in Thingi10K, 8 616 meet this criterion, with 3 413 exhibiting self-intersections. In their paper, Zhou et al. [31] reported that only five models still had self-intersections after up to 20 iterations. We ran experiments with our variant of Zhou et al.'s heuristic (see Section 12.2).

First, for direct comparaison with Zhou et al.'s results, we applied our variant of their heuristic to the 3 413 self-intersecting PWN models from Thingi10K, with a limit of 20 iterations and 10 hours of computation. Fifteen of these models still exhibited self-intersections (note that eight of these models are nearly identical and showed identical behaviour after the first iteration).[58] Zhou et al. reported five models that failed instead of 15, which likely due to our use of self-intersection subdivisions instead of self-union.

We then applied this approach to all the models in Thingi10K, adding the 1 381 non-PWN models, 1 111 of which exhibited self-intersections and which are generally of lower "quality" than the PWN models. As expected, this heuristic works on all models for which the naive rounding works. Among the 527 non-trivial models, 48 did not terminate after 20 iterations or 10 hours, which is about 9% of the 527 models.[59]

Although we performed up to 20 iterations to compare directly with Zhou et al., like with the iterative naive rounding, iterating many times seems useless. Only 16 models were successfully completed after more than five iterations, and just 4 after more than ten. For the unfinished models, further iterations seemed futile for most. In 37 models, the number of intersections increased with each iteration, and the runtime per iteration increased with the number of intersections. Six models got stuck in a loop, with a cyclic number of intersections after more than 100 iterations.[60] Five models seemed likely to finish with more iterations and indeed did so after 21, 22, 32, 36, and 53 iterations, respectively.[61] Of the 52 unfinished models after ten iterations, only nine seem likely to terminate if we had removed the number of iterations limit and the time limit.

In conclusion, while this approach is generally effective, it shows limitations on non-simple

[58] The eight nearly identical models are referenced as 42 323 to 42 331, excluding 42 330. The other models are referenced as 101 632, 131 971, 1 619 332, 247 516, 356 074, 44 874, 74 767.

[59] 11 trivial models are solved with more than one iteration.

[60] These models are referenced as 38 558, 71 377, 131 328, 131 971, 233 963, 356 074.

[61] These models are referenced as 113 422, 521 600, 86 324, 128 001, 496 388.

models. Indeed, it still fails on 9% of the 527 models where the naive rounding failed.

17.3 Our heuristic

We tested our heuristic (see Chapter 13) on the 4 524 self-intersecting models in the Thingi10K dataset. All models, except one, successfully completed in less than half an hour. Only 26 models required two iterations, and only 3 required three iterations.[62] This heuristic is simple, easy to implement, and remarkably efficient. However, it does not guarantee an intersection-free output when the maximum number of iterations is reached and it may not preserve the input topology.

The failed model, referenced as number 996 816 in Thingi10K, is notoriously difficult due to its incredibly high number of intersections and has caused failures in many various implementations. The implementation crashes on this model during the self-intersection refinement. This model contains 76 111 vertices and 171 436 triangles (after standard cleaning). Of these triangles, 86 552 are involved in 5 048 804 pairs of intersecting triangles; each of these triangles intersects an average of 120 other triangles. Computing the intersection between two triangles generally creates two new vertices, so refining this model would generate at least 10 million vertices, and likely far more due to vertices created by the intersection of three triangles. This explains why the computation of the intersections fails. However, we successfully rounded this model by rounding the vertices on a coarser grid, namely by scaling the largest absolute value of each coordinate to the interval $[2^{15}, 2^{16})$ instead of $[2^{22}, 2^{23})$. Choosing a coarser grid collapses some intersecting triangles before resolving intersections. Even with this coarser grid, the output contains about 5 millions vertices and 30 millions faces. We produced a smaller output with 272 387 vertices in one iteration by scaling to the interval $[2^{14}, 2^{15})$ instead of $[2^{15}, 2^{16})$. Even with this coarse grid, the distance between input and output is at most about $30\mu m$ for an object width of one meter and the output is visually identical to the input (see Fig. 17.1). The running times for these two scalings are 3 minutes and about 40 minutes.

Note that any model can be rounded using this method by choosing a "coarse enough" grid, at the cost of increasing the distance between the input and output. Indeed, in the extreme case of a grid with only one voxel, all intersecting triangles collapse to one point, resolving the self-intersections.

[62] These models are referenced as 1 368 052, 106 838, 78 227. 16 trivial models are solved with two iterations.

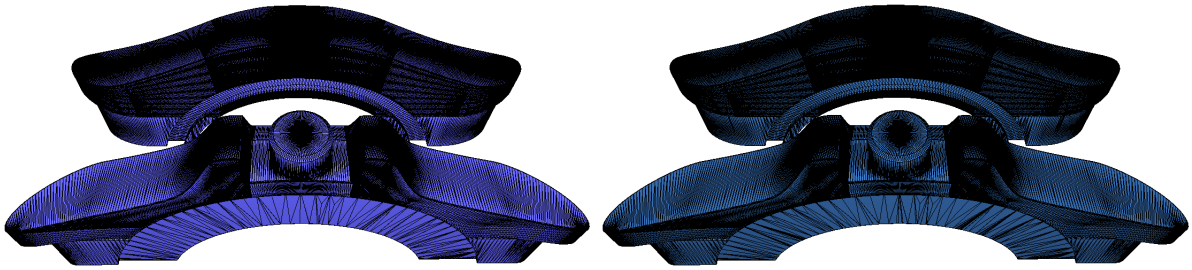


Figure 17.1: Left: Model referenced as number 996 816 in Thingi10K with 76 111 vertices, 171 436 triangles and 5 048 804 pairs of intersecting triangles. Right: Output by our heuristic with a coarse grid with 272 387 vertices, 1 167 742 triangles and no proper intersections.

17.4 Size of output for the three heuristics

We study here the number of vertices in the output for the three different methods.[63] Good predictors for the size of the output are the size of the input and the number of pairs of properly intersecting triangles. Recall that refining the intersection of two triangles generally creates two vertices.[64] Thus, we define the "input size" of a model as the number of vertices plus twice the number of pairs of properly intersecting triangles. This definition of input size gives a good approximation of the size of the arrangement of triangles while being easier to compute.

Definition 34. *We define the input size of a mesh as $n + 2k$, where n is the number of vertices in the mesh and k is the number of pairs of properly intersecting triangles in the mesh.*

Fig. 17.2 shows the number of vertices in the output of each heuristics relative to the input size of the model for each of the 527 non-trivial models. This ratio ranges between 0.1 and 1.79 for all heuristics, making it a reasonable estimate of the output size regardless of the heuristic. For our heuristic, this ratio averages to 0.79 with a median of 0.86. For Zhou et al.'s heuristic, it averages to 0.81 with a median of 0.88. For models where the iterative naive heuristic terminates, the ratio averages to 0.84 with a median of 0.91.[65] The models with a low ratio are more likely to fail under the iterative naive heuristic, and if they do not, their output is significantly larger than ours (see Fig. 17.2).

[63] The number of faces in a triangulated surface without boundaries is $2n - 4 + 4g$, where n is the number of vertices and g is the genus. Output models are typically quasi-surfaces, with the genus being far smaller than the number of vertices, so the number of faces in the output is nearly twice the number of vertices. Note that this is not always true for the input.

[64] Degenerate cases between two triangles create between zero and six vertices to refine self-intersections. Additional vertices can be created by the intersection of three triangles.

[65] The average and the median are computed only for solved models. These values for the iterative naive rounding are thus computed for only half of the non-trivial models. Our approach averages to 0.81 with a median of 0.89 on the same models than iterative naive heuristic.

In Figure 17.2, six models have an output size that is much larger than the input size (a ratio much larger than one). These six models all have many triplets of intersecting triangles, which implies that $n + 2k$ is not a good predictor of the input size in those cases.

On average, our heuristic produces models that are 7.1% smaller than those produced by the iterative naive heuristic and 5.4% smaller than those produced by Zhou et al.'s heuristic. However, as seen in Fig. 17.2, the variance in these ratios is high. For most models, the output sizes of the three heuristics are nearly identical, but for a few models, our output is significantly smaller.

17.5 Running time of the three heuristics

We study here the running times of the three different heuristics. The results of these experiments are presented in Figures 17.3 and 17.4.

For each heuristic, we expect that solving the self-intersections is the bottleneck of the algorithm. The complexity of computing the self-intersection is $O(n \log n + k + k')$, where n is the number of vertices of the input, k is the number of pair of intersecting triangles and k' the number of triplet of intersecting triangles. In practice k' is often small,[66] thus we expect a near-linear behaviour of the running time in terms of the input size, defined as $n + 2k$ in the previous section.

We begin by comparing the running times on the 3 997 trivial models from the Thingi10K dataset, where the naive rounding method works. As shown in Fig. 17.3, the running times of the three heuristics are nearly equivalent for these models. On average, the naive rounding is 8% slower, while the heuristic of Zhou et al. is 26% slower than our approach.

We then compare the running times on the 527 non-trivial models of Thingi10K on which the naive rounding fails. We observe (see Fig. 17.4) that the running time of the three heuristics are similar on the models on which they terminate within the time limit.[67] Roughly, the running time is about one second per 10 000 vertices in the input size.[68] More precisely, the iterative naive heuristic is the slowest of the three heuristics and is on average about two times slower

[66] k' is small except on the six models with a ratio larger than one in Fig. 17.2.

[67] The power law interpolations of the running times (in microseconds) in terms of the input size are: $113x^{1.01}$ for the iterated naive heuristic, $97x^{0.99}$ for Zhou et al. heuristic and $21x^{1.09}$ for our heuristic.

[68] Recall that the input size actually counts not only the number of input vertices but also twice the number of pairs of intersecting triangles.

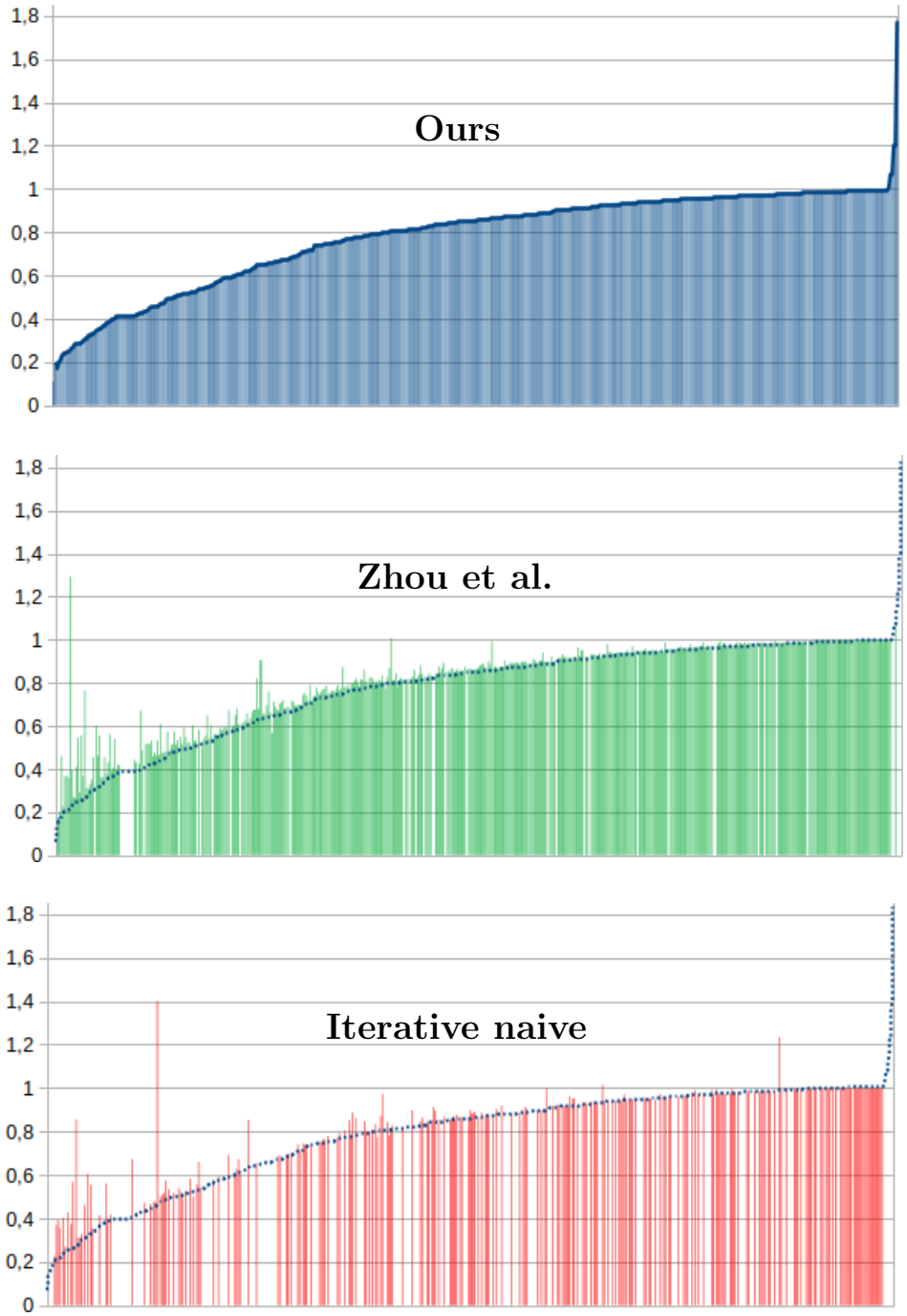


Figure 17.2: Output size versus input size. y-axis: ratio between the number of vertices in the output and the input size of the model (see Def. 34). x-axis: one pixel for each the 527 non-trivial models of Thingi10K ordered by the y-values in our heuristic. Blank columns correspond to models that did not terminate within time or iteration limit.

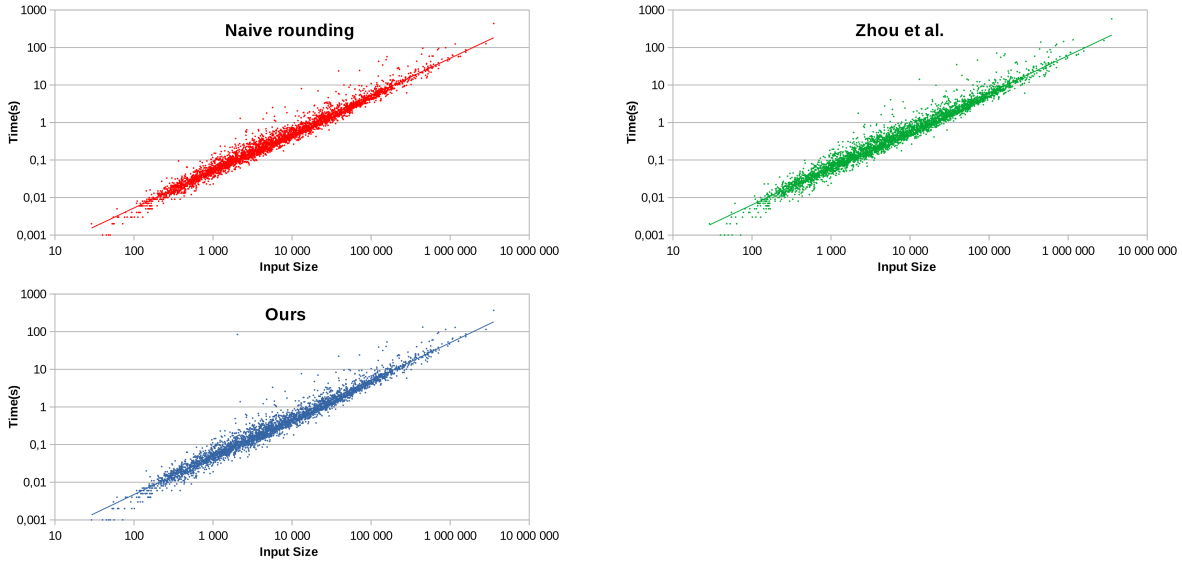


Figure 17.3: Running time in seconds for the 3 997 trivial models of Thingi10K depending on the size of the input (see Def. 34). Iterative naive results are drawn in red, Zhou et al. in green and ours in blue.

than ours on the 240 models on which it sucessfully terminates. Zhou et al.'s heuristic takes on average about 60% more time than ours on the 479 models on which it sucessfully terminates. Note however that the variance in these averages is large.

Our interpretation of these running-time behaviours is as follows. For all trivial and non-trivial models, unlike the naive and iterative naive heuristics, Zhou et al. and our heuristics tend to collapse some vertices together, which reduces the number of intersections and consequently the running time. Furthermore, our heuristic tends to collapse more vertices together than Zhou et al. heuristic, notably when one coordinate is close to zero. As a consequence, our heuristic tends to perform less iterations than that of Zhou et al., which performs less iterations than the iterative naive heuristic. Another more complex reason which might explain the worse running time of the naive heuristics, especially on the trivial models, is due to the rate of filter failures in the predicates: the filters in the predicates for testing triangle intersections are always performed using interval arithmetic on doubles and they presumably fail less often in Zhou et al. and our heuristics because the input coordinates of these triangles are rounded to floats, while they are doubles in the naive heuristics (see Section 14.2 for details).

Recall that these experiments are performed without parallelizations. While not done, such parallelizations would not be difficult.

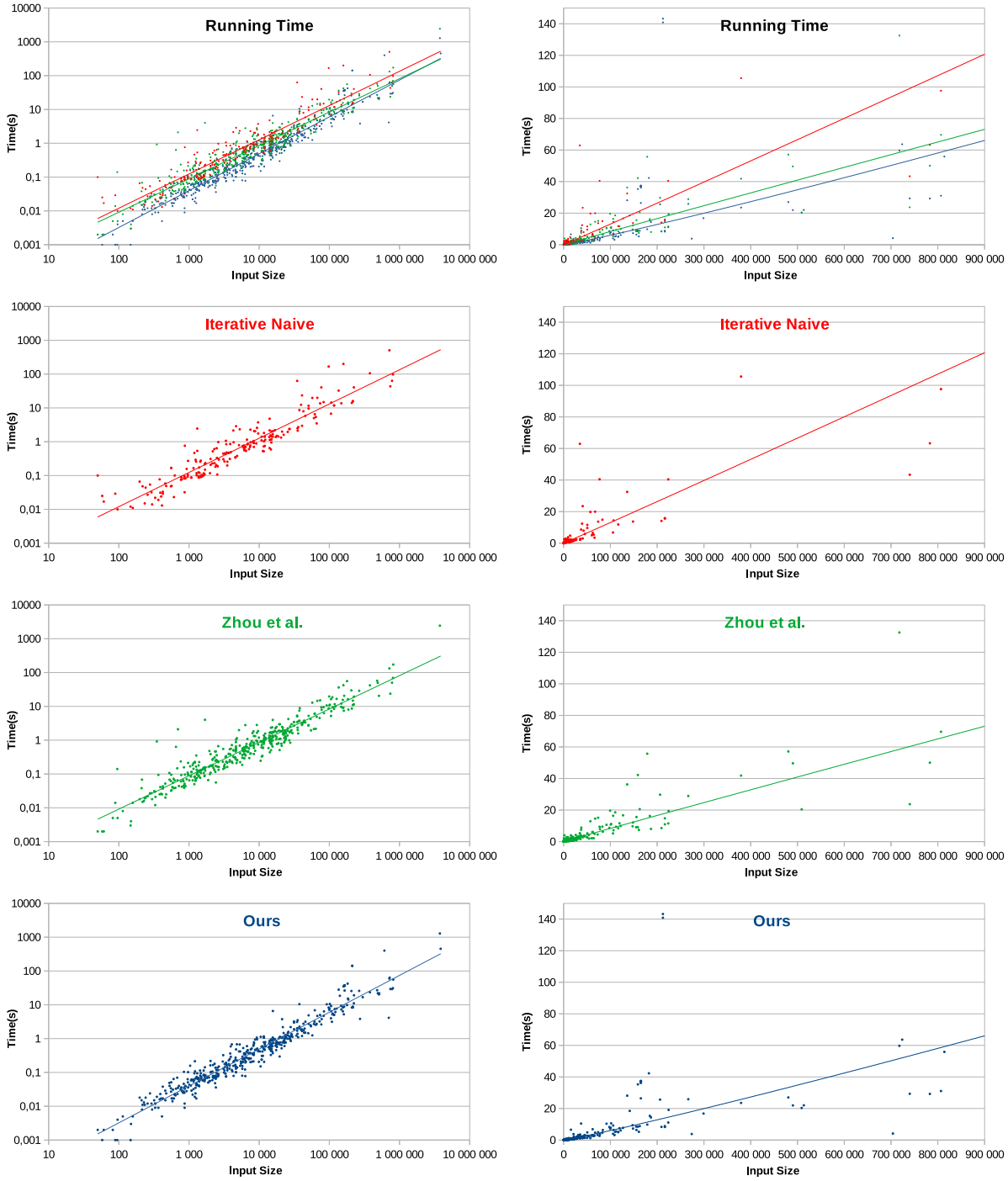


Figure 17.4: Running time in seconds for the 527 non-trivial models of Thingi10K depending on the size of the input (see Def. 34). The figures on the left use logarithmic axes, while those on the right use linear axes. Iterative naive results are drawn in red, Zhou et al. in green and ours in blue. Data are truncated on the linear axis for readability.

17.6 Our heuristic with different scalings

In our heuristic, by default, we scale the input coordinates such that the largest absolute coordinate lies in $[2^{22}, 2^{23})$. As mentioned before, this value is quite arbitrary. To observe the influence of the choice of the scaling, we conducted two similar experiments as in Section 17.3, this time performing a scaling such that the largest absolute value of each coordinate lies within respectively $[2^{15}, 2^{16})$ and $[2^{30}, 2^{31})$. The first is called the "coarse grid" experiment and the second is called the "fine grid" experiment.

All models successfully completed for the coarse grid and all except six successfully completed for the fine grid.[69] 19 models required two iterations with the coarse grid and 3 needed three iterations; these models are nearly the same as for the default grid. 93 models required two iterations with the fine grid, 11 needed three and 2 needed more than five. The running times and size of the output follow slightly the coarseness/fineness of the grid (see Table 17.2).

This experiment confirms our intuition that coarser is a grid, easier is the solving of intersections and the rounding. However a coarser grid also increases the distance between the input and the output. The computation time depends on the number of iterations and on the number of intersections after rounding the vertices, so a coarser grid often runs faster by reducing the number of iterations and the number of intersections.

17.7 Experiments on rotated cubes

To reach the limit of our heuristic, we experimented our algorithm on rotated cubes. At each step, we add a unit cube to the scene rotated by a random rotation in 3D.[70] Except at first step, the mesh is self-intersecting and we apply one rounding heuristic and go to the next step. We did this experiment for the iterative naive rounding, Zhou et al. and ours. The results are presented in Table 17.1.

As shown in Table 17.1, the number of vertices increases quickly at each step. All triangles are kept when refining the self-intersections leading to very dense models contained in the unit ball. Surprisingly, the iterative naive rounding and our heuristic with a coarse grid give equivalently the best results for this test. Concerning the result of iterated naive rounding, our interpretation

[69] These models are referenced as 44 874, 102 610, 101 633, 60 458, 101 632 and of course 996 816. Note that model 60 458 finishes with Zhou et al.

[70] We fixed the seed for reproductibility and to compare the three algorithms on the same input.

Iterative Rotated Cube Intersection and Rounding											
				It. naive		Zhou et al.		Ours		Ours(coarse)	
Step	**Input V**	**Input F**	**Nb Inter**	**It.**	**Time**	**It.**	**Time**	**It.**	**Time**	**It.**	**Time**
1	16	22	44	1	<1s	1	<1s	1	<1s	1	<1s
2	68	212	222	1	<1s	1	<1s	1	<1s	1	<1s
3	278	1 072	585	1	<1s	1	<1s	1	<1s	1	<1s
4	799	3 280	1 284	1	<1s	1	<1s	1	<1s	1	<1s
5	1 955	8 156	2 228	1	1s	1	1s	1	1s	1	1s
6	3 915	16 528	3 440	1	1s	1	1s	1	1s	1	1s
7	6 947	29 468	4 744	1	2s	1	2s	1	2s	1	2s
8	11 091	47 240	6 703	1	3s	1	3s	1	3s	1	3s
9	17 058	72 576	8 976	1	4s	1	4s	1	4s	1	4s
10	25 058	106 524	11 079	1	8s	1	7s	1	6s	1	6s
11	34 861	148 284	13 424	1	11s	1	8s	1	8s	2	14s
12	46 733	198 872	17 629	1	14s	1	12s	1	11s	2	19s
13	62 542	265 744	20 379	1	18s	1	15s	1	15s	2	24s
14	80 877	343 168	25 214	1	24s	1	19s	1	19s	2	32s
15	103 731	439 300	29 804	1	29s	1	24s	1	24s	1	23s
16	130 759	552 960	30 844	1	36s	1	30s	1	30s	2	49s
17	158 311	669 748	35 022	1	44s	1	37s	1	37s	2	1m00
18	189 693	802 552	40 636	1	53	>4	>5h	2	1m15	2	1m13
19	226 105	956 644	45 235	1	1m03			2	1m30	2	1m27
20	266 632	1 128 164	54 580	1	1m15			2	1m47	2	1m43
21	316 196	1 336 448	67 846	1	1m30			2	2m08	2	2m03
22	378 658	1 597 076	60 397	1	1m43			2	2m29	2	2m23
23	432 723	1 825 996	66 279	1	1m58			2	2m54	2	2m46
24	492 046	2 077 196	75 441	1	2m17			2	3m19	2	3m09
25	560 035	2 364 052	78 924	3	5m53			6	9m50	3	5m00
26	630 907	2 663 648	93 310	2	4m58			22	39m26	2	4m09
27	715 501	3 019 460	111 742	3	5m43			10	10m02	3	6m40
28	817 914	3 447 772	104 265	3	6m23			2	5m33	2	5m20
29	912 189	3 844 852	117 035	3	7m19			>4	>5h	3	8m19
30	1 018 408	4 291 364	126 731	3	8m10					2	6m47
31	1 133 358	4 774 728	140 784	>4	>5h					>4	>5h

Table 17.1: The number of intersections is the number of pairs of properly intersecting triangles at the first iteration of the iterative naive method. The size of the model and the number of intersections are those of the iterative naive rounding method. Those values vary slightly in other methods.

is that the mesh has so many faces in the unit ball, that when rounding the relevant vertices, the other heuristics remove few intersections by collapsing vertices while they create many new edge-face intersections induced by vertices that traverse faces (in the rounding).

Our heuristic is always better than the (non-iterative) naive rounding and Zhou et al., and our heuristic with a coarse grid gives quite equivalent results than that of the iterated naive rounding. This test reaches the limit of our approach by producing extremely dense and difficult models. While this test is quite effective in showing the limits of the heuristics, it is not really representative of real world meshes.

17.8 Validation of some heuristic designs

17.8.1 Naive rounding on floats and on the integer grid

To support our choice and that of Zhou et al. (see Section 12.2) to round respectively on floats or integers only some well-chosen vertices, we conducted two similar experiments as in Section 17.1. However, this time, we rounded the coordinates of all the vertices to single-precision floating-point numbers instead of double precision in the first experiment and to the closest integer in the second, up to a scaling such that the largest absolute value of each coordinate lies within $[2^{22}, 2^{23})$.

In the first experiment, we rounded to floats all the coordinates of the 4 524 self-intersecting models from the Thingi10K dataset. Then, after resolving the self-intersections, naive rounding to floats failed on 1 894 of these models ($\approx 42\%$). We then performed iterative naive rounding with the limit of 20 iterations within a one-hour limit, 1 088 models ($\approx 24\%$) still exhibited self-intersections.

For the second experiment, we scaled and rounded to integers all the coordinates of the 4 524 self-intersecting models from the Thingi10K dataset. Then, after resolving the self-intersections, naive rounding to integers failed on 1 885 of these models ($\approx 42\%$). Similarly, we performed iterative naive rounding with the same iteration and time limits, 1 138 models ($\approx 25\%$) still exhibited self-intersections.

As with iterative naive rounding on doubles, allowing more than 20 iterations seemed useless. Only 138 models for the first experiment and 149 for the second were successfully completed after more than five iterations, and only 44 and 42 models, respectively, were resolved after more than ten iterations. Additionally, for both experiments, more than 90% of the unresolved models

exhibited more intersections at the final iteration than at the start, likely meaning they would not finish for any reasonable number of iterations.

In conclusion, rounding all input coordinates to floats or integers, then applying naive rounding iterated or not, shows to be highly ineffective for resolving self-intersections and rounding. About half of the self-intersecting models in Thingi10K still exhibit self-intersections after naive rounding, and about a quarter remain unresolved after iterative naive rounding.

17.8.2 Our heuristic without rounding all vertices in "hot" voxels

Recall that in our heuristic, we round the vertices of any two triangles that properly intersect; these vertices are contained in some voxels and we also round all the vertices that lie these voxels. To validate this choice, we conducted an experiment where we do not round all the vertices in these voxels.

In this setting, on the 527 non-trivial models, 58 models required two iterations to terminate, 10 required three and two models failed to complete within the limit of 20 iterations and one hour; recall that it is respectively 26, 3 and one for the default setting. One of the failed models is unsurprisingly the difficult model discussed in Section 17.3. In contrast, the other model is relatively small, with 14 769 vertices and 75 pairs of intersecting triangles.[71] It failed because the number of intersections increased significantly through the iterations, a pattern similar to what we observed in experiments with other heuristics. Note that this variant also failed with a similar pattern on one "trivial" model.[72]

In conclusion, while this variant remains significantly more effective that the one of Zhou et al., the results are not as good as in our default setting. Two more models failed (which were apparently simple) and many models (46) required more iterations, which resulted in slower running times. This motivates the importance of rounding all vertices in the "hot" voxels in our heuristic.

17.9 Conclusions

Some of our experiments are summarised in Table 17.2. In conclusion, the naive rounding solves approximativelly 90% of the 4 524 models of Thingi10K that contain self-intersections. Iterating

[71] This model is referenced as 105 867.
[72] This model is referenced as 225 354.

the naive rounding resolves about half of these 527 unsolved models. The heuristic by Zhou et al. [31] successfully solves about 90% of the models that the naive approach fails on, while our heuristic solves all the models in the Thingi10K dataset although in one instance, we have to consider a coarser approximation than the default one. In terms of running time, all heuristics perform similarly, when successful, with slighlty better performences for ours.

Heuristics	**Iterative naive**	**Zhou et al.**	**Ours**	**Ours (coarse)**	**Ours (fine)**
Non-trivial models of Thingi10K	527	527	527	527	527
Unsolved models	287	48	1	0	6
Models solved with more than one iterations	527	368	29	22	106
Max. number of iterations (< 20)	18	17	3	3	8
Average output vertices compared to ours	1.08	1.06	1 (Ref)	0.99	1.02
Average running time compared to ours	2.16	1.63	1 (Ref)	0.95	1.08
Max. distance between input and output	$k\,2^{-52}$	$k\,2^{-23}$	$k\,2^{-23}$	$k\,2^{-16}$	$k\,2^{-30}$

Table 17.2: Performences on the 527 non-trivial models of Thingi10K on which the naive rounding fails. The maximum distance between the input and the output depends on the number of iteration k performed by the heuristic.

Part VI

Conclusion

Chapter 18

Conclusion and perspectives

18.1 Conclusion

We presented the first local and practival 3D snap rounding algorithm, which improves upon the work by Devillers et al. [8]. Their algorithm was the first and lone 3D snap rounding algorithm but had a significant drawback: it subdivided the entire model, even in areas where no intersections occurred during the rounding, leading to an excessive computational burden and making their algorithm unusable even for small instances. Our algorithm presented here takes a local approach, by recursively identifying and subdividing only the critical faces that intersect during the rounding process, along with their adjacent faces. This drastically reduces the number of subdivisions and makes the algorithm far more practical.

While this local algorithm is a significant advancement, it still struggles when the number of critical faces increases, leading to too many subdivisions. To address this issue, we developed two optimizations. The first one locally subdivides pair of edges and faces that intersect during the rounding, regardless of the rest of the model. Although simple in concept, the subdivision scheme needs to be carefully designed to ensure that it indeed removes the intersections between the considered pairs of features. However, many meshes exhibit numerous pair of faces that become coplanar and intersect when rounding, which induces difficulties with this local optimization. The second optimization targets this issue specifically. These optimizations significantly reduce the number of intersections, which, in turn, reduces the number of critical faces in the main algorithm.

We implemented this algorithm and its optimizations. The implementation proved to be

particularly challenging due to the numerous degeneracies; indeed, while some degeneracies are often present in the input models, the subdivisions performed by our algorithm tend to always create many degeneracies. Currently the implementation is still not mature. However, we succeded to round the significant portion of 94% of the non-straightforward self-intersecting models from the Thingi10K dataset. This implementation serves as a proof of concept, demonstrating that our algorithm is practical and capable of handling self-intersections while ensuring topological guarantees.

In the absence of a universally applicable algorithm, we also presented a highly efficient heuristic for mesh rounding, inspired by the work of Zhou et al. [31]. This heuristic handles self-intersecting models by rounding some well-chosen vertices on some uniform grid before subdividing the triangles along their self-intersections and then rounding the newly created vertices. The process is repeated until the model becomes intersection-free or at most a fixed number of times. When topological guarantees are not needed, this approach is particularly effective for handling models. When tested on the Thingi10K dataset, this heuristic successfully produces intersection-free simplicial complexes for all self-intersecting models, outperforming the previous state-of-the-art method, which solved only about 90% of the non-trivial models. A significant example is the notoriously difficult model[73] of Thingi10k (see Figure 17.1), which consists of about 170 000 triangles and contains about 5 millions pairs of intersecting triangles, which we succeed to round in about 3 minutes with an appropriate scaling. It should be stressed that on this input model, the number of intersections is so large that we do not even succeed to compute the exact model subdivided along the self-intersections. Although this heuristic lacks formal guarantees, we believe that its simplicity and efficiency will establish it as a new standard for repairing self-intersecting models, when topological guarantees are not needed.

18.2 Perspectives

Several research directions arise from the contributions presented in this thesis, which we categorize in three main axes.

The first axis is the further development and optimization of our current algorithm. From a theoretical perspective, the worst-case complexity of our algorithm is not tight and empirical

[73] Model referenced as 996 816.

evidence suggests that its complexity is near-linear in practice. An objective is to prove that the algorithm's complexity is bounded by a near-linear function in the input size and polynomial in specific input parameters, such as the number of edges in the model's silhouette or the number of close non-incident pairs of faces. The goal would be to show that, if these parameters remain small (i.e., if the model is relatively clean), the algorithm's complexity is near-linear.

On the practical side, our implementation is not mature, and there are several avenues for improvement. As highlighted in the thesis, the first two steps of the algorithm (see Chapter 6) tend to oversubdivide faces, particularly when they are almost parallel to the xy-plane. A more refined approach could to be to apply these steps only to carefully chosen faces, or to explore more local or lazy versions of the algorithm that remove unnecessary subdivisions. Furthermore, the degeneracies introduced by our algorithm pose a significant burden to the implementation. Developing techniques that reduce or eliminate these degeneracies is another potential area of research. Another direction of interest would be to round to the integer grid only the vertices of features involved in proper intersections, avoiding the need to round the entire model when unnecessary. Overall, we believe that all the key components for a highly efficient algorithm capable of rounding all models in the Thingi10K dataset are reached and that the task ahead is to assemble them effectively.

The second axis is the further development of the heuristic presented in the thesis. A key objective is to integrate this heuristic into the CGAL library and optimize it, notably through parallelization. Our experiments show that the heuristic not only removes self-intersections but also drastically reduces the number of vertices and faces when edges are extremely short and lie entirely in one voxel of the considered uniform grid, which depends on the scaling factor used. Automatizing the selection of an appropriate scaling factor to clean the model while maintaining geometric fidelity is another area for exploration. While the heuristic effectively removes self-intersections, it does not address issues such as open models or non-manifold edges and vertices. These problems already represent active areas of research, and combining these efforts with our heuristic could lead to a comprehensive pipeline for mesh repair.

The third axis is to extend snap rounding beyond 3D meshes. A natural question arises: how can snap rounding be generalized to higher dimensions? This involves working with segments, triangles, or higher-dimensional simplices, where the main difficulty is to prevent intersections during the rounding of one coordinate without breaking the operations performed on others.

To illustrate the complexity of these questions, it is worth noting that the only paper on the subject by Milenkovic [23] in 1990 was later found to be flawed. Another natural direction is the rounding of 3D volumetric meshes, which could be achieved by rounding the triangles of tetrahedra and subsequently re-tetrahedralizing. However, re-tetrahedralization is a complex process with open questions. Snap rounding could also be explored in non-Euclidean spaces, such as hyperbolic or projective planes, or applied to models with curved edges like circular arcs. These areas remain largely unexplored, though numerical instabilities in these geometries also demand attention. Additionally, alternative rounding strategies, such as collapsing tiny features instead of rounding to the nearest integer, could be explored. While naive collapsing might introduce new intersections, developing algorithms that handle these issues would present difficulties similar to those found in snap rounding.

Chapter 19

Résumé en francais

19.1 Introduction

En informatique, les algorithmes géométriques sont principalement décrits à l'aide du modèle Réel RAM [3]. Dans ce modèle, chaque unité de mémoire peut stocker un nombre réel, et l'accès à une unité de mémoire s'effectue en temps constant, indépendamment de son emplacement. Toutes les opérations standard (addition, soustraction, comparaison, etc.) sont également réalisées en temps constant avec des résultats exacts. Ce modèle est pratique pour décrire les algorithmes sans se soucier des limitations de précision numérique inhérentes aux ordinateurs réels. Toutefois, le modèle RAM réel est purement théorique, et toute implémentation réelle doit représenter les objets géométriques avec un nombre fini de bits, entraînant ainsi une précision finie.

De nombreuses implémentations manipulant des objets polygonaux en 3D, tant dans le milieu académique qu'industriel, exigent que les polygones en entrée soient disjoints deux à deux, avec des coordonnées de sommets exprimées en précision fixe (généralement 32 ou 64 bits). Lorsque des algorithmes géométriques génèrent de nouveaux objets à partir de ces entrées, leur représentation exacte requiert souvent plus de bits que l'entrée d'origine pour une représentation précise. Par exemple, l'intersection de deux segments de droite dont les extrémités ont des coordonnées entières sur b bits produit en général un point d'intersection dont les coordonnées rationnelles exactes nécessitent $5b + O(1)$ bits (voir Section 2.2). De même, l'intersection de trois triangles en 3D génère des coordonnées rationnelles nécessitant en général $13b + O(1)$ bits (voir Section 3.2.2). L'application d'une rotation à un polyèdre introduit de nouveaux sommets aux coordonnées impliquant des fonctions trigonométriques, tandis que l'échantillonnage de surfaces algébriques

génère des sommets comme solutions de systèmes algébriques, qui peuvent être arbitrairement proches les uns des autres (en fonction du degré de la surface). Cet écart entre la précision des entrées et celle des sorties est problématique, en particulier dans l'industrie, car il empêche souvent l'utilisation directe de la sortie d'un algorithme comme entrée pour un algorithme suivant. Par conséquent, l'arrondi des structures polygonales 3D entre les opérations est crucial, faisant de cette problématique un enjeu fondamental en géométrie algorithmique et en infographie.

La plupart des implémentations d'algorithmes géométriques arrondissent leur sortie à la même précision que l'entrée, en utilisant généralement des nombres flottants en simple ou double précision. Dans ces implémentations, l'arrondi est souvent réalisé directement par l'unité de calcul en virgule flottante du processeur, selon la norme IEEE 754 [1]. Dans cette norme, les résultats sont arrondis vers l'un des nombres flottants les plus proches après chaque opération.[74]

Cependant, lorsque l'arrondi est effectué selon la norme IEEE ou d'autres méthodes dites « naïves », les propriétés topologiques des objets géométriques peuvent être altérées. Par exemple, comme illustré en Figure 1.1, un triangle et un quadrilatère peuvent devenir intersectants après une rotation suivie d'un arrondi, alors qu'ils étaient disjoints à l'origine. De telles erreurs sont encore plus marquées en 3D, où l'arrondi peut introduire des auto-intersections au sein des structures polygonales, les rendant inutilisables pour de nombreux algorithmes. De nombreuses études en infographie [2, 5, 6, 9, 19, 20, 31] ont mis en évidence les défis majeurs posés par l'arrondi, soulignant son importance dans le développement d'algorithmes et de logiciels. L'arrondi sans intersection des structures polygonales 3D est donc un problème fondamental en géométrie algorithmique, en infographie et dans l'industrie.

Certaines approches utilisent l'arithmétique exacte dans les calculs géométriques pour éviter ces problèmes, comme celles de Pion et Fabri [27], implémentées dans des bibliothèques telles que la Computational Geometry Algorithms Library (CGAL) [28]. Néanmoins, l'arrondi reste souvent nécessaire en raison de la croissance exponentielle de la taille des représentations et de considérations pratiques comme le stockage et la diffusion des résultats. En effet, dans des constructions géométriques en cascade, où la sortie d'un algorithme est utilisée comme entrée d'un autre, la précision requise pour une représentation exacte peut rapidement devenir

[74] La norme IEEE 754 définit cinq modes d'arrondi : vers la valeur inférieure (floor"), vers la valeur supérieure (ceil"), vers la valeur inférieure ou supérieure la plus proche de zéro (trunc"), et deux autres modes pour l'arrondi au plus proche, distingués par leur gestion des cas d'égalité : arrondi vers la valeur dont le dernier chiffre est nul (ties to even") et arrondi vers la valeur la plus éloignée de zéro ("ties to away").

ingérable. Par exemple, enchaîner k intersections de segments nécessiterait $5^k(b + O(1))$ bits pour représenter la sortie finale avec précision. De plus, certaines opérations, comme les fonctions trigonométriques ou les racines, ne peuvent pas être représentées exactement avec des nombres rationnels, nécessitant des extensions complexes pour une description fidèle. Ainsi, le nombre de constructions en cascade réalisables sans arrondi est limité et, la plupart des algorithmes prenant en entrée des coordonnées flottantes en précision fixe, une sortie exacte non arrondie est rarement réutilisable comme entrée dans d'autres algorithmes.

En réponse à cela, le développement d'algorithmes permettant d'arrondir les coordonnées des points géométriques tout en préservant la topologie de la sortie a fait l'objet de nombreuses recherches. Ces algorithmes doivent garantir que la sortie reste proche de l'entrée, avec une définition adéquate de proximité. Par exemple, le résultat arrondi ne doit pas nécessairement conserver la plus haute précision, mais doit refléter fidèlement la structure combinatoire du résultat original, bien que sous une précision plus faible. Ce problème, connu sous le nom de problème de Snap Rounding Problem, consiste à arrondir des entrées géométriques (souvent des sommets de segments ou de triangles) tout en maintenant certaines propriétés topologiques. Le problème en deux dimensions a été largement étudié depuis la fin des années 1980 [7,13–16,18,22], aboutissant à un algorithme de Snap rounding stable en 2D proposé par Hershberger en 2013 [17].

Cependant, malgré l'importance du problème, les résultats en 3D sont rares. Dans les années 1990, deux algorithmes ont été proposés [12, 23], mais tous deux ont été prouvés erronés par Fortune [11], qui a introduit deux algorithmes alternatifs [10, 11] ne résolvant toutefois pas réellement le problème (voir Section 3.3). Milenkovic et Sacks [25] ont présenté un algorithme en 2019, mais sans garantie de proximité entre l'entrée et la sortie. Ce n'est qu'en 2020 que Devillers et al. [8] ont proposé une solution théorique au problème de Snap rounding en 3D, bien que sa complexité ne le rende pas utilisable en pratique (voir Section3.3).

Faute de solutions théoriques satisfaisantes, des chercheurs en infographie ont développé des heuristiques pour résoudre le problème. L'état de l'art actuel est celui de Zhou et al. [31], qui traite avec succès la plupart des problèmes d'arrondi, mais sans garantie de terminaison ni de préservation topologique (voir Section3.3).

S'appuyant sur les travaux de Devillers et al. [8], cette thèse présente le premier algorithme pratique et certifié pour résoudre le problème de Snap rounding en 3D. La première partie de cette thèse passe en revue l'état de l'art du Snap rounding en 2D et en 3D et met en avant les

difficultés associées. La seconde partie détaille notre algorithme de Snap rounding en 3D, suivie d'une description de son implémentation dans la troisième partie. La quatrième partie présente les résultats expérimentaux. Enfin, la cinquième partie propose une nouvelle heuristique, non certifiée mais très efficace, pour l'arrondi des sommets.

19.2 Contributions

Dans ce chapitre, nous résumons nos contributions. La première section présente un algorithme de snap rounding 3D local certifié et son implémentation. La deuxième section présente des optimisations de cet algorithme, que nous pensons d'un intérêt indépendant. La troisième section décrit une heuristique non certifiée mais très efficace pour l'arrondi des coordonnées des sommets des maillages. La dernière section présente une contribution mineure indépendante pour le snap rounding 2D.

19.2.1 Un algorithme de snap rounding 3D local et son implémentation

Dans la Partie II, nous présentons le premier algorithme de snap rounding 3D local. Rappelons que Devillers et al. [8] ont présenté en 2020 le premier et unique algorithme de snap rounding 3D sur une grille uniforme. Cependant, un inconvénient majeur de leur algorithme est qu'il n'est pas local : il subdivise l'ensemble du modèle, quelle que soit la localisation des intersections potentielles lors de l'arrondi. Cela conduit à de nombreuses subdivisions même pour des modèles simples, rendant l'algorithme inefficace en pratique [29].

En nous appuyant sur leurs travaux, nous avons développé un algorithme de snap rounding 3D local. Notre algorithme simule le processus d'arrondi et marque comme "critiques" toutes les paires de faces qui s'intersectent proprement pendant cette simulation. L'algorithme subdivise ensuite uniquement les faces critiques et celles qui leur sont adjacentes, réduisant ainsi considérablement le nombre de subdivisions.

Cet algorithme est le premier qui parvient, en pratique, à arrondir directement toutes les faces sans provoquer d'intersection lors de l'arrondi et à ne subdiviser que les faces proches des zones critiques. Toutefois, bien que les subdivisions restent local, elles restent aussi nombreuses. Nous avons donc développé des optimisations pour les réduire autant que possible, comme discuté dans la section suivante.

Avec ces optimisations, notre algorithme est le premier algorithme de snap rounding 3D fonctionnel sur des cas non triviaux. Parmi les 4 524 modèles contenant des auto-intersections dans Thingi10K, environ 90% sont résolus en arrondissant naivement les sommets. Pour les 520 modèles non triviaux restants, comportant des intersections propres durant l'arrondi, notre implémentation réussit à en résoudre environ 95%, échouant sur 33 instances.

19.2.2 Optimisations pour l'algorithme de snapping 3D

D'après nos expérimentations (voir Section 16.7), bien que notre algorithme soit beaucoup plus pratique que celui de Devillers et al. [8], il reste inefficace sur des cas complexes. En effet, si le nombre de faces critiques augmente, le nombre de subdivisions devient également très important. Nous avons donc développé trois optimisations pour notre algorithme (voir Chapitre 8).

Nous pensons que notre première optimisation, qui subdivise localement les arêtes et les faces, est d'intérêt indépendant pour le snapping 3D. La deuxième optimisation, qui traite les faces presque coplanaires, revisite des briques algorithmiques déjà connues mais apporte une nouveauté dans un contexte d'optimisation. La troisième est une version paresseuse de snapping 2D, principalement intéressante d'un point de vue pratique.

La première optimisation subdivise localement les arêtes et les faces qui s'intersectent proprement lors de la simulation d'arrondi. L'idée intuitive est très simple : si un sommet intersecte un triangle pendant la simulation, nous subdivisons le triangle pour éviter cette intersection, et de même pour deux arêtes qui se croisent. Cette idée naïve est cependant complexe à implémenter. Nous avons con$\mathcal{C}$cu un schéma de subdivision garantissant que deux arêtes subdivisées selon cette méthode ne s'intersecteront plus lors de l'arrondi, et de même pour un sommet et une face. Cette approche reste heuristique, car bien que deux arêtes subdivisées ainsi ne s'intersectent plus entre elles, elles peuvent encore croiser d'autres arêtes de la scène.

La deuxième optimisation cible les faces presque coplanaires. Nos expérimentations montrent que de nombreux maillages présentent un grand nombre d'intersections coplanaires pendant la simulation. Nous avons donc con$\mathcal{C}$cu une optimisation spécifique : étant donné un ensemble de triangles devenant coplanaires pendant l'arrondi, nous les projetons sur ce plan, appliquons un snapping 2D (voir Section 2.6), puis réinjectons les subdivisions sur les triangles 3D d'origine. Nous utilisons ensuite ces subdivisions comme entrée pour notre algorithme principal.

La troisième optimisation est une variante paresseuse du snapping 2D. Notre algorithme utilise

un snapping 2D modifié qui insère des frontières de pixels chauds pour éviter les intersections pendant l'arrondi. Cette approche multiplie systématiquement le nombre de triangles par un facteur constant. Nous avons con$\mathcal{C}$cu une version paresseuse qui n'ajoute ces pixels chauds que lorsque c'est nécessaire.

Il n'y a aucune garantie que ces subdivisions résolvent tous les problèmes d'arrondi, mais elles sont con$\mathcal{C}$cues pour gérer les cas simples, tandis que notre algorithme certifié traite les cas complexes. Notons que ces optimisations préservent toutes les garanties topologiques et de distance de l'algorithme principal, contrairement à l'heuristique présentée en Section 4.3, car elles ne font que subdiviser les faces d'entrée.

19.2.3 Une heuristique non certifiée mais extrêmement efficace

En l'absence d'un algorithme certifié efficace fonctionnant dans tous les cas, nous présentons dans la Partie III une nouvelle heuristique très performante. Cette heuristique s'appuie sur celle de Zhou et al. [31]. Étant donné une soupe de triangles auto-intersectante avec des coordonnées de sommets représentées en virgule flottante double précision, notre heuristique produit un complexe simplicial sans intersection, avec des coordonnées de sommets également représentées en double précision.

L'heuristique fonctionne en mettant d'abord à l'échelle la scène pour qu'elle soit aussi grande que possible tout en gardant la partie entière des coordonnées représentable en flottant. Ensuite, les sommets des paires de triangles qui s'intersectent sont arrondis sur la grille entière. Nous arrondissons également sur la grille les sommets situés dans les voxels contenant des sommets arrondis. Les auto-intersections sont ensuite supprimées en subdivisant les triangles en conséquence, et les nouveaux sommets créés sont arrondis en double précision. Ce processus est répété un nombre maximal de fois fixé tant que le modèle contient des auto-intersections, avant un retour à l'échelle initiale. La distance entre un sommet arrondi et sa position d'origine est bornée par le nombre d'itérations multiplié par une constante dépendant de l'échelle utilisée.

Nous avons testé notre heuristique sur les modèles auto-intersectants de l'ensemble de données Thingi10K, et nous sommes les premiers à produire des complexes simpliciaux sans intersection pour tous ces modèles. Cette heuristique améliore les résultats de l'état de l'art obtenus par Zhou et al. [31], qui ont résolu avec succès environ 99% des modèles auto-intersectants de Thingi10K, mais seulement environ 90% des cas non triviaux (voir Section 17.2).

Cette heuristique est remarquablement simple et efficace, et nous pensons qu'elle deviendra une référence pour supprimer les auto-intersections des modèles lorsque des garanties topologiques ne sont pas nécessaires.

19.2.4 Un algorithme d'arrondi en 2D avec nombres flottants

Nous présentons également le premier algorithme d'arrondi 2D qui, étant donné un ensemble de segments dans le plan, arrondit les sommets au centre de leur "pixel" lorsque la grille sous-jacente n'est pas uniforme. Cela permet d'arrondir directement aux nombres flottants plutôt que sur une grille uniforme, comme le font les algorithmes existants.

Étant donné un ensemble de segments en entrée, l'algorithme produit un complexe simplicial avec des coordonnées représentées en nombres flottants (de toute précision), en garantissant que la distance de Hausdorff entre un sommet de sortie et son point correspondant sur l'entrée est au plus une unité de la moindre précision (ulp). En d'autres termes, l'arrondi est fait avec la précision maximale permise par le format flottant, au lieu de se limiter à une grille uniforme. Notre algorithme préserve également la topologie "à effondrement près".

Concrètement, l'algorithme subdivise les segments en utilisant des bandes verticales espacées entre deux nombres flottants consécutifs, chaque bande contenant un sommet de l'arrangement (une bande ayant une largeur d'une ulp). La complexité en sortie est de $O(n(n+k))$ sommets, où n est le nombre de sommets en entrée et k le nombre d'intersections des segments d'origine. Bien que simple, cet algorithme est une contribution indépendante et mineure de cette thèse, sans lien direct avec l'arrondi en 3D.

19.3 Conclusion

Nous avons présenté le premier algorithme de 3D snap rounding local et pratique, améliorant les travaux de Devillers et al. [8]. Leur algorithme, bien qu'innovant, présentait une limite majeure : il subdivisait l'ensemble du modèle, y compris dans les zones sans intersections, ce qui engendrait un coût computationnel excessif et rendait l'algorithme inutilisable pour des instances même de petite taille. Notre algorithme adopte une approche locale en identifiant et subdivisant récursivement uniquement les faces critiques qui s'intersectent lors de l'arrondi, ainsi que leurs faces adjacentes. Cette stratégie réduit drastiquement le nombre de subdivisions,

rendant l'algorithme bien plus pratique.

Bien que cette approche locale constitue une avancée majeure, elle reste limitée lorsque le nombre de faces critiques devient trop élevé, ce qui engendre encore trop de subdivisions. Pour pallier cette limitation, nous avons développé deux optimisations. La première subdivise localement les paires d'arêtes et de faces qui s'intersectent lors de l'arrondi, indépendamment du reste du modèle. Toutefois, bien que conceptuellement simple, ce schéma de subdivision doit être soigneusement con$\mathcal{C}$cu pour éliminer les intersections des paires de features concernées. Cependant, de nombreux maillages présentent des paires de faces qui deviennent coplanaires et s'intersectent lors de l'arrondi, ce qui complique cette optimisation locale. La deuxième optimisation vise précisément à résoudre ce problème. Ces optimisations réduisent significativement le nombre d'intersections et, par conséquent, le nombre de faces critiques dans l'algorithme principal.

Nous avons implémenté cet algorithme et ses optimisations, mais l'implémentation reste pas ecnore complètement mature. Néanmoins, nous avons réussi à arrondir 94 % des modèles auto-intersectants non triviaux du jeu de données Thingi10K, démontrant ainsi que notre approche est fonctionnelle et capable de gérer les auto-intersections de modèles compliqués tout en garantissant des propriétés topologiques.

En l'absence d'un algorithme universel, nous avons également présenté une heuristique de maillage inspirée des travaux de Zhou et al. [31]. Cette heuristique arrondit certains sommets sur une grille uniforme, subdivise les triangles le long des auto-intersections, puis réitère le processus jusqu'à l'obtention d'un modèle sans intersection ou jusqu'à un nombre d'itérations maximal. Bien qu'elle ne fournisse pas de garanties topologiques, cette approche est très efficace en pratique. Sur Thingi10K, elle produit des complexes simpliciaux sans intersection pour tous les modèles auto-intersectants, surpassant la méthode de l'état de l'art précédent qui ne résolvait qu'environ 90 % des modèles non triviaux.

Un exemple significatif est le modèle 996 816 de Thingi10K (voir Figure 17.1), qui comporte environ 170 000 triangles et près de 5 millions de paires de triangles intersectants. Nous avons réussi à l'arrondir en environ 3 minutes avec un facteur d'échelle adapté, alors même que le nombre d'intersections est si élevé que le modèle exact subdivisé reste incalculable par une machine standart.

19.4 Perspectives

Nos contributions ouvrent plusieurs axes de recherche, que nous classons en trois grandes catégories.

Le premier axe concerne le perfectionnement de notre algorithme. Théoriquement, la complexité dans le pire cas n'est pas précisément bornée, mais les observations empiriques suggèrent une complexité quasi-linéaire. Un objectif serait de démontrer que cette complexité est bornée par une fonction quasi-linéaire de la taille de l'entrée et polynomiale en des paramètres spécifiques, comme le nombre d'arêtes de la silhouette du modèle. Côté pratique, notre implémentation peut être affinée : les deux premières étapes de l'algorithme tendent à sur-subdiviser certaines faces quasi-parallèles au plan xy, ce qui pourrait être atténué par des subdivisions plus ciblées ou paresseuses.

Le deuxième axe porte sur le développement de l'heuristique. L'intégrer à CGAL et l'optimiser, notamment par parallélisation, est une voie prometteuse. Nos expériences montrent que l'heuristique réduit non seulement les auto-intersections mais aussi le nombre de sommets et de faces lorsque des arêtes très courtes se retrouvent dans une même cellule de la grille. Automatiser le choix du facteur d'échelle pour nettoyer le modèle tout en préservant la géométrie constitue un autre problème intéressant.

Le troisième axe est l'extension du snap rounding au-delà des maillages 3D. La question naturelle est celle de l'extension aux dimensions supérieures, où la principale difficulté réside dans la prévention des intersections lors de l'arrondi d'une coordonnée sans compromettre les autres. La seule publication sur ce sujet, due à Milenkovic en 1990 [23], s'est avérée erronée, soulignant la complexité de la problématique. Une autre direction serait d'appliquer l'arrondi à des maillages volumiques 3D, voire d'explorer des espaces non-euclidiens comme les plans hyperboliques ou projectifs. Enfin, des stratégies alternatives, comme le collapse des features infimes au lieu de leur arrondi strict, pourraient être explorées, malgré le risque d'introduire de nouvelles intersections.

Ces pistes, bien que complexes, montrent que le champ d'exploration est vaste et porteur d'avancées fondamentales et pratiques pour la géométrie algorithmique.

Bibliography

[1] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[2] Marco Attene. Indirect predicates for geometric constructions. *Computer-Aided Design*, 126:102856, 2020.

[3] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic geometry*. Cambridge university press, 1998.

[4] Franck Cappello, Frédéric Desprez, Michel Daydé, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *6th IEEE/ACM International Workshop on Grid Computing - GRID 2005*, Seattle, USA, United States, November 2005. Grid 2005 held in conjunction with SC'05, the International Conference for High Performance Computing, Networking and Storage. URL: `https://inria.hal.science/inria-00000284`.

[5] Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. Fast and robust mesh arrangements using floating-point arithmetic. *ACM Transactions on Graphics (TOG)*, 39(6):1–16, 2020.

[6] Gianmarco Cherchi, Fabio Pellacini, Marco Attene, and Marco Livesu. Interactive and robust mesh booleans. 41(6), nov 2022. `doi:10.1145/3550454.3555460`.

[7] Mark de Berg, Dan Halperin, and Mark Overmars. An intersection-sensitive algorithm for snap rounding. *Computational Geometry*, 36(3):159–165, 2007. `doi:10.1016/j.comgeo.2006.03.002`.

[8] Olivier Devillers, Sylvain Lazard, and William Lenhart. Rounding meshes in 3D. *Discrete and Computational Geometry*, 64(1):32–67, April 2020. URL: https://inria.hal.science/hal-02549290, doi:10.1007/s00454-020-00202-2.

[9] Lorenzo Diazzi and Marco Attene. Convex polyhedral meshing for robust solid modeling. *ACM Transactions on Graphics (TOG)*, 40(6):1–16, 2021.

[10] Steven Fortune. Polyhedral modelling with multiprecision integer arithmetic. *Computer-Aided Design*, 29(2):123 – 133, 1997. Solid Modelling. doi:doi.org/10.1016/S0010-4485(96)00041-3.

[11] Steven Fortune. Vertex-rounding a three-dimensional polyhedral subdivision. *Discrete & Computational Geometry*, 22(4):593–618, 1999. doi:10.1007/PL00009480.

[12] Michael T. Goodrich, Leonidas J. Guibas, John Hershberger, and Paul J. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 284–293. ACM, 1997. doi:10.1145/262839.262985.

[13] Daniel H Greene. Integer line segment intersection. Unpublished manuscript (cited by [17]).

[14] Daniel H. Greene and F. Frances Yao. Finite-resolution computational geometry. In *27th Annual Symposium on Foundations of Computer Science, 1986*, pages 143–152. IEEE, 1986. doi:10.1109/SFCS.1986.19.

[15] Leonidas J. Guibas and David H. Marimont. Rounding arrangements dynamically. *International Journal of Computational Geometry & Applications*, 8(02):157–178, 1998. doi:10.1142/S0218195998000096.

[16] Dan Halperin and Eli Packer. Iterated snap rounding. *Computational Geometry*, 23(2):209–225, 2002. URL: https://www.sciencedirect.com/science/article/pii/S0925772101000645, doi:https://doi.org/10.1016/S0925-7721(01)00064-5.

[17] John Hershberger. Stable snap rounding. *Computational Geometry*, 46(4):403–416, 2013. doi:10.1016/j.comgeo.2012.02.011.

[18] John D. Hobby. Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199–214, 1999. doi:10.1016/S0925-7721(99)00021-8.

[19] Bruno Lévy. Exact predicates, exact constructions and combinatorics for mesh csg. *arXiv preprint arXiv:2405.12949*, 2024.

[20] Marco Livesu. Advancing Front Surface Mapping. *Computer Graphics Forum*, 2024. doi:10.1111/cgf.15026.

[21] Maplesoft, a division of Waterloo Maple Inc.. *Maple*. Waterloo, Ontario. URL: https://hadoop.apache.org.

[22] Victor Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.

[23] Victor Milenkovic. Rounding face lattices in d dimensions. In *Proceedings of the 2nd Canadian Conference on Computational geometry*, pages 40–45, 1990.

[24] Victor Milenkovic and Lee R. Nackman. Finding compact coordinate representations for polygons and polyhedra. *IBM Journal of Research and Development*, 34(35):753–769, 1990.

[25] Victor Milenkovic and Elisha Sacks. Geometric rounding and feature separation in meshes. *Computer-Aided Design*, 108:12–18, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0010448518302896, doi:https://doi.org/10.1016/j.cad.2018.10.003.

[26] Eli Packer. Iterated snap rounding with bounded drift. *Computational Geometry*, 40(3):231–251, 2008. URL: https://www.sciencedirect.com/science/article/pii/S0925772107000922, doi:https://doi.org/10.1016/j.comgeo.2007.09.002.

[27] Sylvain Pion and Andreas Fabri. A generic lazy evaluation scheme for exact geometric computations. *Science of Computer Programming*, 76(4):307–323, 2011. Special issue on library-centric software design (LCSD 2006). URL: https://www.sciencedirect.com/science/article/pii/S016764231000167X, doi:https://doi.org/10.1016/j.scico.2010.09.003.

[28] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html.

[29] Leo Valque. 3D Snap Rounding. Master's thesis, Université de Lyon, June 2019. URL: https://hal.inria.fr/hal-02393625.

[30] Chee K. Yap Vikram Sharma. *Robust geometric computation*. Chapman and Hall/CRC, 3rd edition edition, 2017. URL: https://doi.org/10.1201/9781315119601.

[31] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)*, 35(4):1–15, 2016.

[32] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016.

Resumé

La plupart des algorithmes qui traitent des objets polygonaux en 3D utilisent des coordonnées où la précision est fixée pour les données d'entrée et de sortie. Cependant, les opérations géométriques produisent souvent des coordonnées de sortie qui nécessitent une précision supérieure à celle des données d'entrée. Cet écart implique la nécessité d'arrondir les nouvelles coordonnées pour qu'elles correspondent à la précision des données d'entrée, tout en préservant l'intégrité du modèle.

Le problème critique que nous abordons est la suppression des auto-intersections dans les modèles 3D, obtenue en subdivisant les faces le long de leurs intersections et en arrondissant les coordonnées résultantes, tout en veillant à ce que le modèle reste exempt d'auto-intersections. Ce problème est connu sous le nom de « snap rounding problem ».

Dans cette thèse, nous présentons le premier algorithme pratique et certifié de snap rounding en 3D, qui élimine avec succès les auto-intersections dans 94% des modèles présentants des auto-intersections de l'ensemble de données Thingi10K, et démontrant ainsi son effictivité dans les applications du monde réel. En outre, nous introduisons une heuristique non certifiée mais très efficace pour ce problème, qui surpasse l'état de l'art actuel en résolvant avec succès toutes les auto-intersections dans l'ensemble de données Thingi10K.

Abstract

Most algorithms for processing 3D polygonal objects use fixed-precision coordinates for both input and output data. However, geometric operations often produce output coordinates that require higher precision than the input. This discrepancy implies the need for rounding new coordinates to match the precision of the input, while preserving the integrity of the model.

The critical problem we address is the removal of self-intersections in 3D models, achieved by subdividing faces along their intersections and rounding the resulting coordinates, while ensuring that the model remains free from self-intersections. This problem is known as the snap rounding problem.

In this thesis, we present the first practical and certified local 3D snap rounding algorithm, which successfully eliminates self-intersections in 94% of the self-intersecting models in the Thingi10K dataset, demonstrating its effectiveness in real-world applications. Additionally, we introduce an uncertified yet highly efficient heuristic for this problem, which outperforms previous state-of-the-art methods by successfully resolving all self-intersections in the Thingi10K dataset.