

UC Berkeley

UC Berkeley Electronic Theses and Dissertations

Title

Geometric Algorithms for Cleanability in Manufacturing

Permalink

<https://escholarship.org/uc/item/03t6c02q>

Author

Yasui, Yusuke

Publication Date

2011

Peer reviewed|Thesis/dissertation

Geometric Algorithms for Cleanability in Manufacturing

by

Yusuke Yasui

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering – Mechanical Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sara McMains, Chair
Professor David Dornfeld
Professor Jonathan Shewchuk

Fall 2011

Geometric Algorithms for Cleanability in Manufacturing

Copyright 2011
by
Yusuke Yasui

Abstract

Geometric Algorithms for Cleanability in Manufacturing

by

Yusuke Yasui

Doctor of Philosophy in Engineering – Mechanical Engineering

University of California, Berkeley

Professor Sara McMains, Chair

This thesis describes geometric algorithms to check the *cleanability* of a design during the manufacturing process. The automotive industry needs a computational tool to determine how to clean their products due to the trend of miniaturization and increased geometric complexity of mechanical parts. A newly emerging concept in a product design, Design-for-Cleanability, necessitates algorithms to help designers to design parts that are easy to clean during the manufacturing process. In this thesis, we consider cleaning using high-pressure water jets to clean off the surfaces of workpieces. Specifically, we solve the following two problems purely from a geometric perspective: predicting water trap regions of a workpiece and finding a rotation axis to drain a workpiece.

Finding an orientation that minimizes the potential water trap regions and/or controls their locations when the workpiece is fixtured for water jet cleaning is important to increase the cleaning efficiency. Trapped water leads to stagnation areas, preventing efficient flow cleaning. Minimizing the potential water trap also reduces the draining time and effort after cleaning. We propose a new pool segmentation data structure and algorithm based on topological changes of 2D slices with respect to the gravity direction. Then, we can quickly predict potential water trap regions of a given geometry by analyzing our directed graph based on the segmented pools.

Given a workpiece filled with water after cleaning, to minimize the subsequent drying time, our industrial partner first mounts workpieces on a slowly rotating carrier so that gravity can drain out as much water as possible. We propose an algorithm to find a rotation axis that drains the workpiece when the rotation axis is set parallel to the ground and the workpiece is rotated around the axis. Observing that all water traps contain a concave vertex, we solve our problem by constructing and analyzing a directed “*draining graph*” whose nodes correspond to concave vertices of the geometry and whose edges are set according to the transition of trapped water when we rotate the workpiece around the given axis. We first introduce an algorithm to test whether a given rotation axis can drain the workpiece. We then extend these concepts to design an algorithm to find the set of all rotation axes that drain the workpiece. If such a rotation axis does not exist, our algorithm will also detect

that. To the best of our knowledge, our work is the first to tackle the draining problem and to give an algorithm for the problem.

To my family

Contents

Contents	ii
1 Introduction	1
1.1 Cleanability with high-pressure water jets	2
2 Related Work	6
3 Pool Segmentation for Predicting Water Trap Regions	9
3.1 Algorithm Overview	9
3.2 Pool Segmentation	12
3.3 Predicting Water Trap Regions	23
3.4 Results	25
3.5 Complexity Analysis	25
3.6 Conclusion	26
4 Testing a Rotation Axis to Drain a 3D Workpiece	28
4.1 Assumptions and a Key Observation	28
4.2 Approach and Theory	29
4.3 Graph Construction	35
4.4 Checking Drainability	43
4.5 Results	46
4.6 Complexity Analysis	53
4.7 Conclusion	53
5 Finding a Rotation Axis to Drain a 3D Workpiece	55
5.1 Algorithm Overview	56
5.2 Rotation axes for which a water particle is never trapped at concave vertex v	58
5.3 Constructing the extended draining graph	63
5.4 Rotation axes that drain a trapped water particle at concave vertex v	70
5.5 Finding rotation axes	72
5.6 Results	78
5.7 Complexity Analysis	81

5.8	Discussion	83
5.9	Conclusion	86
6	Conclusions and Future Work	87
6.1	Predicting water trap regions using pool segmentation	87
6.2	Finding a rotation axis to drain a 3D workpiece	89
A		92
A.1	Vertex classification implementation	92
A.2	Boundary cycles implementation	92
A.3	Finding boundary cycles generated, completed, and, updated	96
B		98
B.1	Boundary of $H_{i(xy)}$	98
B.2	Finding a closest point on a flat region	98
C		101
C.1	Constructing ∂T_v from edges e_i incident to v	101
C.2	Converting a great arc on the Gaussian Sphere to a region in the dual space	102
C.3	The characteristic of the gravity direction $\mathbf{d}(\hat{g})$ in the working plane	103
C.4	$W^+(\Pi(G))$ and $W^-(\Pi(G))$ in the dual space for great arc G	103
C.5	Determining the side of a T_v -arc where T_v lies in the working plane	106
	Bibliography	111

Acknowledgments

First and foremost, I must express appreciation to my research advisor, Professor Sara McMains. Her careful support made my stay in Berkeley fruitful and comfortable. From her, I learned not only engineering and academic research skills, but also the importance of communicating ideas through writing.

I would like to thank Professor Jonathan Shuwchuk and Professor David Dornfeld for serving on my dissertation committee. Professor Jonathan Shuwchuk gave me helpful advice whenever I encountered a problem. His advice from his extensive knowledge of computational geometry really helped me in solving problems. Professor David Dornfeld helped me begin research in the field of manufacturing even though I had very limited knowledge of it in my first year.

I also would like to thank Prof. Panayiotis Papadopoulos and Prof. Carlo Séquin for being on the committee of my Qualifying examination. They also kindly helped me during office hours when I faced difficulty in understanding course materials.

I thank the U.C. Discovery grant for sponsoring our project. I also thank Daimler AG for sponsoring our project and giving us interesting industrial problems. From Daimler AG, I wish to acknowledge Thomas Glau for the initial idea of tracking water by analyzing slice contour overlaps. I enjoyed working with him during his stay in our lab. We also would like thank Klaus A. Berger for kindly providing us real industrial models. They definitely expedited my research.

Finally, I would like to express my appreciation towards my labmates, Youngung Shon, Xiaorui Chen, Rahul Khardekar, Adarsh Krishnamurthy, Wei Li, Sushrut Pavanaskar, Peter Cottle, and, Youngwook Kwon for enriching my life in Berkeley. Discussion with them helped my research a lot and was an exciting experience as well.

Chapter 1

Introduction

Information technology has been meeting various needs in manufacturing such as shortening development time, reducing cost, and saving energy and resources. One example is computer-aided evaluation of design. It makes the conventional design-to-manufacturing cycle more efficient by identifying undesirable features of a design in a virtual world, thus reducing the required number of physical prototypes. Ultimately, we would like to establish an environment that does not require any physical prototypes at the design stages by fully utilizing information technology to help meet the market's demands faster and less expensively, in a sustainable manner.

Broadly speaking, the computer-aided evaluation of design is divided into two categories based on whether it is a functionality check or a manufacturability check. A functionality check predicts the performance of a design in a real-world environment. For example, estimation of stress and strain resulting from applied forces or determining thermal characteristics of a design using physics-based simulation fall into this category. By replacing stress testing such as a drop test performed in the real world with the corresponding simulation on a computer, we can avoid manufacturing many physical prototypes used only for such tests.

Manufacturability checks determine whether a design is appropriate for actual manufacturing from the point of view of fabrication, assembly, cleaning, etc. Even though the current design may satisfy the design requirements from an engineering and marketing perspective, if the design is too difficult or too costly to manufacture in practice, the design is not ideal. For example, inspecting a die taking into account spring-back in the bending of sheet metal, or inspecting the shape of a mold to determine whether the corresponding part can be easily extracted in casting, fall into this category. Traditionally, these sorts of manufacturability checks have been done based on skilled workers' tacit knowledge (design rules) and trial-and-error. The advent of sophisticated algorithms running on computers not only makes this process easier and faster, even for non-experts, but also helps designers to design products that are better from the viewpoint of manufacturing.

Of the various aspects of design evaluation, in this thesis, we discuss algorithms to check the *cleanability* of a design. We aim to develop algorithms that help designers to design parts that are easy to clean during the manufacturing process.

Design-for-cleanability, encouraging a design that is easy to clean, is a newly emerging concept in product design. Reliably removing solid particle contaminants from the surfaces of mechanical parts has become increasingly important in the automotive industry [Berger 2006]. As the complexity and precision of mechanical parts and assemblies have increased, the possibility of in-service failures caused by manufacturing-related hard particle contamination has increased considerably. Even tiny manufacturing byproducts such as detached burrs and chips from machining and sand from casting may shorten product life or may cause a failure [Arbelaez et al. 2008; Avila et al. 2006].

1.1 Cleanability with high-pressure water jets

In this thesis, we consider cleaning using high-pressure water jets to clean off the surfaces of workpieces. Specifically, we solve the following two problems in this thesis: predicting water trap regions of a workpiece and finding a rotation axis to drain a workpiece.

We solve these two problems purely from a geometric perspective. Although physics-based methods such as the finite element method are popular in computer evaluation of design, a geometric approach can also play an important role. Generally speaking, a geometric approach has an advantage over a physics-based approach with respect to performance; we do not have to wait a long time to get simulation results. Our goal is to develop an algorithm that does not rely on a computationally expensive method.

1.1.1 Predicting Water Trap Regions of a Workpiece

Although water jets are effective for removing contaminants from the surface of mechanical parts, the water may become trapped inside the part if the geometry of voids is complex. Since water traps decrease water jet cleaning efficiency, predicting such cleaning-incompatible regions is important to reduce manufacturing time and cost. Trapped water not only creates stagnation areas that prevent efficient flow cleaning, but eventually all the water must be drained and the workpiece must be dried after cleaning is complete.

Taking into account these factors, finding an orientation that minimizes the potential water trap regions or controls their locations when the workpiece is fixtured for water jet cleaning is important to increase the cleaning efficiency and reduce the draining time and effort after cleaning.

In chapter 3, we propose a new method to detect potential water trap regions in voids of oriented polygonal models that approximate the geometry of mechanical parts. We construct a directed graph that captures the flow of water in voids of a 3D input model, based on a fast orientation-dependent volume segmentation approach. We can quickly find the water trap regions by analyzing the directed graph. Since we take a purely geometric approach to solve this problem without employing a physical simulation, even if the geometry of the voids is complicated, we can find such regions quickly. We assume that the part geometry is given as a 2-manifold triangulated polygonal mesh.

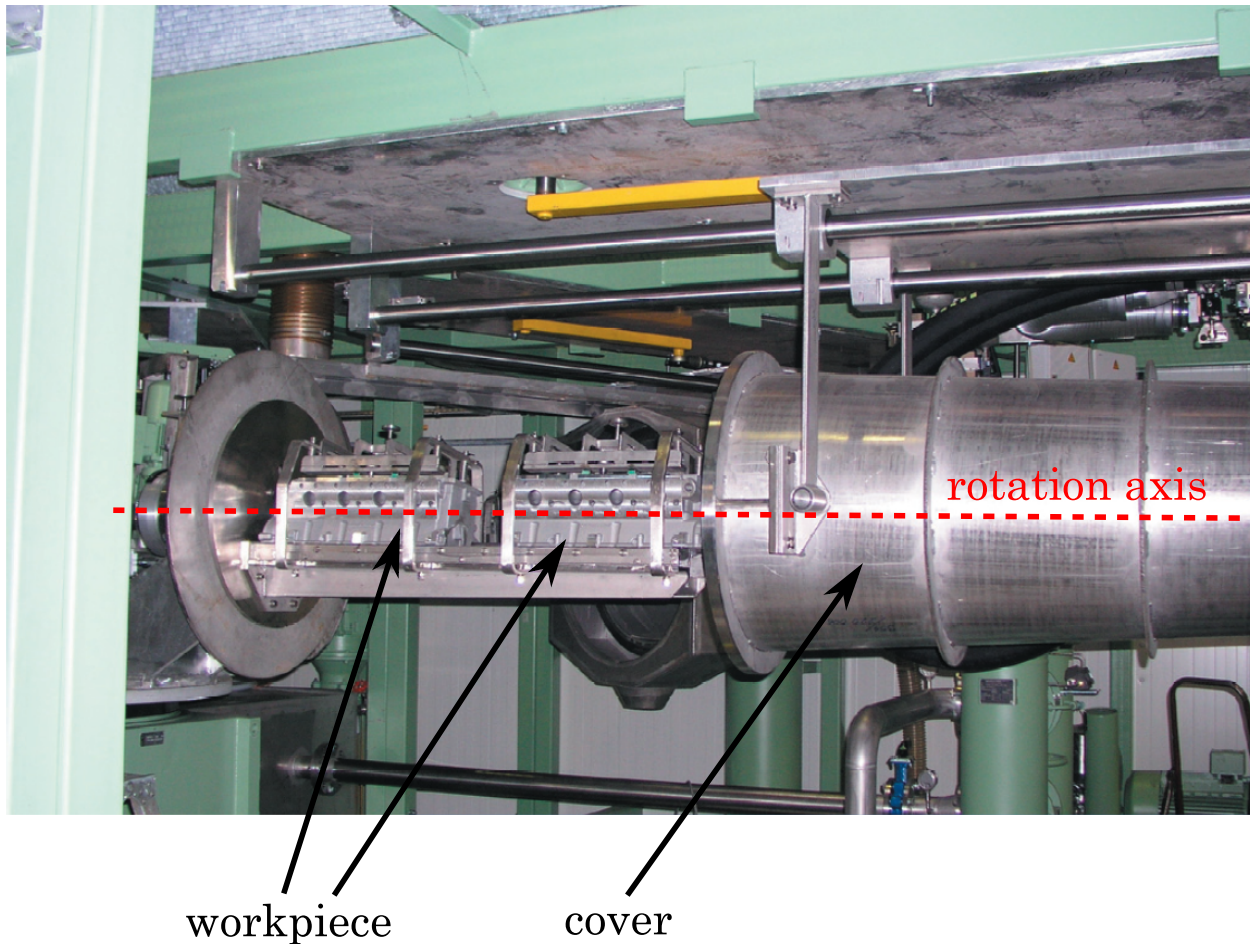


Figure 1.1: Device for draining and subsequent drying (<http://www.mtm-gmbh.com/>)

1.1.2 Finding a Rotation Axis to Drain a Workpiece

When manufacturing byproducts such as chips from machining and sand from casting are cleaned off the surfaces of workpieces by high pressure water-jets, if the workpiece has complicated concave regions, the cleaning water may not easily drain from the workpiece. To minimize the subsequent drying time, our industrial partner first mounts workpieces on a slowly rotating carrier so that gravity can drain out as much water as possible. Their current setup rotates in one direction (either clockwise or counterclockwise) around a single axis oriented parallel to the ground.

Figure 1.1 shows a typical device used for the draining and subsequent drying. After we drain as much water as possible from the workpiece by rotating it around the rotation axis, the cover is closed to form a vacuum chamber. The vacuum causes evaporation of the residual cleaning water and dries the workpiece (heat may also be applied here). The less water remains inside the workpiece, the faster we can dry the workpiece, and with less

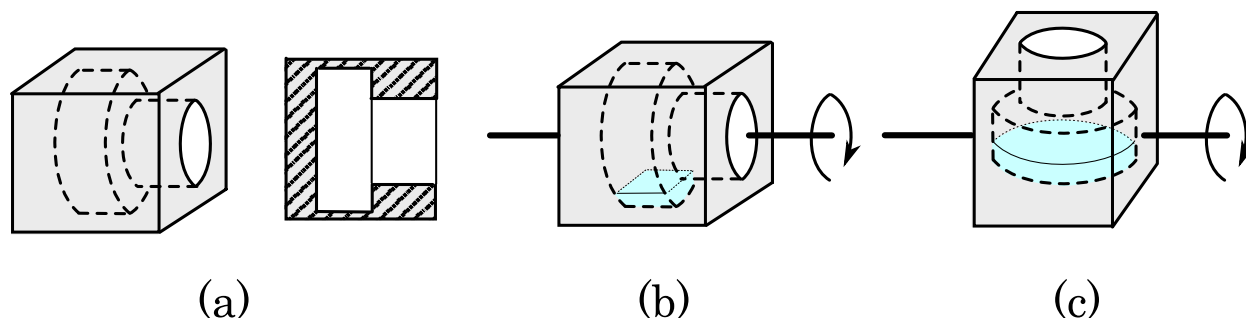


Figure 1.2: The choice of a rotation axis determines whether or not trapped water will drain. (a) A workpiece geometry and its cross-section. (b) A rotation axis relative to the workpiece that cannot drain the trapped water. (c) A rotation axis relative to the workpiece that can drain the trapped water.

energy.

The choice of a rotation axis determines whether trapped water will drain completely under gravity. Figure 1.2 illustrates a simple example. Given a workpiece geometry we would like to drain (Figure 1.2 (a)) and a rotation axis set parallel to the ground, if we initially choose a rotation axis relative to the workpiece as shown in Figure 1.2 (b), we cannot drain the trapped water. On the other hand, if we choose a rotation axis relative to the workpiece as shown in Figure 1.2 (c), we can completely drain all the trapped water.

Our ultimate goal is to find a rotation axis for a given workpiece geometry such that when the workpiece is first oriented with this axis parallel to the ground and then rotated slowly around the axis, all water drains from all voids of the workpiece. If such a rotation axis is not found, the system should notify a designer that the current design is not desirable from a cleanability perspective.

In chapter 4, given a triangular mesh defining the geometry of a 3D workpiece filled with water, we propose an algorithm to test whether, for a specified axis, the workpiece will be completely drained under gravity when the rotation axis is set parallel to the ground and the workpiece is rotated around the axis. Observing that all water traps contain a concave vertex, we solve our problem by constructing and analyzing a directed *draining graph* whose nodes correspond to concave vertices of the geometry and whose edges are set according to the transition of trapped water when we rotate the workpiece around the given axis. Our algorithm to test whether or not a given rotation axis drains the workpiece outputs a result in about a second for models with more than 100,000 triangles, after a few seconds of preprocessing.

In chapter 5, we introduce an algorithm to find the set of *all* rotation axes that would drain a given workpiece geometry (again represented as a triangulated mesh). Suppose that we are given a rotation axis that is found to drain the workpiece by the previous algorithm. In practice, it is also essential to make sure that the rotation axis's nearby rotation axes also drain the workpiece because, although the size of a water particle is finite in the real

world, we have assumed that the size of a water particle is infinitesimal in the algorithm. If one of the nearby rotation axes cannot drain the workpiece, the given rotation axis might not drain the workpiece in the real world. Taking this into account, if we were to use the previous testing algorithm to find a rotation axis that can drain the workpiece in practice, we may need to test a great many rotation axes to find one that is feasible. This is generally time-consuming and less accurate. Thus, we are motivated to move beyond a sample-based approach (testing given axes) to a configuration space approach (finding all drainable axes). We introduce a new algorithm to find rotation axes that would drain the workpiece by introducing the *extended draining graph* that represents all the possible transitions of water particles considering all the possible rotation axes and rotation directions. We introduce a dual-space stabbing line approach to efficiently analyze the drainability of all possible paths through this graph, for all possible rotation axes. Since we take a configuration space approach, our algorithm can find every possible rotation axis that drains the workpiece.

Chapter 2

Related Work

Analytical tools that predict cleaning effectiveness at the design and process planning stages are needed to increase the efficiency of the cleaning processes. Initial research has focused on understanding the effect of key cleaning process parameters [Arbelaez et al. 2008; Avila et al. 2006; Avila et al. 2005; Berger 2006].

Generally speaking, manufacturing processes are complex phenomena, especially when fluid is involved. The most straightforward approach to solve our problems might at first appear to be a general-purpose physics based approach such as computational fluid dynamics (CFD).

In the computer graphics community, several efforts have been made to accelerate algorithms borrowed from computational sciences while maintaining plausibility [Müller et al. 2008]. The first real-time GPU (Graphics Processing Unit) implementation of fluid simulation using a regular grid of cubical cells was reported in [Nguyen 2007]. Unfortunately, because the algorithm accuracy is dependent on the 3D grid resolution, it is not appropriate for our complex target geometries since we would be required to split space into a tremendous number of grid cells to perform the simulation reliably. To avoid this issue, particle-based approaches using smoothed particle hydrodynamics (SPH) are popular for real-time simulations since they do not require a grid throughout the whole domain [Müller-Fischer et al. 2003; Müller-Fischer et al. 2007]. Harada et al. introduced the algorithm to implement the entire SPH computation on the GPU and showed that we can perform real-time SPH simulation with several tens of thousands of particles [Harada et al. 2007].

Although the computational power of CPUs and GPUs is increasing every year, the computational cost of such a physics-based approach is still more expensive than necessary to be suitable for our applications that require interactivity. Since we would like to provide interactive feedback to designers, we need an algorithm that does not rely on a computationally expensive method. Considering that we do not care about the full details of the fluid flow, only predicting the regions of the workpiece where water traps are potentially formed (chapter 3) and whether or not the workpiece drains completely (chapter 4, 5), we are motivated to devise an algorithm to solve our problem geometrically to reduce computational cost. On top of that, a configuration space approach such as we introduce in chapter 5 (which is by

nature geometric) allows us to examine the entire solution space in an integrated manner, rather than solving the problem heuristically using simulation for individual samples in the solution space. Our algorithms combine analysis of (free) fluid flow and accessibility from a geometric perspective.

In the case of fluid flows inside geometric models, Aloupis et al. introduced the problem of rolling a single small ball (akin to a single water particle) out of a closed polygon by rotating the shape in 2D space [Aloupis et al. 2008]. Given a closed polygon and a trapped single particle inside of the polygon, they proposed an algorithm to find how many holes must be punctured to “drain” the particle by rotation. Letting N be the number of vertices of the polygon, they showed that $\lfloor N/6 \rfloor$ holes are sometimes necessary and $\lceil N/4 \rceil$ holes are sufficient to drain any polygon. They proposed an $O(N^2 \log N)$ -time algorithm to find the minimum number of holes needed to drain.

Geometric problems often arise in manufacturing processes. Janardan and Woo gave a survey and introduced open manufacturing problems that can be considered as geometric problems [Janardan and Woo 2004]. For example, geometric analysis has been developed to study the flow of liquid in a mold, a problem with some similarities to our first problem, predicting water trap regions of workpiece (chapter 3). Bose and Toussaint proposed an algorithm to find an orientation for a gravity casting mold that eliminates surface defects and insures a complete fill without air traps [Bose and Toussaint 1995; Bose et al. 1993]. Letting N be the number of vertices of the polyhedron representing the geometry of the mold, their algorithm determines whether the mold can be filled without forming air traps in $O(N)$ time. They also proposed an $O(N^2)$ -time algorithm to find the orientation that minimizes the number of venting holes that need to be added to allow air to escape to insure a complete fill.

A similar problem to our other problem, finding a rotation axis to drain a 3D workpiece (chapters 4 and 5), arises in planning for NC machining. To manufacture a desired shape using an NC machine, multiple setups are often required because, in general, the cutting tool cannot access all the part of the workpiece in a single setup. For 3-axis machines, the tool can only translate x , y , and, z for a given set up; for a 4-axis machine, one additional rotational degree of freedom is provided. Since fixturing for each setup can be time-consuming and multiple setups are a source of errors, the number of setups should be minimized. To consider this problem, visibility plays a vital role. Chen et al. converted the accessibility problem to geometric problems on the Gaussian sphere by introducing the concept of the visibility map of the surface [Chen et al. 1993; Chen and Woo 1992; Woo 1994]. Given a portion of the workpiece’s surface, the corresponding visibility map is the set of all directions along which the cutting tool (which is treated as a point) can see every point on the surface portion. The visibility map of a surface is represented as a spherical polygon on the Gaussian sphere.

Using the visibility map, finding a rotation axis that maximizes “visible” surfaces in a single setup in 4-axis NC machining is solved as a geometric problem of finding a great circle passing through spherical convex polygons [Chen et al. 1993]. Tang et al. solved this problem by converting it to the problem of finding a line passing through convex polygons on a 2D plane using the Gnomonic projection [Tang et al. 1992]. The algorithm runs in $O(EN^2 +$

$N^3 \log N$) time, where N is the number of spherical polygons and E is the total number of edges. Gupta et al. solved the problem using duality transformations and improved the running time to $O(E^2)$ [Gupta et al. 1996]. Tang et al. further consider the problem of finding arcs of great circles that span 180 degrees (called a semi-great circle), passing through spherical convex polygons using the Gnomonic projection and duality transformations [Tang et al. 1998]. We also take advantage of the Gnomonic projection and duality transformations in our algorithm described in chapter 5.

Chapter 3

Pool Segmentation for Predicting Water Trap Regions

In this chapter, we propose a new method to detect potential water trap regions in voids of oriented polygonal models that approximate the geometry of mechanical parts. Water traps decrease water jet cleaning efficiency because water traps create stagnation areas and prevent efficient flow cleaning. Minimizing the potential water trap also reduces the draining time and effort after cleaning. Therefore, predicting such cleaning-incompatible regions is important to reduce manufacturing time and cost. We construct a directed graph that captures the flow of water in voids of a 3D input model, based on a fast orientation-dependent volume segmentation approach. We can quickly find the water trap regions by analyzing the directed graph. Since we take a purely geometric approach to solve this problem without employing any physical simulation, even if the geometry of the voids is complicated, we can find such regions quickly.

3.1 Algorithm Overview

Our algorithm predicts regions in voids of the geometry where, for a given orientation, the effectiveness of cleaning with water jets will be compromised by water traps. We assume throughout this chapter that the part geometry has been rotated to the desired test orientation, so that gravity always acts vertically (i.e. down the z -axis).

Figure 3.1 illustrates an overview of our algorithm. Letting \mathcal{M} be the geometry of the input model and \mathcal{B} be a slightly enlarged bounding box that encloses \mathcal{M} , the space \mathcal{W} where water flows can be represented as $\mathcal{W} = \mathcal{B} \setminus \mathcal{M}$. We split the space \mathcal{W} horizontally into multiple regions called *pools* based on topological changes of \mathcal{W} with respect to the z -axis. Then, we build a directed graph whose nodes correspond to the pools and whose edges connect two nodes if water flowing out of the source node’s corresponding pool could enter the destination node’s corresponding pool. We determine water trap regions by analyzing the directed graph.

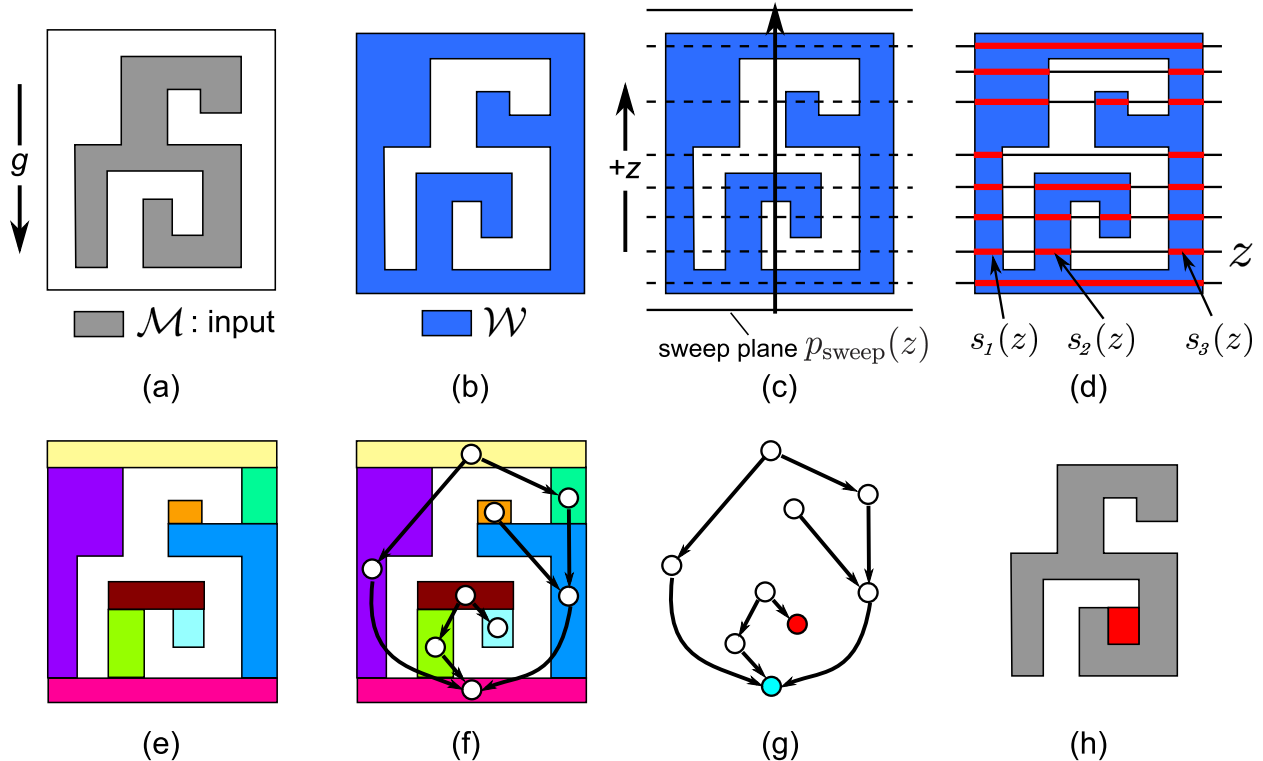


Figure 3.1: Algorithm overview: (a) From the input geometry \mathcal{M} , (b) we define the space $\mathcal{W} = \mathcal{B} \setminus \mathcal{M}$ where water can flow. (c) With a sweep plane $p_{\text{sweep}}(z)$, (d) we track the evolution of connected slice components $s_i(z) \in \mathcal{W} \cap p_{\text{sweep}}(z)$. (e) At locations where slice components split or merge, we segment \mathcal{W} into pools, and (f) assign directed edges that capture the water flow between pairs of pools. (g) From the graph, we locate potential water trap regions. (h) We map the region(s) to the input \mathcal{M} .

The Reeb graph [Reeb 1946] is a data structure for representing the topology of shapes that captures topological changes with respect to a real function defined on the shapes. The directed graph we construct is mathematically equivalent to a Reeb graph of a 3-manifold with boundary with respect to the height function (z -value). Hence, we could construct the directed graph from \mathcal{W} using a Reeb graph construction algorithm such as that proposed by Pascucci et al. [Pascucci et al. 2007] or Tierny et al. [Tierny et al. 2009] and segment \mathcal{W} into pools based on the Reeb graph constructed. However, since their approaches require the extra burden of tetrahedralizing \mathcal{W} , we propose an alternative efficient approach of segmenting \mathcal{W} into pools and constructing the corresponding directed graph simultaneously in our work.

Our pool segmentation data structure equipped with the directed graph has more potential than just predicting water trap regions. For just predicting water trap regions, the pool segmentation data structure is somewhat redundant since, whereas predicting water trap

regions is a local problem, the pool segmentation data structure provides global information. Nevertheless, we chose our data structure because, once we find potential water trap regions, we could also use it when performing the actual filling state simulation inside mechanical parts, taking an inflow location as an input as a next step. We believe that our pool segmentation data structure would help to accelerate the simulation. For further discussion on this topic, please refer to chapter 6.

3.1.1 Preliminaries

First, we introduce some notation to explain how we split \mathcal{W} into pools and add the directed edges between nodes corresponding to pools. We consider a sweep plane $p_{\text{sweep}}(z = z)$ perpendicular to the z-axis (i.e. the gravity direction) intersecting it at z . Given a sweep plane $p_{\text{sweep}}(z)$, we define the *slice* at z , $S(z)$, as the intersection of \mathcal{W} and $p_{\text{sweep}}(z)$: $S(z) = \mathcal{W} \cap p_{\text{sweep}}(z)$. As shown in Figure 3.1 (d), slice $S(z)$ may consist of multiple disconnected slice components, which in 3D will be 2D polygons (possibly with holes). We call these *slice polygons*. We denote the different slice polygons constituting $S(z)$ as $s_i(z)$ ($1 \leq i \leq |S(z)|$).

Then, we let $proj(s_i(z))$ be the projection of $s_i(z)$ to the plane perpendicular to the z-axis, and the z-value just below z be $z^- = z - \epsilon$ and the z-value just above z be $z^+ = z + \epsilon$, ϵ a positive infinitesimal number. Given a slice polygon $s_i(z) \in S(z)$, we define overlapping slice polygon(s) just below $s_i(z)$, $S_{\text{below}}(s_i(z))$, as the set of slice polygons $s_j(z^-) \in S(z^-)$ such that $proj(s_i(z)) \cap proj(s_j(z^-)) \neq \emptyset$. Similarly, we define overlapping slice polygon(s) just above $s_i(z)$, $S_{\text{above}}(s_i(z))$, as the set of slice polygons $s_j(z^+) \in S(z^+)$ such that $proj(s_i(z)) \cap proj(s_j(z^+)) \neq \emptyset$.

Based on the cardinality of $S_{\text{below}}(s_i(z))$ and $S_{\text{above}}(s_i(z))$, the slice polygons just below and above $s_i(z)$, we classify each slice polygon $s_i(z)$ as one of four types as follows. Given a slice polygon $s_i(z)$, if $|S_{\text{below}}(s_i(z))| = 0$, we call $s_i(z)$ a *beginning slice polygon* since a new slice polygon appears as the sweep plane moves from $p_{\text{sweep}}(z^-)$ to $p_{\text{sweep}}(z^+)$. On the other hand, if $|S_{\text{above}}(s_i(z))| = 0$, we call $s_i(z)$ an *ending slice polygon*, since an existing slice polygon disappears as the sweep plane moves from $p_{\text{sweep}}(z^-)$ to $p_{\text{sweep}}(z^+)$. If $|S_{\text{below}}(s_i(z))| \geq 2$ and $|S_{\text{above}}(s_i(z))| \geq 1$ or $|S_{\text{below}}(s_i(z))| \geq 1$ and $|S_{\text{above}}(s_i(z))| \geq 2$, we call $s_i(z)$ a *merge/split slice polygon* since multiple slice polygons merge into one slice polygon and/or one slice polygon splits into multiple slice polygons as the sweep plane moves from $p_{\text{sweep}}(z^-)$ to $p_{\text{sweep}}(z^+)$. Finally, if $|S_{\text{below}}(s_i(z))| = |S_{\text{above}}(s_i(z))| = 1$, we call $s_i(z)$ a *no-change slice polygon* since no topological change of slice polygon $s_i(z)$ occurs as the sweep plane moves from $p_{\text{sweep}}(z^-)$ to $p_{\text{sweep}}(z^+)$.

3.1.2 Pool Segmentation

We define a pool as the union of no-change slice polygons bounded by either a beginning or a merge/split slice polygon from below and either an ending or a merge/split slice polygon from above. Given a slice polygon $s_i(z)$, we let $pool(s_i(z))$ be the pool $s_i(z)$ defines.

We segment \mathcal{W} into pools using a sweep plane algorithm, where we imagine moving $p_{\text{sweep}}(z)$ from $z = -\infty$ to $z = +\infty$. If $\mathcal{W} \cap p_{\text{sweep}}(z)$ yields a beginning slice polygon, we generate a new pool bounded from below by the beginning slice polygon. If $\mathcal{W} \cap p_{\text{sweep}}(z)$ yields a no-change slice polygon, the no-change slice polygon $s_i(z)$ defines the pool $pool(s_j(z^-))$ where $s_j(z^-) \in S_{\text{below}}(s_i(z))$. If $\mathcal{W} \cap p_{\text{sweep}}(z)$ yields an ending slice polygon, we complete the corresponding existing pool, bounding it from above with the ending slice polygon. Finally, if $\mathcal{W} \cap p_{\text{sweep}}(z)$ yields a merge/split slice polygon, we complete the corresponding existing pool(s) by bounding from above with the merge/split slice polygon, and generate new pool(s) by bounding from below with the same merge/split slice polygon. Then, for $1 \leq i \leq |S(z^-)|$ and for $1 \leq j \leq |S(z^+)|$, we compute $proj(s_i(z^-)) \cap proj(s_j(z^+))$. If there are p and q such that $proj(s_p(z^-)) \cap proj(s_q(z^+)) \neq \emptyset$, and $pool(s_p(z^-)) \neq pool(s_q(z^+))$, we add a directed edge from the node corresponding to $pool(s_q(z^+))$ to the node corresponding to $pool(s_p(z^-))$ in the directed graph (Figure 3.1 (f)).

3.1.3 Predicting Water Trap Regions

After completing the sweep from $z = -\infty$ to $z = +\infty$, the space \mathcal{W} is segmented into pools that are connected to each other in the graph by edges oriented in the direction of gravity if they are bounded by the same merge/split slice polygon. Each pool represents a region that could potentially be a water trap region except the bottom-most pool, which represents the exterior of \mathcal{M} . Water flowing in \mathcal{W} under gravity will flow between pools according to the directed edges. Once such flowing water reaches the bottom-most pool, since by construction it is outside the input geometry, we consider the water to be drained. Thus, as shown in Figure 3.1 (g), given a pool, if there is no path such that we can reach the bottom-most pool from the corresponding node, the pool is a potential water trap region (whether or not this water trap is actually formed depends upon the inflow location). Since we can compute the volume of water each pool can hold, we can also quantitatively evaluate a given part orientation by summing the volumes of pools that are determined to be water trap regions.

3.2 Pool Segmentation

In this section, we describe the details of our pool segmentation algorithm summarized above, given a 2-manifold triangulated input mesh \mathcal{M} .

From \mathcal{M} , we can easily obtain the corresponding \mathcal{W} by flipping the orientation of the triangles in \mathcal{M} and introducing six rectangles that represent the enlarged axis-aligned bounding box \mathcal{B} . Each rectangle should be split into two triangles such that all the faces of \mathcal{W} are represented by triangles as well.

To implement the pool segmentation algorithm, we have to know for which values of z beginning, ending, and merge/split slice polygons occur. We determine all of these values of z by tracking the evolution of the boundary of slice polygons. Even when a slice polygon boundary appears, disappears, merges, or splits, the corresponding slice polygon does not

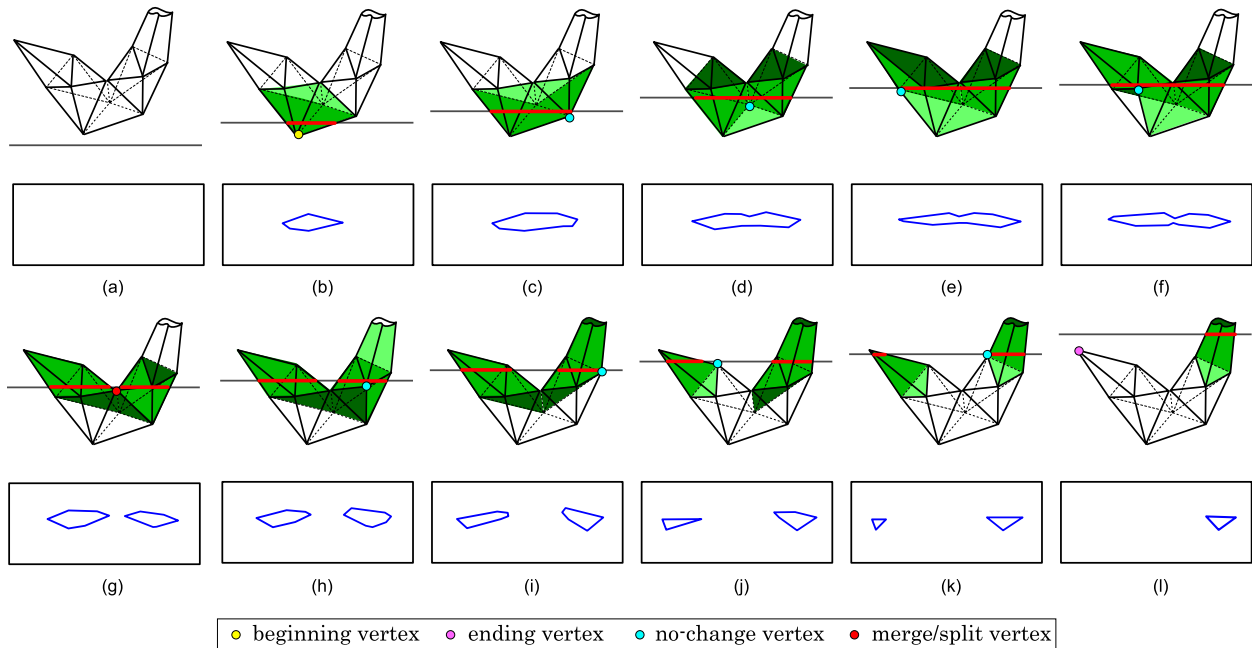


Figure 3.2: Figures (a)–(l) illustrate how the triangles in boundary cycles are updated as the sweep plane moves from bottom to top over a portion of \mathcal{W} . The top drawing in each subfigure shows the set of triangles (colored) currently in boundary cycles just after the sweep plane processes the indicated vertex; the bottom drawing in each subfigure shows the corresponding slice polygon boundaries, with the sweep plane shown as a rectangle.

necessarily appear, disappear, merge, or split (e.g. because the boundary could correspond to a hole in a polygon). However, when a slice polygon appears, disappears, merges, or splits, the corresponding slice polygon boundary does also appear, disappear, merge, or split. Therefore, checking all the values of z where a slice polygon boundary appears, disappears, merges, or splits is sufficient to determine all the values of z where beginning, ending, and merge/split slice polygons occur.

We track the evolution of slice polygon boundaries by modifying McMains’ sweep plane slicing algorithm [McMains 2000; McMains and Séquin 1999]. Observing that slice polygon boundaries appear, disappear, merge, or split only when the sweep plane passes through one of the vertices of the input polygonal mesh, they showed that all such changes can be identified as long as all vertices are checked in ascending order of z -coordinate (vertices whose z -coordinates are the same can be processed in arbitrary order without affecting the final result). In other words, when $\mathcal{W} \cap p_{\text{sweep}}(z)$ yields any of beginning, ending, and/or merge/split slice polygons, $p_{\text{sweep}}(z)$ always intersects one of the vertices in \mathcal{W} .

3.2.1 Boundary Cycles

Our modified sweep plane slicing algorithm tracks the evolution of slice polygon boundaries and determines when a slice polygon boundary appears, disappears, merges, or splits. For this purpose, we manage a status structure called the *boundary cycle* (Figure 3.2). Each boundary cycle consists of a set of triangles. Every triangle in \mathcal{W} is visited three times during sweeping since each triangle has three vertices. Given a triangle, when it is visited for the first time, the triangle is inserted into a boundary cycle. When it is visited for the third time, the triangle is deleted from the boundary cycle. (When it is visited for the second time, nothing happens.) Since we process each vertex in order of ascending z-coordinate, triangles currently in a boundary cycle always intersect the current sweep plane.

Each slice polygon boundary at z is represented by a closed polygonal chain on $p_{\text{sweep}}(z)$. Each line segment constituting the closed polygonal chain is defined by the intersection between $p_{\text{sweep}}(z)$ and a triangle. Letting $V(z)$ be the set of vertices in \mathcal{W} whose z-coordinate is z , a set of triangles in a boundary cycle defines a slice polygon boundary at z where $V(z) = \emptyset$ (i.e. where $p_{\text{sweep}}(z)$ does not intersect any vertices in \mathcal{W}) as shown in Figure 3.2. Therefore, the number of line segments constituting a slice polygon boundary is equal to the number of triangles in the corresponding boundary cycle. For such z , there is exactly one boundary cycle for each slice polygon boundary. For z where $V(z) \neq \emptyset$ (i.e. where $p_{\text{sweep}}(z)$ intersects at least one vertex in \mathcal{W}), a set of triangles in a boundary cycle does not necessarily define a slice polygon boundary since some triangles in the boundary cycle may be parallel to the sweep plane; the intersection between such a triangle and $p_{\text{sweep}}(z)$ is not a line segment. However, this limitation does not become a problem because, to determine the type of slice polygons at z where $V(z) \neq \emptyset$, it is sufficient to consider the slice polygons just below and just above vertices in $V(z)$ (i.e. at z^- and z^+) as explained previously in section 3.1.1. During sweeping from $z = -\infty$ to $z = +\infty$, we track the evolution of slice polygon boundaries by tracking the set of triangles in each boundary cycle.

3.2.1.1 Boundary Cycle Management

Boundary cycles are generated, completed, or updated when we process each vertex in \mathcal{W} . As McMains and Séquin showed in their work, we can classify each vertex into one of four types: *beginning vertex*, *ending vertex*, *no-change vertex*, and *merge/split vertex*. A beginning vertex is where a new boundary cycle is generated. An ending vertex is where an existing boundary cycle is completed. A no-change vertex is where some triangles may be deleted from and inserted into an existing boundary cycle. A merge/split vertex is where multiple boundary cycles merge into one boundary cycle or one boundary cycle splits into multiple boundary cycles. At a merge/split vertex, we complete existing boundary cycle(s) and generate new boundary cycle(s) according to the merge or split. Appendix A.1 and A.2 describe, for a given $V(z)$, how to classify each vertex $v \in V(z)$ into one of the four types, and generate, complete, and update boundary cycles accordingly. When $\mathcal{W} \cap p_{\text{sweep}}(z)$ yields a beginning, an ending, or a merge/split slice polygon, $p_{\text{sweep}}(z)$ always intersects with a beginning, an

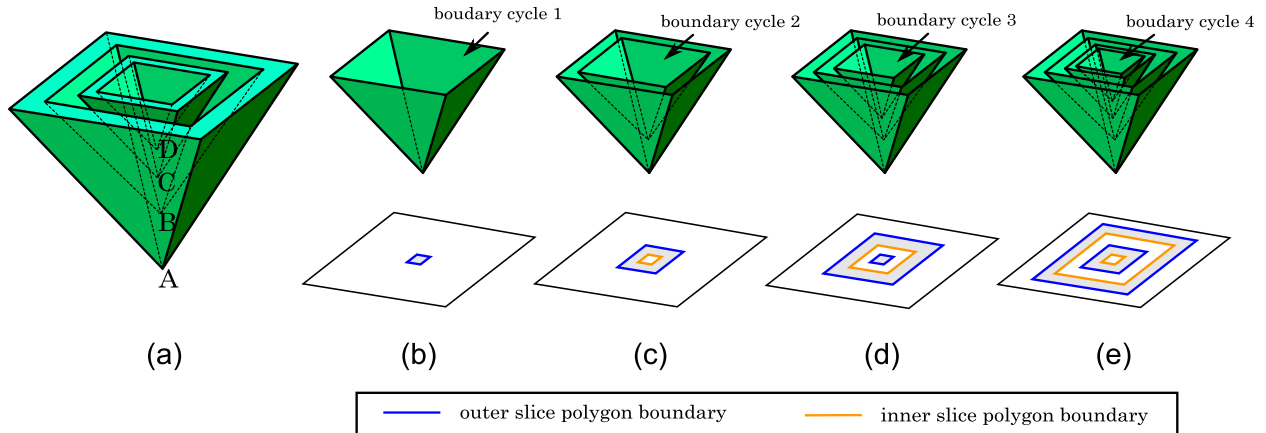


Figure 3.3: Each of subfigures (b)–(e) shows the boundary cycles and the corresponding slice polygon boundaries on the sweep plane just after processing vertices A , B , C , and, D shown in (a), respectively. Boundary cycles 1 and 3 are outer boundary cycles; Boundary cycles 2 and 4 are inner boundary cycles. For each slice polygon, the inner boundary cycles that define the inner slice polygon boundaries are associated with the outer boundary cycle that defines the outer slice polygon boundary. Boundary cycle 2 is associated with boundary cycle 1; boundary cycle 4 is associated with boundary cycle 3.

ending, or a merge/split vertex, respectively (but not vice versa).

When a new boundary cycle is generated at a beginning vertex, a new slice polygon boundary appears (Figure 3.2 (b)); when an existing boundary cycle is completed at an ending vertex an existing slice polygon boundary disappears (Figure 3.2 (l)); and, when multiple boundary cycles merge into one boundary cycle or one boundary cycle splits into multiple boundary cycles at a merge/split vertex, multiple slice polygon boundaries merge into one slice polygon boundary or one slice polygon boundary splits into multiple slice polygon boundaries (Figure 3.2 (g)).

We update triangles in an existing boundary cycle at a no-change vertex (Figure 3.2 (c)–(f) and (h)–(k)). We consider two boundary cycles at different values of z to be the same boundary cycle if one is obtained from the other by processing only no-change vertices. Thus, a new boundary cycle will be generated at a beginning vertex or a merge/split vertex and an existing boundary cycle will be completed at an ending vertex or a merge/split vertex.

3.2.1.2 Boundary Cycle Classification

As shown in Figure 3.3, a slice polygon may be bounded by more than one slice polygon boundary. More specifically, a slice polygon is always bounded by one outer slice polygon boundary plus zero or more inner slice polygon boundaries. Given a slice polygon $s_i(z)$, let $\partial s_i(z)$ be the set of slice polygon boundaries that bound $s_i(z)$. We also let $(\partial s_i(z))_1$ be the outer slice polygon boundary and $(\partial s_i(z))_j$ ($j \geq 2$) be the inner slice polygon boundaries of

$s_i(z)$. Then, $\partial s_i(z) = \{(\partial s_i(z))_1, \dots, (\partial s_i(z))_{|\partial s_i(z)|}\}$.

A boundary cycle is classified as either an *outer boundary cycle* or an *inner boundary cycle* depending on whether the intersection between triangles in the boundary cycle and $p_{\text{sweep}}(z)$ define an outer or an inner slice polygon boundary. An inner boundary cycle is always associated with an outer boundary cycle that immediately encloses the inner boundary cycle (Figure 3.3).

For a given z , a boundary cycle is classified as an outer boundary cycle or an inner boundary cycle by shooting a ray perpendicular to the z -axis from an arbitrary point at z on a triangle in the boundary cycle and counting the number of intersections between the ray and triangles in \mathcal{W} , excluding triangles in the boundary cycle that we are testing. If it is even, the boundary cycle is an outer boundary cycle. If it is odd, it is an inner boundary cycle. For each inner boundary cycle, we can find the outer boundary cycle immediately enclosing the inner boundary cycle by counting the number of intersections with triangles in each outer boundary cycle. If there is an outer boundary cycle where the number of the intersections is odd, the outer boundary cycle encloses this inner boundary cycle. If multiple such enclosing boundary cycles exist, the one with the closest intersection is the immediately enclosing one.

3.2.2 Pool Segmentation

Here, we describe how to implement pool segmentation. In section 3.1.2, we defined that a pool is the union of no-change slice polygons bounded by either a beginning or a merge/split slice polygon from below and either an ending or a merge/split slice polygon from above. We have observed that we can determine where these slice polygons occur during sweeping by tracking the evolution of boundary cycles. Therefore, we construct a pool according to generation, completion, and updating of boundary cycles. In practice, we construct a pool by finding its boundary. Specifically, the side of a pool is defined by triangles from \mathcal{W} , possibly trimmed. The bottom and top face of a pool is defined by the slice polygons where the pool is generated and completed.

Each pool is defined by one outer boundary cycle plus zero or more inner boundary cycles. Triangles in such boundary cycles form the sides of the pools. Each of these triangles is trimmed if a portion of the triangle is lower than the lower bound z -coordinate and/or higher than the upper bound z -coordinate of the pool. The intersection between the triangles and the sweep plane at the lower/upper bound z -coordinate define the bottom face and the top face of the pool, respectively.

We generate a new pool p at a z -coordinate z where the slice topology changes by performing the following operations that *initialize* the pool defined by the outer boundary cycle b_{outer} at z . Letting $\text{inner}(b_{\text{outer}})$ be the set of enclosed inner boundary cycles associated with b_{outer} , we assign the triangles in b_{outer} and each $b_{\text{inner}} \in \text{inner}(b_{\text{outer}})$ to p . Then, for each triangle assigned to p at its lowest z -value, if a portion of the triangle is lower than z^+ , we trim that portion (which may be the entire triangle). Then, we define the bottom face of p by connecting the line segments defined by intersections between the trimmed triangles and

Algorithm 1 InitializePool(b_{outer}, z)

Input: b_{outer} : an outer boundary cycle, z : z-coordinate

Output: p : pool generated at z

$b_{outer} \rightarrow pool \leftarrow p$

$p \rightarrow T_1 \leftarrow \emptyset$

$(\partial s)_1 \leftarrow \emptyset$

for each triangle $t \in b_{outer}$ **do**

 Compute t_{cut} , the portion of t higher than z

$p \rightarrow T_1 \leftarrow (p \rightarrow T_1) \cup t_{cut}$

$(\partial s)_1 \leftarrow (\partial s)_1 \cup (t \cap p_{sweep}(z^+))$

end for

$j \leftarrow 2$

for each $b_{inner} \in inner(b_{outer})$ **do**

$b_{inner} \rightarrow pool \leftarrow p$

$p \rightarrow T_j \leftarrow \emptyset$

$(\partial s)_j \leftarrow \emptyset$

for each triangle $t \in b_{inner}$ **do**

 Compute t_{cut} , the portion of t higher than z

$p \rightarrow T_j \leftarrow (p \rightarrow T_j) \cup t_{cut}$

$(\partial s)_j \leftarrow (\partial s)_j \cup (t \cap p_{sweep}(z^+))$

end for

$j \leftarrow (j + 1)$

end for

$p \rightarrow \text{BottomFace} \leftarrow \bigcup_{k=1}^{j-1} (\partial s)_k$

return p

$p_{sweep}(z^+)$. Note that, if p is defined by n boundary cycles, the bottom face consists of n closed polygonal chains. The bottom face corresponds to a slice polygon $s_i(z^+)$ and each of the closed polygonal chains corresponds to slice polygon boundary $(\partial s_i(z^+))_j$ ($1 \leq j \leq n$), where $n = |\partial s_i(z^+)|$. Algorithm 1 gives the corresponding pseudocode for initializing a pool. In a similar manner, we complete an existing pool p at a z-coordinate z where the slice topology changes again by performing analogous operations to *finalize* the pool defined by the outer boundary cycle b_{outer} at this z , the highest z-value for the pool. The difference is that we trim the triangles and define the top face at z^- , instead of the bottom face at z^+ (Algorithm 2).

We describe how to construct each pool based on generation, completion, and updating of boundary cycles in detail. For the sake of simplicity of explanation, we assume that any slice polygon $s_i(z)$ is bounded by only one outer slice polygon boundary for a moment (i.e. $|\partial s_i(z)| = 1$ for any z and $inner(b_{outer}) = \emptyset$ for any outer boundary cycle b_{outer}). Thus, any boundary cycle we encounter during sweeping will always be an outer boundary cycle.

In this case, the appearance, disappearance, merging, and splitting of slice polygon

Algorithm 2 FinalizePool(b_{outer}, z)

Input: b_{outer} : outer boundary cycle, z : z -coordinate
Output: p : pool completed at z
 $p \leftarrow b_{outer} \rightarrow pool$
 $b_{outer} \rightarrow pool \leftarrow \mathbf{nil}$
 $n \leftarrow 1 + |inner(b_{outer})|$ // number of boundary cycles
for $j = 1$ to n **do**
 $T_{cut} \leftarrow \emptyset$
 $(\partial s)_j \leftarrow \emptyset$
 for each triangle $t \in (p \rightarrow T_j)$ **do**
 Compute t_{cut} , a portion of t lower than z
 $T_{cut} \leftarrow T_{cut} \cup t_{cut}$
 $(\partial s)_j \leftarrow (\partial s)_j \cup (t \cap p_{sweep}(z^-))$
 end for
 $p \rightarrow T_j \leftarrow T_{cut}$
end for
 $p \rightarrow \text{TopFace} \leftarrow \bigcup_{j=1}^n (\partial s)_j$
return p

boundaries always leads to appearance, disappearance, merging, and splitting of the corresponding slice polygons. Thus, for a given z where $V(z) \neq \emptyset$, if a new boundary cycle is generated, we initialize a new pool defined by the boundary cycle. If an existing boundary cycle is completed, we finalize the existing pool defined by the boundary cycle.

For a given z where $V(z) \neq \emptyset$, let G , C , and U be the sets of new boundary cycles generated, completed, and updated at $v \in V(z)$, respectively. The specific algorithm to find G , C , and U from $V(z)$ is described in Appendix A.3. A pool is constructed by the following rules.

For a given z where $V(z) \neq \emptyset$:

1. For each updated boundary cycle $b \in U$, we add new triangles inserted into b at z to the pool defined by b .
2. For each completed boundary cycle $b \in C$, we finalize the pool defined by b .
3. For each newly-generated boundary cycle $b \in G$, we initialize the pool p defined by b .
4. Let P_{init} be the set of pools initialized and P_{final} be the set of pools finalized at z . If $P_{init} \neq \emptyset$ and $P_{final} \neq \emptyset$, for $p_i \in P_{init}$ and for $p_f \in P_{final}$, we compare the bottom face of p_i and the top face of p_f . If they overlap, we add a directed edge from the node corresponding to p_i to the node corresponding to p_f in the directed graph.

Algorithm 3 shows the corresponding pseudocode. In Algorithm 3, **InitializePool** and **FinalizePool** were shown in Algorithm 1 and 2, respectively. **ProcessVertices** takes $V(z)$

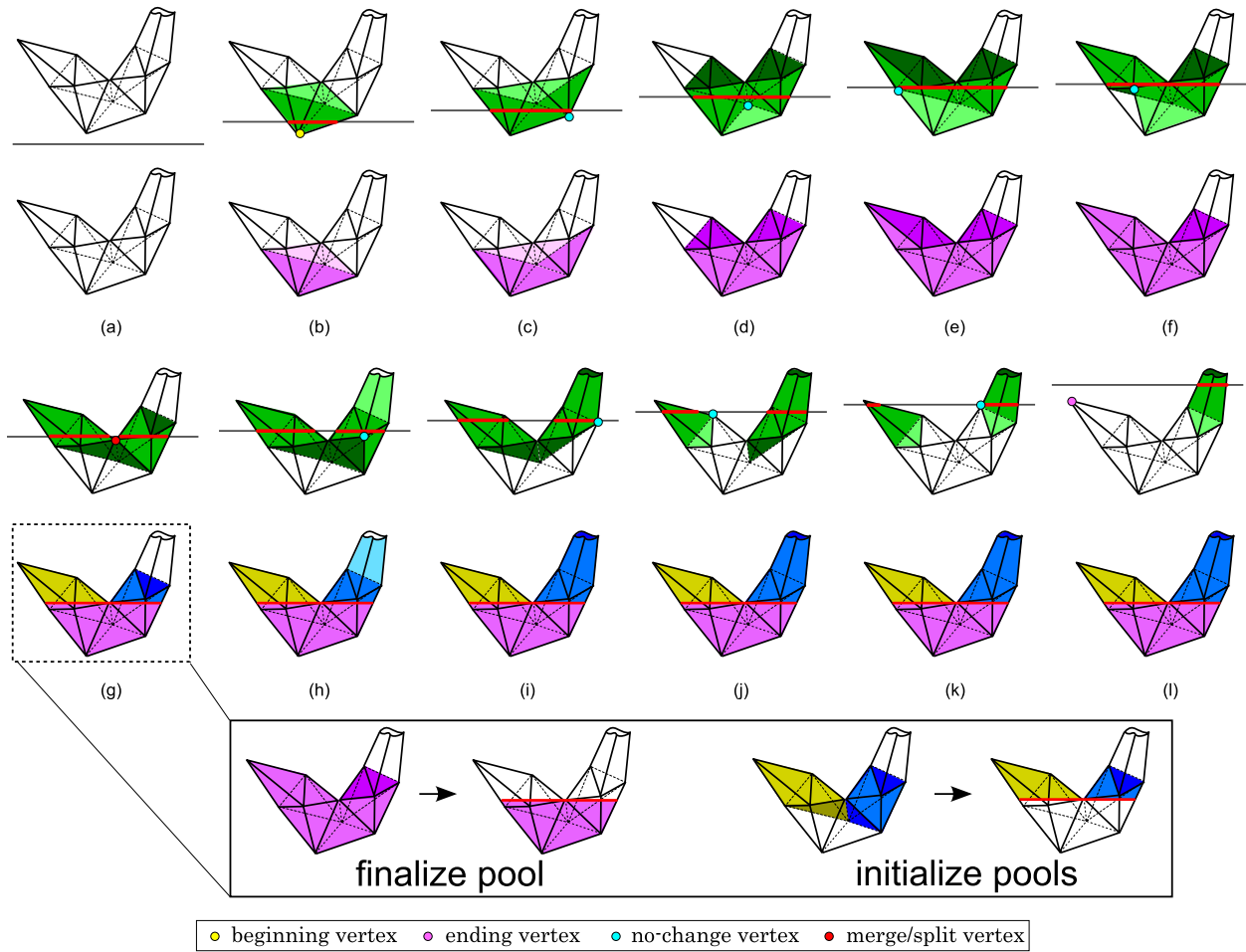


Figure 3.4: Figures (a)–(l) show how pools are constructed according to generation, completion, and updating of boundary cycles. The top drawing in each subfigure shows the set of triangles in boundary cycles just after the sweep plane processes the indicated vertex as in Figure 3.2; the bottom drawing shows the corresponding construction of pools.

as input and returns a set of boundary cycles generated, completed, and updated at z , respectively (refer to Algorithm 15). **ConstructConnectivity** compares the bottom face(s) of generated pool(s) and the top face(s) of completed pool(s) by performing a 2D polygon intersection test.

A series of steps to construct pools based on these rules is illustrated in Figure 3.4. When a new boundary cycle is generated, we initialize a new pool defined by the boundary cycle (Figure 3.4 (b)). In this example, since the intersection between the triangles and the corresponding sweep plane becomes a point, no triangles are trimmed and the bottom face consists of a single point. When the triangles in the boundary cycles are updated, we assign the inserted triangles to the pool defined by their boundary cycles (Figure 3.4 (c)–(f) and

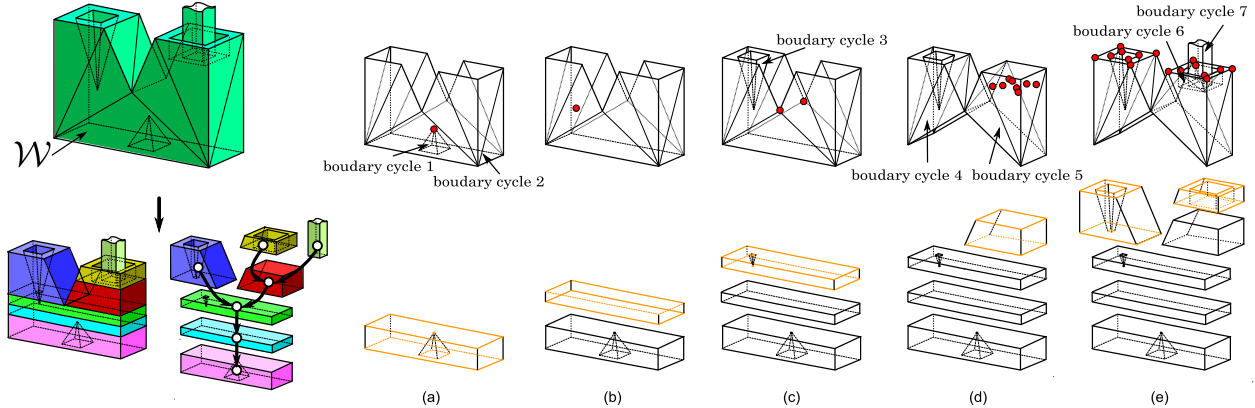


Figure 3.5: Figures (a)–(e) show how pools are constructed according to generation, completion, and updating of boundary cycles in general pool segmentation. We segment \mathcal{W} into pools where the topology of slice polygon changes. The top drawing in each subfigure shows the boundary cycles just before processing the indicated vertices; the bottom drawing shows the pools already completed just after processing the indicated vertices. The line segments shown in orange indicate the bottom face and top face of just completed pools. Notice that, for each pool, the bottom face and top face is defined by the triangles in the same set of boundary cycles.

(h)–(k)). When one boundary cycle splits into multiple boundary cycles (Figure 3.4 (g)), we finalize the pool defined by the existing boundary cycle (the purple pool). The portions of the triangles above the z -coordinate of the indicated vertex are trimmed and the set of the corresponding intersection line segments defines the top face of the finalized pool. At the same time, we initialize new pools defined by the new boundary cycles (the yellow and blue pools). The triangles in the boundary cycles after processing the merge/split vertex indicated in the top of subfigure (g) are assigned to the pools. The portions of the triangles below the z -coordinate of the indicated vertex are trimmed; the sets of corresponding intersection line segments define the bottom face of the newly generated pools. When a boundary cycle is completed, we finalize the pool defined by the boundary cycle (Figure 3.4 (l)). In this example, since the intersection between the triangles in the pool and the corresponding sweep plane becomes a point, no triangles are trimmed and the top face consists of a single point.

3.2.2.1 Pool Segmentation, General Case

We now remove the simplifying assumption that each slice polygon is bounded by only one outer slice polygon boundary. In the general case, a pool is defined by one outer boundary cycle and zero or more inner boundary cycles. Unlike in the previous simplified case, the appearance, disappearance, merging, or splitting of slice polygon boundaries does not necessarily lead to the appearance, disappearance, merging, and splitting of the corresponding slice polygons for the general case. For example, in Figure 3.3, the appearance of the

Algorithm 3 SimplifiedCasePoolSegmentation(V)

Input: V : set of vertices in \mathcal{W}
 where $V = \{V(z_1), V(z_2), \dots, V(z_n)\}$ ($z_i < z_j$ if $i < j$)
 // $V(z_i)$ is a set of vertices whose z-coordinate is z_i
for $i = 1$ to n **do**
 $P_{init} \leftarrow \emptyset$
 $P_{final} \leftarrow \emptyset$
 $(G, C, U) \leftarrow \mathbf{ProcessVertices}(V(z_i))$
 for each boundary cycle $b \in U$ **do**
 $b \rightarrow pool \rightarrow T_1 \leftarrow (b \rightarrow pool \rightarrow T_1) \cup (b \rightarrow T_{new}(z_i))$
 end for
 for each boundary cycle $b \in C$ **do**
 $P_{final} \leftarrow P_{final} \cup \mathbf{FinalizePool}(b, z_i)$
 end for
 for each boundary cycle $b \in G$ **do**
 $P_{init} \leftarrow P_{init} \cup \mathbf{InitializePool}(b, z_i)$
 end for
 if $P_{init} \neq \emptyset$ and $P_{final} \neq \emptyset$ **then**
 $\mathbf{ConstructConnectivity}(P_{init}, P_{final})$
 end if
end for

inner slice polygon boundary does not lead to the appearance of a new slice polygon; the appearance of the inner slice polygon boundary just changes the topology of the existing slice polygon. However, to simplify our implementation (as well as the volume computation described below in section 3.3.1), we segment \mathcal{W} into pools whenever a boundary cycle is generated or completed (which is equivalent to saying whenever the topology of a slice polygon changes). As stated in section 3.2, when a slice polygon appears, disappears, merges, or splits, the corresponding slice polygon boundary also appears, disappears, merges, or splits, and thus at least one boundary cycle is generated or completed. Therefore, the modified segmentation rule satisfies our original pool segmentation criteria.

Thus, a pool is initialized and finalized when an outer boundary cycle defining the pool is generated and completed in the same manner as in the simplified case. In addition, given a pool, every time one of the inner boundary cycles defining the pool is generated or completed, we finalize the pool and initialize a new one. Using this segmentation rule, each pool is entirely defined by the same set of boundary cycles, i.e. the set of boundary cycles when the pool is initialized and finalized is the same (although their triangles will have changed if any no-change vertices are on the boundary between the bottom and top face).

Subfigures 3.5 (a)–(e) show the boundary cycles just before processing the indicated vertices and the pools already completed just after processing the indicated vertices. In Figure 3.5 (a), boundary cycle 1, an inner boundary cycle, completes. We finalize the first

pool defined by outer boundary cycle 2 immediately enclosing boundary cycle 1. Since outer boundary cycle 2 is not completed, we initialize a new pool defined by outer boundary cycle 2. In Figure 3.5 (b), boundary cycle 3, an inner boundary cycle, is generated. We associate boundary cycle 3 with boundary cycle 2, its enclosing boundary cycle. Since boundary cycle 2 has already defined a pool not finalized yet, we finalize that pool, and initialize a new pool defined by boundary cycle 2 and boundary cycle 3. In Figure 3.5 (c), boundary cycle 2 splits into boundary cycle 4 and boundary cycle 5. We finalize the existing pool defined by boundary cycle 2. Since inner boundary cycle 3, immediately enclosed by boundary cycle 2, is not completed, we reassociate boundary cycle 3 with boundary cycle 4, which immediately encloses boundary cycle 3 just above the indicated vertices. We initialize a new pool defined by boundary cycle 4 and boundary cycle 3 (immediately enclosed by it), and boundary cycle 5, respectively. In Figure 3.5 (d), boundary cycle 6, an inner boundary cycle, and boundary cycle 7, an outer boundary cycle, are generated. We associate boundary cycle 6 with boundary cycle 5, its enclosing boundary cycle. Since boundary cycle 5 has already defined a pool not finalized yet, we finalize that pool, and initialize a new pool defined by boundary cycle 5 and boundary cycle 6. We also initialize a new pool defined by boundary cycle 7. In Figure 3.5 (e), boundary cycle 4 and boundary cycle 3 (immediately enclosed by it), and boundary cycle 5 and boundary cycle 6 (immediately enclosed by it) are completed. We finalize the pool defined by these boundary cycles.

Now, we give the algorithm to implement this segmentation rule. For a given z where $V(z) \neq \emptyset$, let G_{outer} and G_{inner} be the sets of new outer and inner boundary cycles, respectively, generated at $v \in V(z)$, and C_{outer} and C_{inner} be the sets of existing outer and inner boundary cycles, respectively, completed at $v \in V(z)$. Then, a pool is constructed by the following rules.

For a given z where $V(z) \neq \emptyset$:

1. For each updated boundary cycle $b \in U$, we add new triangles inserted into b at z to the pool defined by b .
2. For each completed outer boundary cycle $b_{outer} \in C_{outer}$, we finalize the pool defined by b_{outer} . We let $inner(b_{outer})$ be the set of inner boundary cycles immediately enclosed by b_{outer} at z^- . For each $b_{inner} \in inner(b_{outer})$, if $b_{inner} \notin C_{inner}$, we add b_{inner} to G_{inner} (Figure 3.5 (c)(e)).
3. For each completed inner boundary cycle $b_{inner} \in C_{inner}$, we let b_{outer} be the outer boundary cycle immediately enclosing b_{inner} at z^- . We dissociate b_{inner} from b_{outer} . If the pool defined by b_{outer} is not finalized, before the dissociation, we finalize the pool and add b_{outer} to G_{outer} (Figure 3.5 (a)).
4. For each newly generated inner boundary cycle $b_{inner} \in G_{inner}$, we find the outer boundary cycle b_{outer} immediately enclosing b_{inner} at z^+ . We associate b_{inner} with b_{outer} . If the pool defined by b_{outer} is not finalized, before the association, we finalize the pool and add b_{outer} to G_{outer} (Figure 3.5 (b)(d)).

5. For each newly generated outer boundary cycle $b_{outer} \in G_{outer}$, we initialize a new pool p defined by b_{outer} and its inner boundary cycles immediately enclosed by b_{outer} at z^+ .
6. Let P_{init} and P_{final} be the sets of pools initialized and finalized at z , respectively. If $P_{init} \neq \emptyset$ and $P_{final} \neq \emptyset$, for $p_i \in P_{init}$ and for $p_f \in P_{final}$, we compare the bottom face of p_i and the top face of p_f . If they overlap, we add a directed edge from the node corresponding to p_i to the node corresponding to p_f in the directed graph.

Algorithm 4 shows the corresponding pseudocode. **ClassifyBoundaryCycle** classifies each newly generated boundary cycle as either an outer boundary cycle or an inner boundary cycle using the method described in 3.2.1.2.

3.3 Predicting Water Trap Regions

After completing the sweep from $z = -\infty$ to $z = +\infty$, the space \mathcal{W} is segmented into pools that are connected to each other if their bottom faces and top faces are overlapping. As we described in section 3.1.3, given a pool, if there is no path from the corresponding node to the node corresponding to the bottommost pool, the pool is a potential water trap region (depending on inflow location). The bottommost pool corresponds to the first pool created during sweeping.

Finding the pools that are potential water trap regions is straightforward. From the node corresponding to the bottommost, we traverse the graph in the opposite direction of the graph edges until we have visited all reachable nodes. The nodes we cannot reach from the node corresponding to the bottommost pool represent potential water trap regions. For the traversal from the bottommost node, we do not have to visit the same node twice; therefore, the time complexity of the procedure is linear with respect to the number of pools.

3.3.1 Quantitative Evaluation of a Part Orientation

Since we can compute the volume of water each pool can hold, we can also quantitatively evaluate a given part orientation by summing the volumes of pools that are determined to be water trap regions.

The volume of an arbitrary polyhedron defined by a set of triangles T can be computed using equation (3.1), where each triangle $t \in T$ is defined by the points v_{t1} , v_{t2} , v_{t3} ordered counterclockwise when viewed from the exterior of the polyhedron.

$$V = \frac{1}{6} \sum_{t \in T} ((v_{t1} \times v_{t2}) \cdot v_{t3}) \quad (3.1)$$

Our pools are bounded on the side by original and trimmed triangles from \mathcal{W} and on the bottom and top by 2D polygons, possibly with holes. Given a pool defined by n boundary cycles, we let the vertices constituting the i -th closed polygonal chain of the bottom face be $l_{ij}(1 \leq j \leq p_i)$, and the vertices constituting the i -th closed polygonal chain of the top

Algorithm 4 GeneralCasePoolSegmentation(V)

Input: V : set of vertices in \mathcal{W}
 where $V = \{V(z_1), V(z_2), \dots, V(z_n)\}$ ($z_i < z_j$ if $i < j$)
 // $V(z_i)$ is a set of vertices whose z-coordinate is z_i
for $i = 1$ to n **do**
 $(G, C, U) \leftarrow \text{ProcessVertices}(V(z_i))$
 $(G_{inner}, G_{outer}) \leftarrow \text{ClassifyBoundaryCycle}(G, z^+)$
 $C_{outer} \leftarrow$ set of outer boundary cycles in C
 $C_{inner} \leftarrow$ set of inner boundary cycles in C
 $P_{init} \leftarrow \emptyset$
 $P_{final} \leftarrow \emptyset$
 for each $b \in U$ **do**
 // suppose b is the j -th boundary cycle of $b \rightarrow pool$
 $b \rightarrow pool \rightarrow T_j \leftarrow (b \rightarrow pool \rightarrow T_j) \cup (b \rightarrow T_{new}(z_i))$
 end for
 for each $b_{outer} \in C_{outer}$ **do**
 $P_{final} \leftarrow P_{final} \cup \text{FinalizePool}(b_{outer}, z_i)$
 for each $b_{inner} \in inner(b_{outer})$ **do**
 if $b_{inner} \notin C_{inner}$ **then**
 $G_{inner} \leftarrow G_{inner} \cup b_{inner}$
 end if
 end for
 end for
 for each $b_{inner} \in C_{inner}$ **do**
 $b_{outer} \leftarrow$ outer boundary cycle immediately enclosing b_{inner} at z^-
 if $b_{outer} \rightarrow pool \neq \text{nil}$ **then**
 $P_{final} \leftarrow P_{final} \cup \text{FinalizePool}(b_{outer}, z_i)$
 $G_{outer} \leftarrow G_{outer} \cup b_{outer}$
 end if
 Dissociate b_{inner} from b_{outer}
 end for
 for each $b_{inner} \in G_{inner}$ **do**
 $b_{outer} \leftarrow$ outer boundary cycle immediately enclosing b_{inner} at z^+
 if $b_{outer} \rightarrow pool \neq \text{nil}$ **then**
 $P_{final} \leftarrow P_{final} \cup \text{FinalizePool}(b_{outer}, z_i)$
 $G_{outer} \leftarrow G_{outer} \cup b_{outer}$
 end if
 Associate b_{inner} with b_{outer}
 end for
 for each $b_{outer} \in G_{outer}$ **do**
 $P_{init} \leftarrow P_{init} \cup \text{InitializePool}(b_{outer}, z_i)$
 end for
 if $P_{init} \neq \emptyset$ and $P_{final} \neq \emptyset$ **then**
 ConstructConnectivity(P_{init}, P_{final})
 end if
end for

Table 3.1: Timing data for pool segmentation and directed graph construction on various models.

	part 1	cylinder head1 (orientation 1)	cylinder head1 (orientation 2)	cylinder head2 (orientation 1)	cylinder head2 (orientation 2)
# vertices	1,294	104,310	104,310	144,546	144,546
# pools	77	824	706	876	1,552
time (sec.)	0.07	0.827	2.661	2.005	3.855

face be $u_{ij}(1 \leq j \leq q_i)$, with the vertices of the outer and inner slice polygon boundaries enumerated in counterclockwise and clockwise order, respectively, when viewed from the exterior of the pool.

Then the volume of this pool can be computed using equation (3.2):

$$\begin{aligned}
 V_{pool} = & \frac{1}{6} \left\{ \sum_{t \in pool} ((v_{t1} \times v_{t2}) \cdot v_{t3}) + \sum_{i=1}^n \sum_{j=2}^{p_i-1} ((l_{i1} \times l_{ij}) \cdot l_{ij+1}) \right. \\
 & \left. + \sum_{i=1}^n \sum_{j=2}^{q_i-1} ((u_{i1} \times u_{ij}) \cdot u_{ij+1}) \right\} \quad (3.2)
 \end{aligned}$$

3.4 Results

Figure 3.6 shows the result of our segmentation and the identified water trap regions using our method on an industrial cylinder head model.

Table 3.1 shows timing data for pool segmentation and directed graph construction on the part shown in Figure 3.6 and other models. The timing was performed on a computer with a 2.66 GHz Intel Core i7 CPU with 4 GB of memory. Running times increase with the number of vertices but not necessarily with the number of generated pools. In our experience, the complexity of the geometry of each pool matters highly. For both cylinder head models, timing is shown for the same part in different orientations. Note that the same part in a different orientation can have twice as many pools, and also three times the running time for pool segmentation and graph construction. Once we obtain segmented pools and the corresponding directed graph, our algorithm to identify water trap regions for a given inflow location takes less than a millisecond, fast enough for even the most complex models.

3.5 Complexity Analysis

Letting N be the number of vertices in \mathcal{W} , we analyze the scalability of our algorithm.

Before starting the segmentation, we first sort the vertices in order of ascending z-coordinate. This takes $O(N \log N)$ time.

Next, we analyze the complexity of our pool segmentation algorithm based on Algorithm 4. For each iteration in the main loop, we process vertices whose z-coordinate is z_i . We let k_i be the number of edges in \mathcal{W} intersecting $p_{\text{sweep}}(z_i)$. The running time of performing **ProcessVertices**, **InitializePool**, and **FinalizePool** is $O(k_i)$, respectively. The running time of performing **ClassifyBoundaryCycle** is also $O(k_i)$ because the number of triangles intersecting $p_{\text{sweep}}(z_i)$ is equal to k_i . In **ConstructConnectivity**, we perform a 2D polygon intersection test. We can perform an intersection test of two polygons defined by m points by performing the equivalent Boolean operation running in $O((m+r) \log m)$ time where r is the number of points defining the intersection of two polygons [de Berg et al. 2008]. In practice, we approximately perform a 2D polygon intersection test using graphics hardware. Given two polygons, we render them and, if there is a pixel covered by both the polygons, we determine that the two polygons are intersecting. In this approach, the performance is dominated by the number of vertices defining the polygons. In our case, the vertices of the polygons are defined by the edges intersecting $p_{\text{sweep}}(z_i)$; therefore, the running time of the intersection test can be stated as $O(k_i)$. Since the number of iterations in the main loop is n , the complexity of our pool segmentation algorithm is $O(\sum_{i=1}^n k_i)$. Letting $\bar{k} = (\sum_{i=1}^n k_i)/n$ i.e. the average of k_i ($1 \leq i \leq n$), the complexity can be represented as $O(\bar{k}n)$. Since the number of iterations n is bounded by N , we can now state that the complexity of our pool segmentation is $O(\bar{k}N)$. Since \bar{k} is $O(N)$, the overall complexity can be $O(N^2)$ in the worst case; however, \bar{k} is usually much smaller than N .

Once we segment \mathcal{W} into pools, we can find potential water trap regions in linear time with respect to the number of pools as discussed in section 3.3.

3.6 Conclusion

In this chapter, we proposed a new pool segmentation data structure and algorithm based on topological changes of 2D slices with respect to gravity direction. We showed that we can predict potential water trap regions of a given geometry by analyzing the directed graph based on the segmented pools.

We would like to suggest utilizing this data structure to accelerate physics-based simulation of fluid flow inside mechanical parts with complex geometry. Although recent advances in CPUs and GPUs make real-time fluid flow simulation possible in a simple computational domain, performing such simulation in a complex domain in real-time is still challenging. We believe that our pool segmentation data structure may also prove useful for other applications analyzing fluid flow inside complex geometry. For further discussion on this topic, please refer to chapter 6.

The directed graph our algorithm constructs has by nature more information than the corresponding Reeb graph of a 3-manifold with boundary with respect to the height function. While our segmentation rule takes into account all the topology changes of 2D slices, the

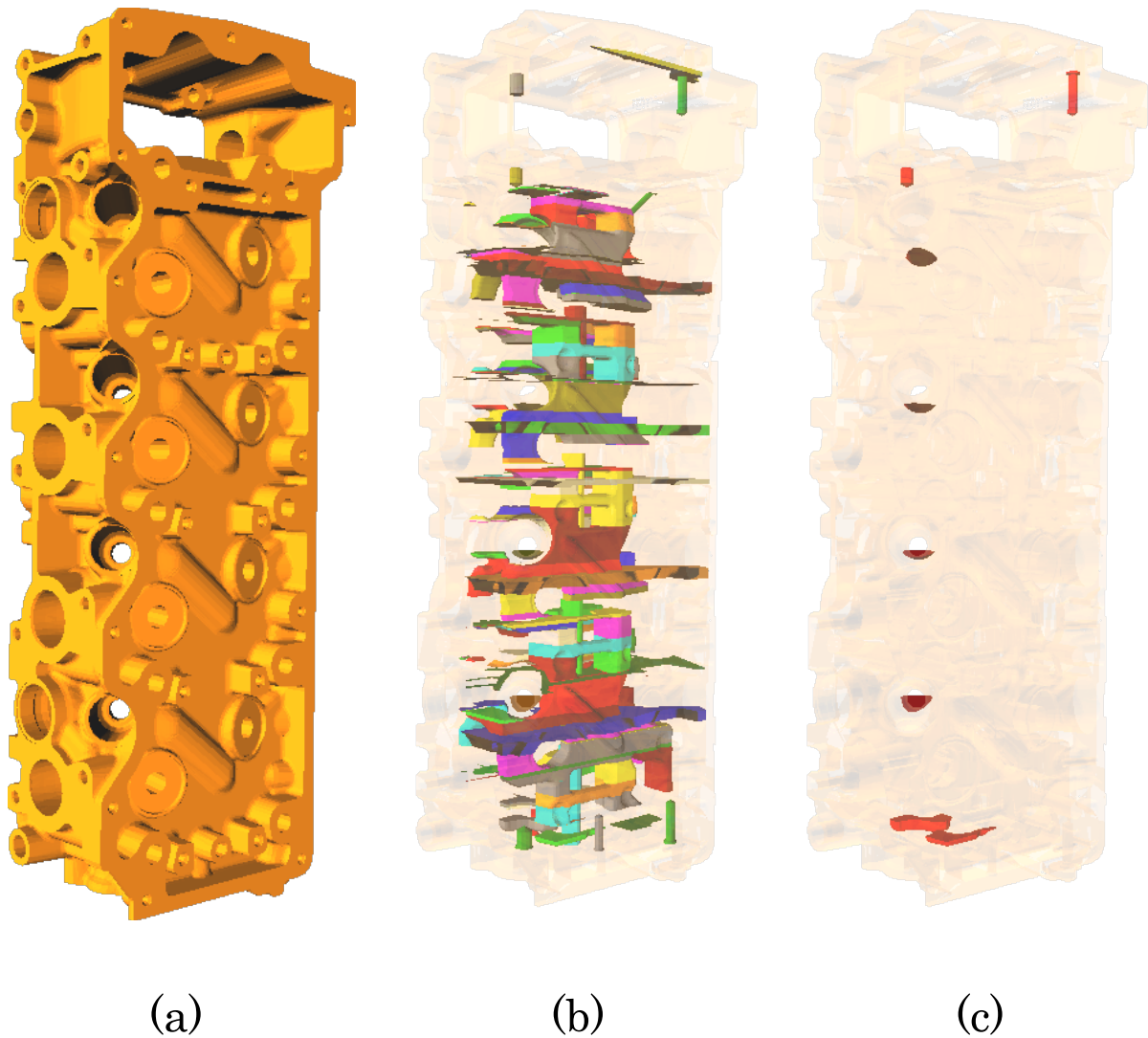


Figure 3.6: We applied our algorithm to a mechanical workpiece shown in (a). (b) Pool segmentation of the workpiece. Pools are assigned random colors. (c) Water trap regions of the workpiece. Note that we do not show the pools bounded by triangles which come from the corresponding bounding box for visualization purpose in this figure.

Reeb graph does not capture them; the Reeb graph only captures the merging and splitting of connected components. Given a geometry and our directed graph, we can easily obtain the corresponding Reeb graph by deleting nodes that have only one node connected above and one node connected below, respectively.

Chapter 4

Testing a Rotation Axis to Drain a 3D Workpiece

In this chapter, given a triangular mesh defining the geometry of a 3D workpiece filled with water, we propose an algorithm to test whether, for an arbitrary given axis, the workpiece will be completely drained under gravity when the rotation axis is set parallel to the ground and the workpiece is rotated around the axis. Observing that all water traps contain a concave vertex, we solve our problem by constructing and analyzing a directed “*draining graph*” whose nodes correspond to concave vertices of the geometry and whose edges are set according to the transition of trapped water when we rotate the workpiece around the given axis. Our algorithm to test whether or not a given rotation axis drains the workpiece outputs a result in about a second for models with more than 100,000 triangles after a few seconds of preprocessing.

The proposed algorithm in this chapter is a first step toward our ultimate goal: *finding* a rotation axis for a given workpiece geometry.

4.1 Assumptions and a Key Observation

We assume that we can approximate a volume of water by a set of water particles whose viscosity is negligibly small. We also assume that the rotation is slow enough that the water particles reach equilibrium for each orientation through which we rotate, and that the particles move only under the effect of gravity (assuming that any other phenomena such as friction or centrifugal force are negligible).

We define a *water trap* in a particular orientation as a connected volume of undrained water, which we approximate by a set of water particles directly or indirectly touching each other (Figure 4.1).

The key observation for our problem is that, for each water trap, there is always at least one concave vertex of the input mesh such that some of its incident edges and faces are touching water particles constituting the water trap. Based on this observation, our goal

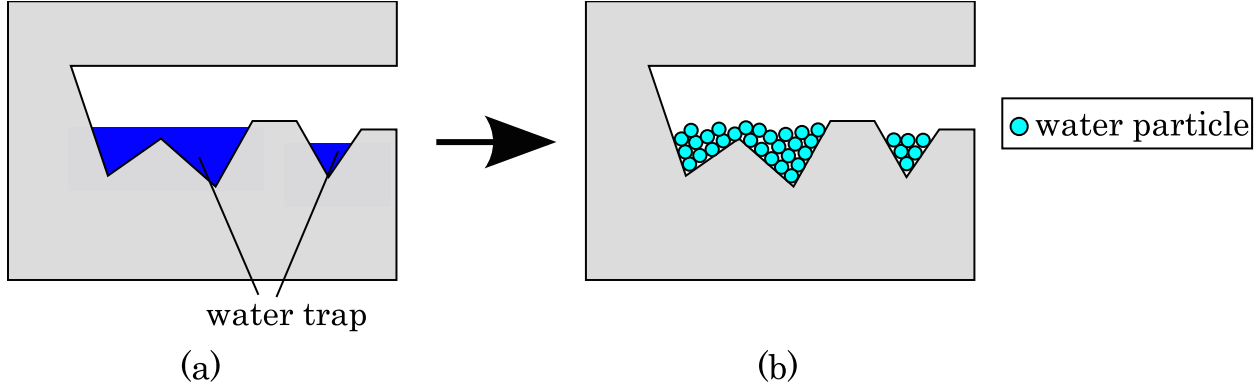


Figure 4.1: We assume that we can approximate a volume of water (shown in (a) for a 2D example) by a set of water particles (shown in (b)). A water trap is a set of water particles directly or indirectly touching each other and some of which are touching the input geometry. In this example, two water traps are formed.

is to drain all the concave vertices of an input mesh since this is equivalent to draining all water traps from all voids of the workpiece.

In the next section, we describe an overview of our approach, going through a 2D example to introduce our directed *draining graph* method. Then, in the following sections, we describe how we actually construct and analyze the draining graph for a 3D geometry and arbitrary 3D rotation axis.

4.2 Approach and Theory

To begin, we discuss a simplified case using a 2D example.

4.2.1 Simplified case: each water trap is represented by a single water particle

First, we consider the case that each water trap consists of only one water particle. Recall that a water trap can only be formed at a concave vertex.

For each concave vertex v , we consider gravity directions such that, if a water particle is at v , it will be trapped. In the 2D case, any gravity direction can be described as a point on the *Gaussian circle* (a circle whose radius is one and center is at the origin). When we rotate a workpiece, the gravity direction moves relative to the workpiece along the Gaussian circle. For each concave vertex v , we define a space T_v on the Gaussian circle consisting of gravity directions such that, if a water particle is at v , it will be trapped. Figure 4.2 shows a specific example. In the 2D case, each of the two gravity directions $g_{v(CCW)}$ and $g_{v(CW)}$ bounding T_v are orthogonal to the two edges incident to v . We define $g_{v(CCW)}^*$ as a point on the Gaussian circle that is not in T_v and is closest to $g_{v(CCW)}$. In a similar manner, we define $g_{v(CW)}^*$ as

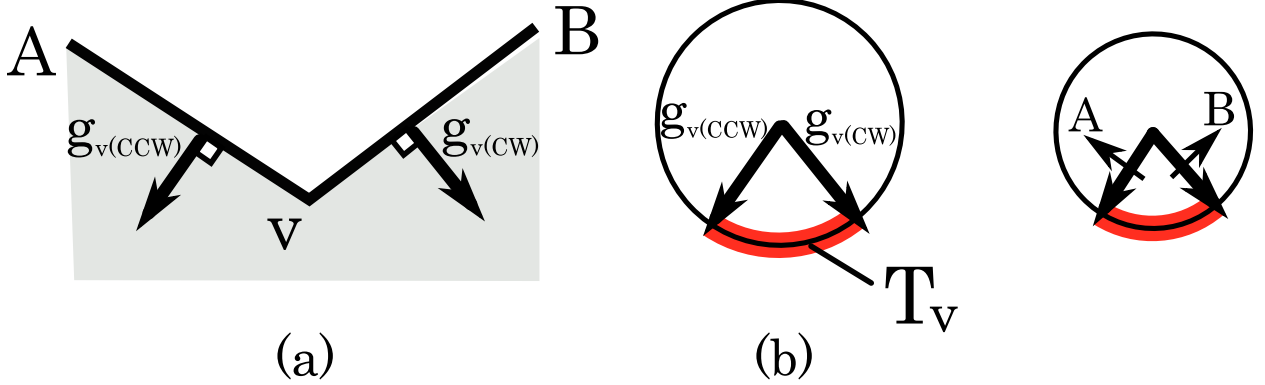


Figure 4.2: (a): Concave vertex v (b): The diagram showing T_v of v .

a point on the Gaussian circle that is not in T_v and is closest to $g_{v(CW)}$. For a workpiece orientation with corresponding gravity direction relative to the workpiece currently in T_v , when the workpiece rotates far enough that the gravity direction relative to the workpiece coincides with $g_{v(CCW)}^*$ (respectively, $g_{v(CW)}^*$), the trapped water particle leaves v and moves along the edge towards A (respectively, B).

We construct a directed *draining graph* whose nodes correspond to the concave vertices. Each node has two kinds of outgoing edges, corresponding to clockwise and counterclockwise rotation, that point to the nodes representing the concave vertices where the trapped water will ultimately settle when the workpiece is rotated clockwise and gravity coincides with $g_{v(CW)}^*$ or counterclockwise and gravity coincides with $g_{v(CCW)}^*$. If the water particle trapped at a vertex exits the workpiece once it is rotated so that gravity coincides with $g_{v(CW)}^*$ or $g_{v(CCW)}^*$, the corresponding edge is set to point to a node labeled “out” representing the workpiece exterior. An example of a draining graph for a 2D geometry is shown in Figure 4.3.

The draining graph is constructed as follows. For each concave vertex, we initialize a corresponding node in the draining graph. Then, we compute the two gravity directions when a water particle trapped at the concave vertex leaves it under clockwise and counterclockwise rotation. These gravity directions (the bounds of T_v) are shown in a diagram next to each node in Figure 4.3 (a). Finally, we trace the path of a trapped water particle under both of these gravity direction to determine in what concave vertex it settles for each, adding a graph edge labeled as *CW* or *CCW* that points to the corresponding node. Figure 4.3 (b)–(g) show the paths a water particle takes under gravity from each concave vertex for the geometry shown in Figure 4.3 (a).

The destination is not necessarily unique since there may be multiple possible paths a water particle takes under gravity. Figure 4.3 (e) shows one such example. A water particle leaving concave vertex D with $g_{D(CW)}^*$ may settle at concave vertex E or exit the workpiece. In this case, we assume that a particle splits into two particles.

For each concave vertex, if there is some path consisting of edges with the same direction

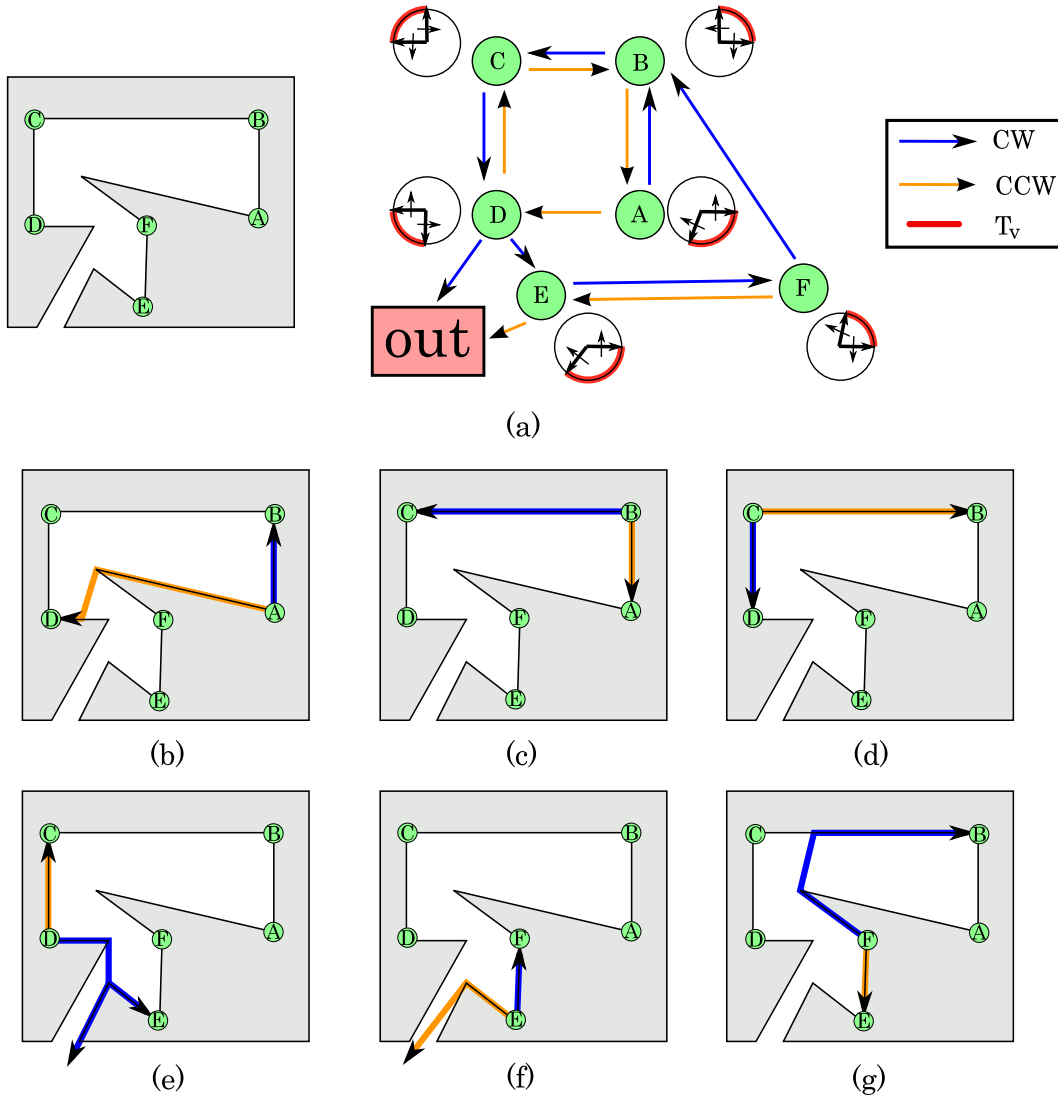


Figure 4.3: (a) A sample geometry in 2D and the corresponding draining graph. The diagram next to each graph node shows the two gravity directions $g_{v(CW)}^*$ and $g_{v(CCW)}^*$ that let a water particle trapped at the corresponding concave vertex leave for the other concave vertices. For each node, when a current gravity direction is in T_v , if a water particle is at the corresponding concave vertex, it will be trapped. (b)–(g) Paths a water particle takes from each concave vertex under $g_{v(CW)}^*$ (blue) and $g_{v(CCW)}^*$ (orange).

label from the corresponding node to the node labeled as “out,” then we can eventually drain the water particle trapped at that concave vertex through concave vertices corresponding to the nodes along the path by rotating in the given direction. For example, suppose a water particle is trapped at concave vertex A , there is a path “ $A \rightarrow B \rightarrow C \rightarrow D \rightarrow out$ ” in Figure 4.3 corresponding to the draining sequence shown in the top row of Figure 4.4. Since there

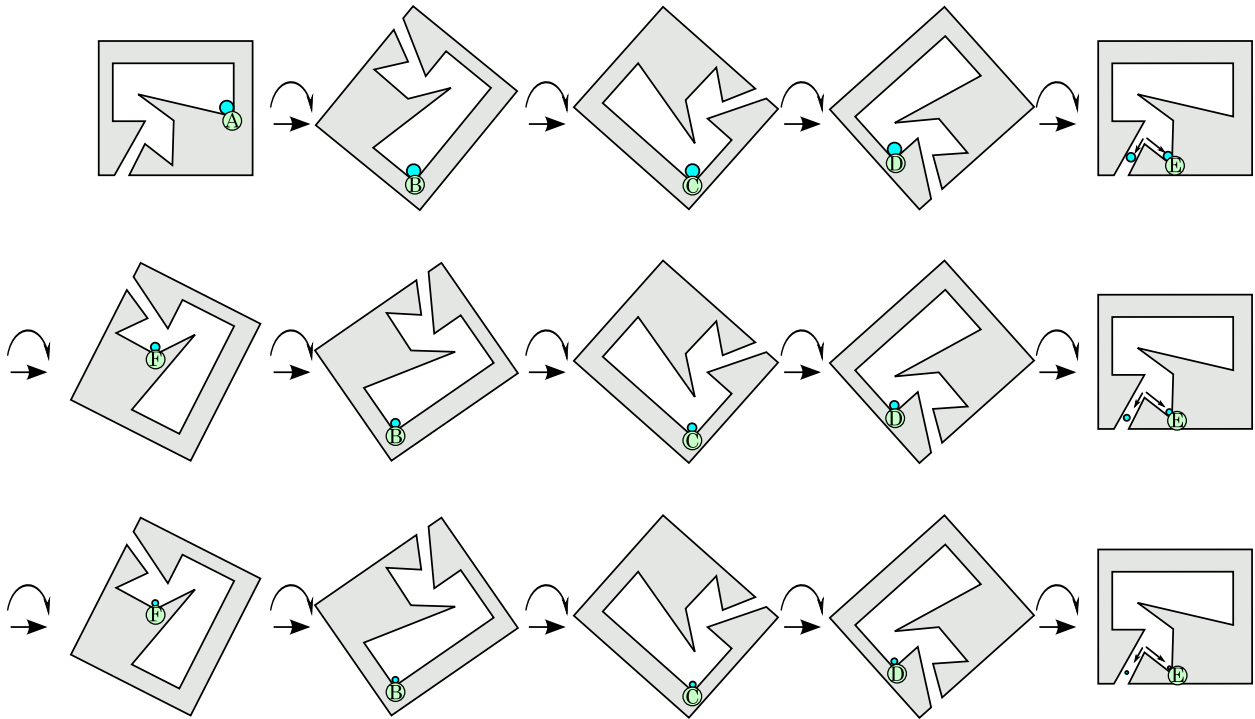


Figure 4.4: (a) The transition and draining of a water particle trapped at concave vertex A by clockwise rotation. As we continue to rotate the geometry, the volume of the trapped particle gradually decreases and becomes negligibly small.

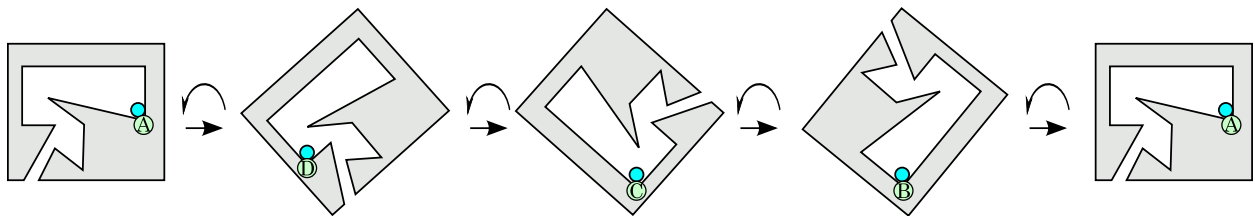


Figure 4.5: We cannot drain a water particle trapped at concave vertex A by counterclockwise rotation.

is an edge from D not only to “out” but also to E , the water particle splits into two smaller particles and only one of them will be drained through the sequence. But as we can see in Figure 4.3, there is a path from E to “out” as well (“ $E \rightarrow F \rightarrow B \rightarrow C \rightarrow D \rightarrow out$ ”); therefore, the other smaller particle going to E will come back to D , split into two further smaller particles, and one of them will be drained. For each 360-degree rotation, the volume of the remaining trapped water particle decreases (Figure 4.4). When the volume of the remaining particle becomes negligibly small, we deem that the trapped water particle is drained. As we have seen through the example, as long as there is a path from each node to

the “out” node in the draining graph, the volume of the remaining water particles gradually decreases and will be drained eventually after rotating enough times.

On the other hand, if there are nodes that do not have a consistently labeled path to “out,” we can never drain their trapped water particles. For example, for a water particle trapped at A in the geometry shown in Figure 4.3, we cannot drain the water particle by counterclockwise rotation, because it just returns to A after each rotation, as shown in Figure 4.5. This also holds for counterclockwise rotation when a water particle is trapped at B , C , or D . In the draining graph, there is no CCW path from the nodes corresponding to any of these vertices to “out.”

There is also a path “ $A \rightarrow D \rightarrow out$.” But we cannot drain using this path because it corresponds to counterclockwise rotation from A to D and clockwise rotation from D to out . This violates our restriction that we can rotate around an axis in one direction only.

4.2.2 General case: each water trap is represented by a set of water particles

In the previous subsection, we took as our premise the case that each water trap consists of only one water particle, and provided the approach to solve the corresponding draining problem. We now show that if a solution exists for this case, it is also a solution for the general draining problem; that is, the case that each water trap consists of a set of water particles.

Suppose a water trap is currently formed at concave vertex v . For the general draining problem, after v is drained, not all the water particles constituting the original water trap will necessarily form a new water trap at the same concave vertex (see Figure 4.6); however, the key observation is that the last water particle to leave the concave vertex (we call this last particle the *core particle* of the water trap) moves in the same manner as a water particle approximating the water trap by only one particle. For example, suppose that a water trap is currently formed at a given concave vertex v as shown in Figure 4.6(a). Figure 4.6(b) shows draining when we approximate a water trap by only one particle and (c) shows draining when we approximate a water trap by a set of particles (general case). In the general case, when we start to rotate an input geometry, water particles constituting the water trap start to leave v and form another water trap at a different concave vertex (or may exit the geometry). As we continue to rotate, when one of the edges incident to v becomes perpendicular to the gravity direction (i.e. parallel to the ground), the core particle of the water trap leaves v . As shown in Figure 4.6 (c), at the point when v is completely drained, water particles constituting the original water trap may constitute different water traps after a rotation; however, the core particle moves in the same manner as a water particle approximating a water trap by only one particle.

Now we show that the approach using the draining graph for the single-particle case works for general draining problems, too, assuming we rotate enough times. To see this, let us consider a simple example of a draining graph with only 4 nodes, A , B , C , and out , with

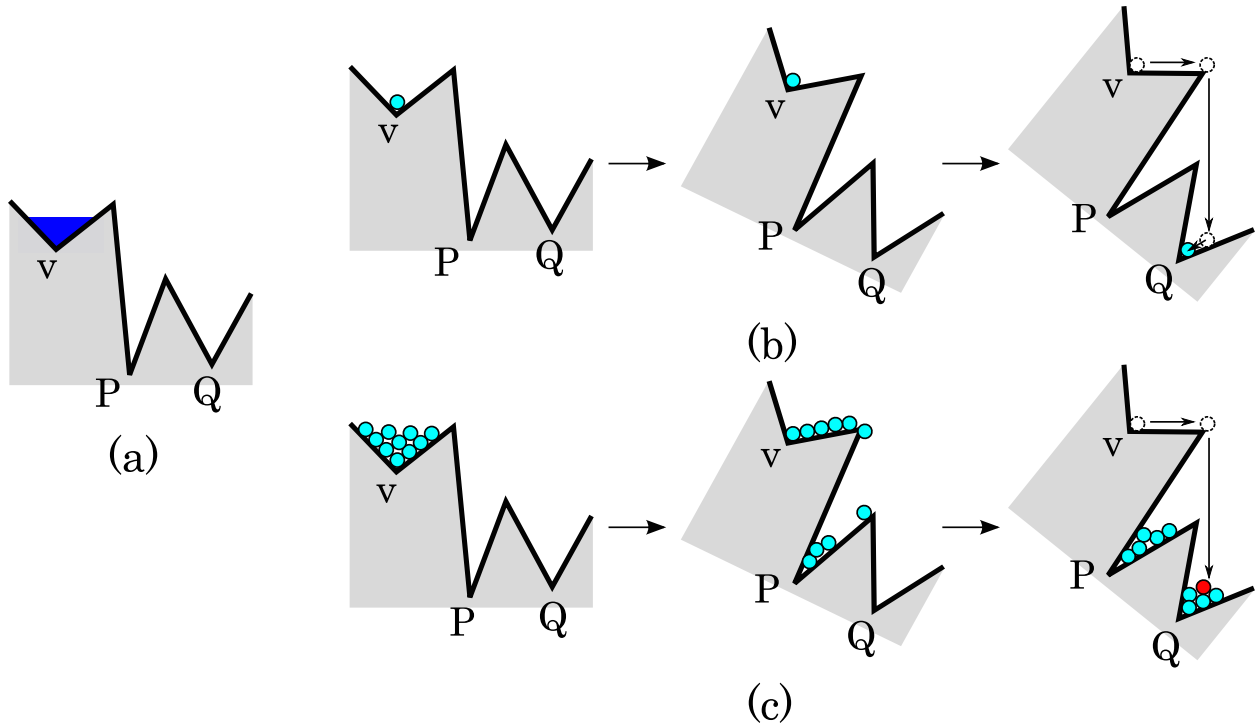


Figure 4.6: (a) A water trap is formed at concave vertex v . (b) The movement of a water particle when we approximate the water trap by only one particle. After v is drained, a new water trap is formed at concave vertex Q . (c) The movement of water particles when we approximate the water trap by a set of particles. After v is drained, new water traps are formed at P and Q . The *core particle* shown in red settles at a water trap at the same concave vertex where the particle in case (b) settles.

three CW directed edges “ $C \rightarrow B \rightarrow A \rightarrow out$ ” such that a single water particle trapped at C is drained through B and A with one 360-degree rotation. Now we consider the general case for this example. For each 360-degree clockwise rotation, at least a core particle at A is drained (note that there is no guarantee that all the water particles at A exit the geometry). If there are remaining water particles (water traps), each of them exists at B or C because a water trap is always formed at a concave vertex. There is a path from C to B ; therefore, a core particle at C goes to B if a water trap is formed at C . There is a path from B to A ; therefore, a core particle at B goes to A if a water trap is formed at B . This implies that as long as there are remaining water particles, one of them must become a core particle at A and be drained in each 360-degree rotation. The number of trapped water particles never increases; therefore, as long as there are CW or CCW paths from all nodes to out , eventually all the particles will be drained. This holds no matter how complicated the draining graph becomes.

4.3 Graph Construction

In the previous section, we have shown that we can solve the general draining problem by considering the case that each water trap consists of a single water particle and considering the corresponding transitions using a draining graph. In this section, given a 3D geometric model and arbitrary 3D rotation axis as input, we explain how we construct the corresponding draining graph.

4.3.1 Graph Nodes

The first step in constructing a draining graph is to determine its nodes, that is, to find the concave vertices. Although water traps in 3D (unlike in 2D) may also contain concave edges, for the concave edge to hold water, one of its endpoints must also be a concave vertex. Therefore, it is still sufficient to consider only the draining of concave vertices, since draining all concave vertices will drain all water traps.

We define a concave vertex for a 3D geometry as follows. Given a vertex v , letting $valence(v)$ be its valence, we check if there is a unit vector d such that, for all the adjacent vertices w_i of v ($i = 1, 2, \dots, valence(v)$), $(w_i - v) \cdot d < 0$ (i.e. v is a locally extreme vertex). If there is such a d and a point $p = v + \epsilon d$ (ϵ is a positive infinitesimal number) is inside of the given geometry, v is a concave vertex. Otherwise, v is not a concave vertex.

4.3.2 Graph Edges

Edges of a draining graph are set according to the transitions of water particles when the geometry is rotated around a given rotation axis. Let V_c be the set of concave vertices. First, for each concave vertex $v \in V_c$, we describe all gravity directions such that a water particle could be trapped at v . Then, we explain how to find the two gravity directions ($g_{v(CW)}^*$ and $g_{v(CCW)}^*$) at which a trapped water particle at v flows out when rotating clockwise or counterclockwise around the given axis. After finding these two gravity directions, for each of them, we find the concave vertices into which water particles flowing out will settle by tracing the particle's path along geometric features (vertices, edges, and triangles).

4.3.2.1 Gravity directions causing a water trap at a concave vertex

In this subsection, we describe all gravity directions such that a water particle may be trapped at v , representing the gravity directions as points on the *Gaussian sphere* (a sphere whose radius is one and center is at the origin).

For each concave vertex $v \in V_c$, let w_i be a member of the set of vertices adjacent to v and let e_i be the vector from v to w_i (i.e. $e_i = w_i - v$) (see Figure 4.7(a)). For each e_i , we define a half-space H_i of directions on the Gaussian sphere $H_i = \{p \mid e_i \cdot p \leq 0, \|p\| = 1\}$. Figure 4.7(b) shows a specific example. A gravity direction not in H_i does not cause a water trap at v . On the other hand, a gravity direction in H_i drags a water particle in the direction

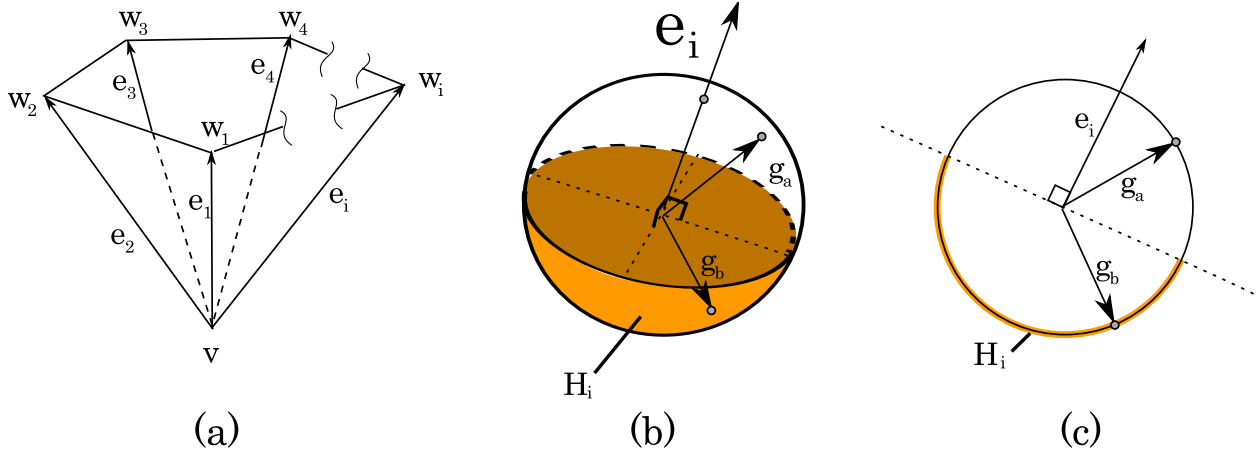


Figure 4.7: (a) Concave vertex v . (b) e_i and the corresponding H_i . Gravity direction g_a never causes a water trap at v , but g_b may cause a water trap at v . (c) The cross section of (b) including g_a and g_b .

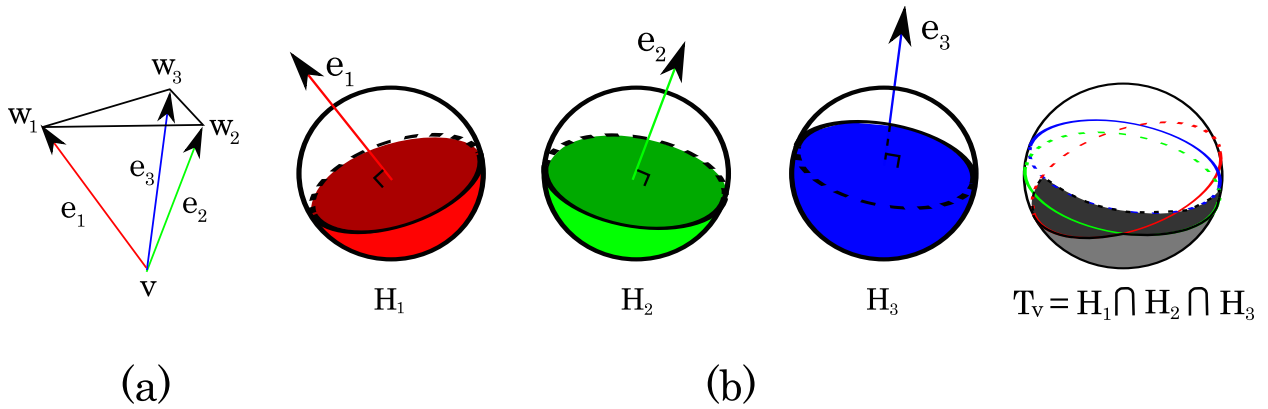


Figure 4.8: (a) Concave vertex v . (b) The corresponding H_i and T_v .

from w_i to v and may cause a water trap at v . For example, in Figure 4.7(b) and (c), the gravity direction g_a never causes a water trap at v , but g_b may cause a water trap at v .

We define the space T_v in 3D as $T_v = \bigcap_i H_i$ (see Figure 5.2). Then, a gravity direction g in T_v potentially causes a water trap at v . On the other hand, since for the complement of T_v , $\overline{T_v}$, we have $\overline{T_v} = \bigcap_i \overline{H_i} = \bigcup_i \overline{H_i}$, therefore g in at least one of $\overline{H_i}$ does not cause a water trap at v .

To construct the draining graph, we need to determine, for each concave vertex $v \in V_c$, in which gravity directions the currently trapped water particle at v flows out when rotating clockwise and counterclockwise around the given axis. These gravity directions correspond to points in $\overline{T_v}$ adjacent to the boundary of T_v .

To find these gravity directions, given a rotation axis, we always choose the coordinate

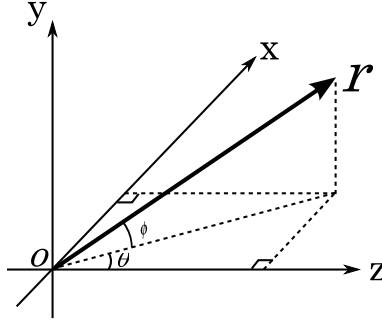


Figure 4.9: We describe any rotation axis relative to workpiece geometry $r = (r_x, r_y, r_z)$ by two variables θ ($0^\circ \leq \theta < 360^\circ$) and ϕ ($0^\circ \leq \phi \leq 90^\circ$) where θ is the azimuthal angle in the xz-plane from the z-axis and ϕ is the polar angle from the xz-plane.

system such that the rotation axis coincides with the z-axis. Then, possible gravity directions are confined in the xy-plane because a gravity direction and the rotation axis are always orthogonal. In this configuration, any gravity direction g can be expressed as a point on a unit circle in the xy-plane with center $(0, 0)$ (i.e. $x^2 + y^2 = 1$). This xy-plane Gaussian circle is the intersection between the Gaussian sphere and the xy-plane.

We can describe any rotation axis relative to workpiece geometry $r = (r_x, r_y, r_z)$ by two variables θ ($0^\circ \leq \theta < 360^\circ$) and ϕ ($0^\circ \leq \phi \leq 90^\circ$) where θ is the azimuthal angle in the xz-plane from the z-axis and ϕ is the polar angle from the xz-plane as shown in Figure 5.11. Using these parameters, the components of r can be expressed as $r_x = \cos \phi \sin \theta$, $r_y = \sin \phi$, and $r_z = \cos \phi \cos \theta$. Then, by multiplying each vertex by the matrix R where

$$R = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ -\sin \theta \sin \phi & \cos \phi & -\cos \theta \sin \phi \\ \sin \theta \cos \phi & \sin \phi & \cos \theta \cos \phi \end{bmatrix},$$

we can set the coordinate system such that a given rotation axis coincides with the z-axis.

Rotating the input geometry around the rotation axis is equivalent to fixing the geometry and moving the gravity direction on the xy-plane Gaussian circle. Given $v \in V_c$, suppose that a gravity direction g is currently in T_v and a water particle is trapped at v . As we move g on the Gaussian circle, when g passes through the boundary of T_v , the trapped water particle at v flows out. If T_v does not intersect with the xy-plane, water is never trapped at v with the given rotation axis.

As shown in Figure 5.2(b), T_v is bounded by a set of great circular arcs on the Gaussian sphere. If T_v is intersected by the xy-plane, it intersects its boundary at two points because T_v is convex. Since $T_v = \bigcap_i H_i$, each of the arcs is defined by the boundary of an H_i .

For the actual calculation to find the gravity directions where trapped water flows out, we do not have to construct T_v in its entirety since the gravity directions are confined in the xy-plane; constructing the portion of T_v intersecting with the xy-plane is sufficient. Call this

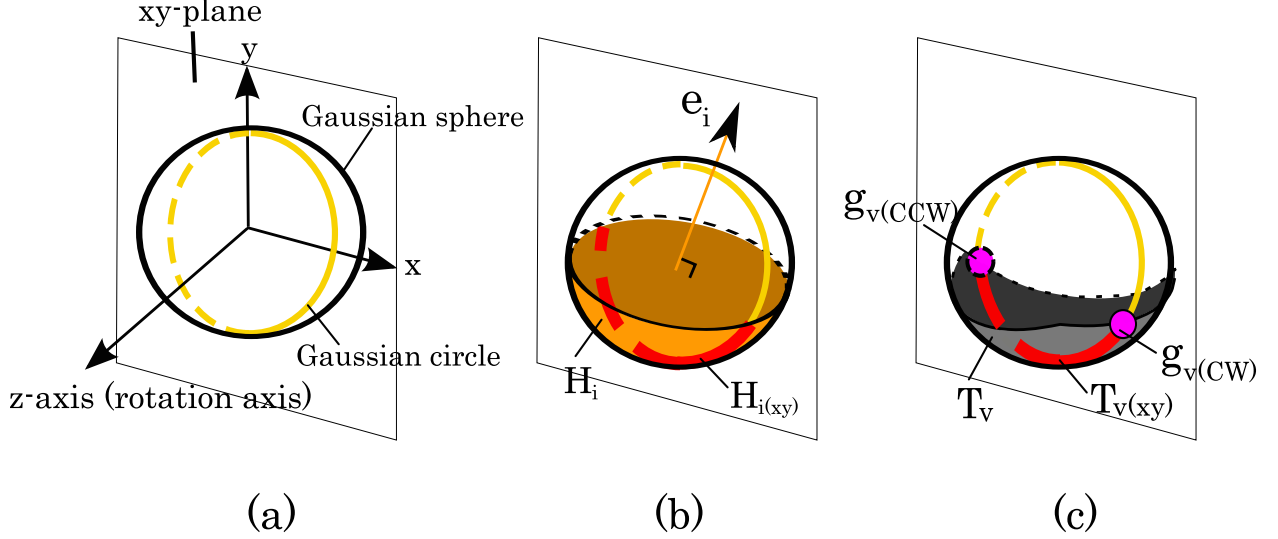


Figure 4.10: (a) The relationship between the Gaussian sphere and the xy-plane Gaussian circle. (b) The relationship between H_i and $H_{i(xy)}$. (c) The relationship between T_v and $T_{v(xy)}$.

portion of T_v defined on the xy-plane Gaussian circle $T_{v(xy)}$ (see Figure 4.10). Each of the boundary points of $T_{v(xy)}$ is defined by the intersection between the xy-plane Gaussian circle and the boundary of one of the H_i because $T_{v(xy)} = \bigcap_i H_{i(xy)}$, where $H_{i(xy)}$ is the intersection between H_i and the xy-plane¹. Appendix B.1 describes how to compute the boundary of $H_{i(xy)}$.

As shown in Figure 4.10(c) and Figure 4.11(a), we let $g_v(CW)$ be the point on the Gaussian circle bounding $T_{v(xy)}$ rotating clockwise (when seen from $+\infty$ on the z-axis – the rotation axis) and $g_v(CCW)$ the point on the Gaussian circle bounding $T_{v(xy)}$ rotating counterclockwise. We compute the boundary points of $T_{v(xy)}$, that is, $g_v(CW)$ and $g_v(CCW)$, incrementally as follows.

Initially, $g_v(CW)$ and $g_v(CCW)$ are set to the two corresponding boundary points of $H_{1(xy)}$. Then, for each i ($i = 2, 3, \dots, valence(v)$), if necessary we update $g_v(CW)$ and $g_v(CCW)$, that is, the boundaries of $T_{v(xy)}$ as follows. For each i , if neither of the $g_v(CW)$ or $g_v(CCW)$ calculated thus far are in $H_{i(xy)}$, $T_{v(xy)}$ is empty (Figure 4.11 (b)). On the other hand, if both $g_v(CW)$ and $g_v(CCW)$ are in $H_{i(xy)}$, we do not have to update $T_{v(xy)}$ (Figure 4.11 (c)). When one of $g_v(CW)$ and $g_v(CCW)$ is not in $H_{i(xy)}$, one of the boundary points of $H_{i(xy)}$ is in $T_{v(xy)}$ (let this be r). If $g_v(CW)$ is not in $H_{i(xy)}$, we set $g_v(CW)$ to r (Figure 4.11 (d)). If $g_v(CCW)$ is not in $H_{i(xy)}$, we set $g_v(CCW)$ to r (Figure 4.11 (e)). After performing this update for each e_i ($i = 2, 3, \dots, valence(v)$), $g_v(CW)$ and $g_v(CCW)$ will be the points bounding $T_{v(xy)}$. If $T_{v(xy)} = \emptyset$, we exclude v from V_c since water is never trapped at v with the given rotation

¹ $T_{v(xy)} = T_v \cap xy = (\bigcap_i H_i) \cap xy = \bigcap_i (H_i \cap xy) = \bigcap_i H_{i(xy)}$
 where xy is the xy-plane.

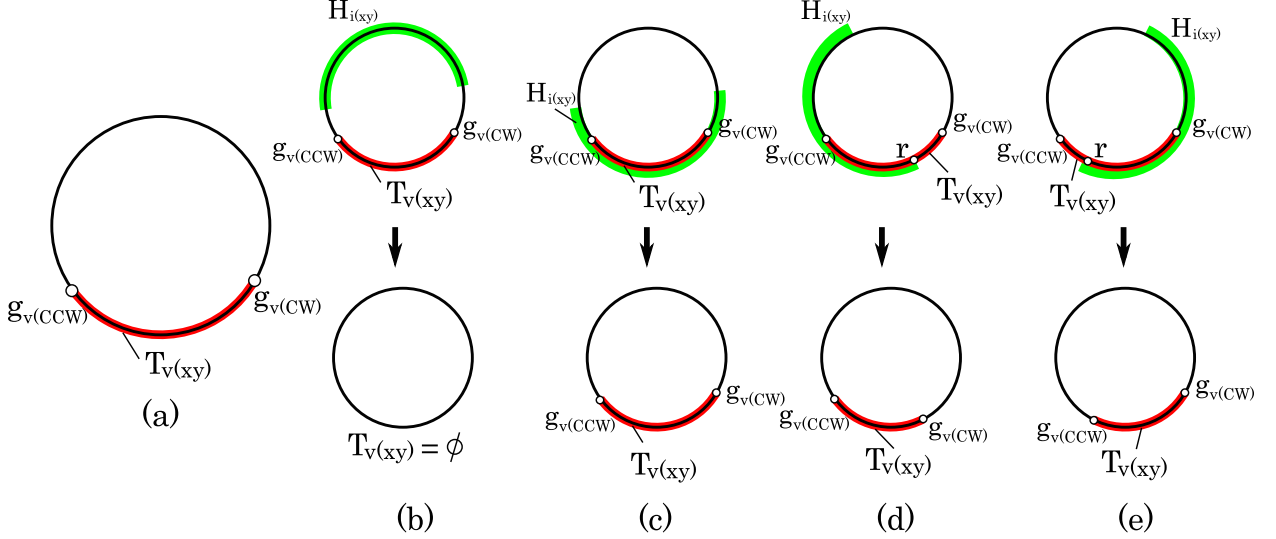


Figure 4.11: (a) $g_{v(CW)}$, $g_{v(CCW)}$, and $T_{v(xy)}$ on the Gaussian circle. Four cases of updating $g_{v(CW)}$, $g_{v(CCW)}$, and $T_{v(xy)}$ when a new $H_{i(xy)}$ is introduced are shown in (b)–(e).

axis.

We define $g_{v(CW)}^*$ as the point on the xy -plane Gaussian circle that is not in $T_{v(xy)}$ and is closest to $g_{v(CW)}$. In a similar manner, we define $g_{v(CCW)}^*$ as the point on the xy -plane Gaussian circle that is not in $T_{v(xy)}$ and is closest to $g_{v(CCW)}$. Thus $g_{v(CW)}^*$ and $g_{v(CCW)}^*$ are gravity directions at which a trapped water particle at v flows out when rotating clockwise or counterclockwise around a given rotation axis.

Letting $g_{v(CW)} = ((g_{v(CW)})_x, (g_{v(CW)})_y, 0)$ and $g_{v(CCW)} = ((g_{v(CCW)})_x, (g_{v(CCW)})_y, 0)$, we can describe $g_{v(CW)}^*$ and $g_{v(CCW)}^*$ as

$$g_{v(CW)}^* = \begin{pmatrix} (g_{v(CW)})_x \cos \epsilon - (g_{v(CW)})_y \sin \epsilon \\ (g_{v(CW)})_x \sin \epsilon + (g_{v(CW)})_y \cos \epsilon \\ 0 \end{pmatrix}$$

$$g_{v(CCW)}^* = \begin{pmatrix} (g_{v(CCW)})_x \cos \epsilon + (g_{v(CCW)})_y \sin \epsilon \\ -(g_{v(CCW)})_x \sin \epsilon + (g_{v(CCW)})_y \cos \epsilon \\ 0 \end{pmatrix}$$

where ϵ is a positive infinitesimal number representing the infinitesimal rotation.

4.3.2.2 Concave vertex where trapped water flowing out settles

In the previous subsection, for each concave vertex $v \in V_c$, we showed how we compute the two gravity directions when the trapped water particle at v flows out under rotation around the rotation axis. Now, we describe how to determine which concave vertex the water particle flowing out from v settles in (or if it exits the geometry).

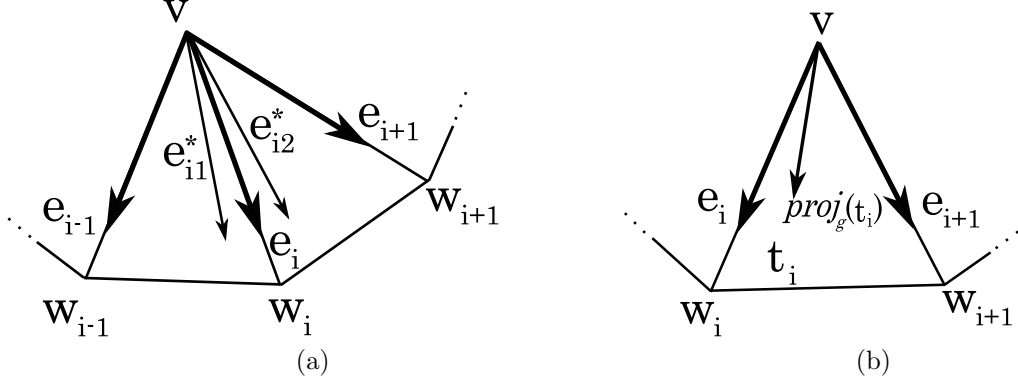


Figure 4.12: (a) Edge e_i is a *locally-steepest* edge if $e_i \cdot g > 0$ and e_i is steeper with respect to gravity g than any vectors on e_i 's adjacent triangles, e.g. e_{i1}^* and e_{i2}^* . (b) Triangle t_i is a *locally-steepest* triangle with respect to gravity g if $n_{t_i} \cdot g < 0$ and $proj_g(t)$ lies on triangle t_i .

We will use the following notation. Given a vertex v , let w_i be a member of the set of adjacent vertices of v and e_i be the normalized vector from v to w_i , that is, $e_i = (w_i - v) / \|w_i - v\|$ ($i = 1, 2, \dots$, $valence(v)$). We call edge e_i the *locally-steepest* edge from v with respect to unit gravity vector g if $e_i \cdot g > 0$ and e_i is “steeper” with respect to gravity g than any vectors from v on e_i 's adjacent triangles. An edge is steeper if $e_i \cdot g > e_{i1}^* \cdot g$ and $e_i \cdot g > e_{i2}^* \cdot g$, where $e_{i1}^* = (\epsilon e_{i-1} + (1 - \epsilon)e_i) / \|\epsilon e_{i-1} + (1 - \epsilon)e_i\|$ and $e_{i2}^* = (\epsilon e_{i+1} + (1 - \epsilon)e_i) / \|\epsilon e_{i+1} + (1 - \epsilon)e_i\|$ (ϵ is a positive infinitesimal number) (Figure 4.12a). Let t_i be the triangle incident to v defined by vertices v , w_i , and w_{i+1} . We define $proj_g(t_i)$ as the projection of gravity vector g onto the plane of triangle t_i ; thus, for triangle t_i 's normal vector n_{t_i} , we can calculate $proj_g(t_i) = (I - n_{t_i}n_{t_i}^T)g$. We call triangle t_i the *locally-steepest* triangle with respect to gravity g if $n_{t_i} \cdot g < 0$ and $proj_g(t)$ lies on triangle t_i . That is, there exist scalar values α and β that satisfy $v + proj_g(t_i)\alpha = w_i(1 - \beta) + w_{i+1}\beta$, $\alpha > 0$, and $0 < \beta < 1$. (Note that there may be more than one locally-steepest edge or triangle for a given vertex.)

We only describe the case when we rotate the geometry clockwise because the same procedure works for the counterclockwise case. Let the set of concave vertices where the water particle leaving v settles when we trace the particle with $g_{v(CW)}^*$ be $S_{v(CW)}$ and when we trace the particle with $g_{v(CCW)}^*$ be $S_{v(CCW)}$. For the sake of simplicity of notation, we let $S_v \stackrel{def}{=} S_{v(CW)}$ and $g \stackrel{def}{=} g_{v(CW)}^*$ for this explanation. The three cases of a particle leaving a vertex v_{cur} and falling through space, traveling along an edge, or traveling along the face of a triangle are handled by Procedure 1: **TraceFromVertex** (Figure 4.13 (a)(b)(c)). The three cases for a particle leaving a location p_{cur} in the middle of an edge e_{cur} and falling through space, traveling along the face of a triangle, or traveling along an edge are handled by Procedure 2: **TraceFromEdge** (Figure 4.13 (e)(f)(g)(h)). Procedure 3: **ParticleDrop**,

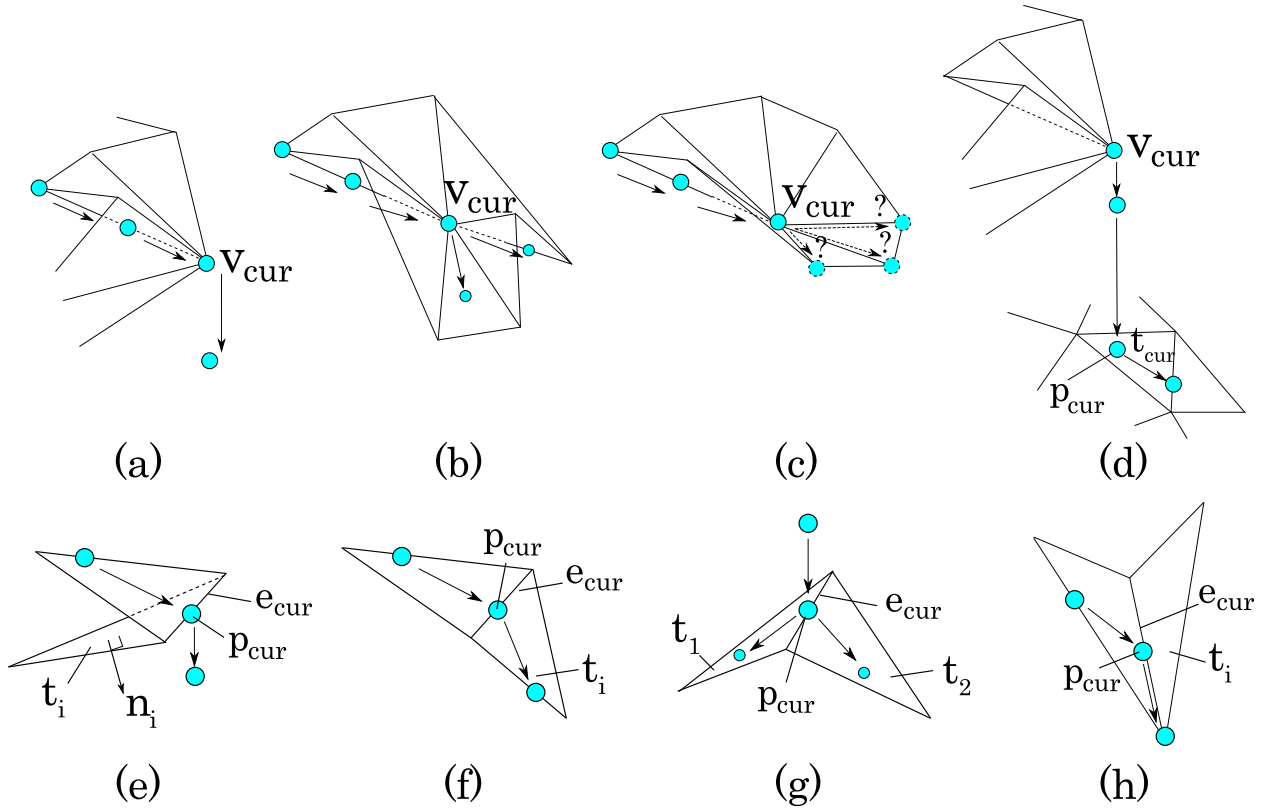


Figure 4.13: Transition cases of a water particle under gravity on various geometric shapes. (a)(b)(c) Possible movements from a vertex. (d) Movement when a particle drops vertically. (e)(f)(g)(h) Possible movements from an edge.

Procedure 4: **FindNextEdge**, and Procedure 5: **TraceOnFlatRegion** handle the transitions between these states. We outline the logic below; the corresponding detailed pseudocode for each subroutine is shown in Algorithms 5, 6, 7, 8, and 9.

We find S_v for v under gravity force g by tracking the particle location starting with Procedure 1: **TraceFromVertex**, initially setting v_{cur} to v .

1. **TraceFromVertex**:

- If a point $v_{cur} + \epsilon g$ (ϵ is a positive infinitesimal number) is outside of the geometry, the water particle falls down parallel to g from v_{cur} (Figure 4.13 (a)). To simulate this, we shoot a half-ray $v_{cur} + \gamma g$ (where γ is a positive scalar). **Go to 3.**
- Otherwise, we define $m = \arg \max_i (e_i \cdot g)$ (i.e. $\forall i, e_m \cdot g \geq e_i \cdot g$).
 - if $e_m \cdot g < 0$, the water particle flowing out from v settles at v_{cur} . We add v_{cur} to S_v .
 - if $e_m \cdot g > 0$, we let the set of locally-steepest edges of v_{cur} be E_s and the set of locally-steepest triangles of v_{cur} be T_s . For each edge e_j in E_s

($j = 1, \dots, |E_s|$), we set v_{cur} to w_j (the endpoint of e_j that is not v) and **Go to 1** for each e_j . For each triangle t_j in T_s ($j = 1, \dots, |T_s|$), we solve $v_{cur} + proj_g(t_j)\alpha = w_j(1 - \beta) + w_{j+1}\beta$. We set e_{cur} to the edge $\overline{w_j w_{j+1}}$ and p_{cur} to point $w_j(1 - \beta) + w_{j+1}\beta$. **Go to 2** for each t_j . (Figure 4.13 (b).)

- otherwise ($e_m \cdot g = 0$); we cannot decide whether the water particle settles at v_{cur} or moves to another point by looking only at local information at v_{cur} (Figure 4.13 (c)). We set p_{cur} to v_{cur} . **Go to 5**.

2. **TraceFromEdge:** Let the two triangles adjacent to e_{cur} be t_1 and t_2 with normals n_1 and n_2 respectively.

- If e_{cur} is a ridge edge and $n_1 \cdot g \geq 0$ or $n_2 \cdot g \geq 0$, the water particle falls down; we shoot a half-ray $p_{cur} + \gamma g$ (where γ is a positive scalar). **Go to 3**. (Figure 4.13 (e).)
- Otherwise, if t_1 is not perpendicular to g , we set t_{cur} to t_1 and **Go to 4**; then, if t_2 is not perpendicular to g , we set t_{cur} to t_2 and **Go to 4**. (Figure 4.13 (f)(g).) If the particle does not move along either t_1 or t_2 , the particle goes along e_{cur} (Figure 4.13 (h)). Letting the two endpoints of e_{cur} be v_a and v_b ,
 - if $v_a \cdot g > v_b \cdot g$, we set v_{cur} to v_a . **Go to 1**.
 - if $v_a \cdot g < v_b \cdot g$, we set v_{cur} to v_b . **Go to 1**.
 - otherwise ($v_a \cdot g = v_b \cdot g$), **Go to 5**.

3. **ParticleDrop:**

- If the ray does not hit any part of the input geometry, the water particle exits the geometry; we add “out” to S_v .
- Otherwise,
 - if the ray hits a vertex, we set v_{cur} to the vertex. **Go to 1**.
 - else if the ray hits an edge, we set e_{cur} to the edge and p_{cur} to the point the ray hits. **Go to 2**.
 - else if the ray hits a triangle, we set t_{cur} to the triangle and p_{cur} to the point the ray hits. If t_{cur} is not perpendicular to g , **Go to 4** (Figure 4.13 (d)). Otherwise, **Go to 5**.

4. **FindNextEdge:** We find the edge of t_{cur} such that, letting the two endpoints of the edge be v_a and v_b , there exist scalar values α and β that satisfy $p_{cur} + proj_g(t_{cur})\alpha = v_a(1 - \beta) + v_b\beta$, $\alpha > 0$, and $0 \leq \beta \leq 1$.

- If we find such an edge, then
 - when $\beta = 0$, set v_{cur} to v_a . **Go to 1**.
 - when $\beta = 1$, set v_{cur} to v_b . **Go to 1**.

- when $0 < \beta < 1$, set e_{cur} to this intersecting edge and set p_{cur} to $v_a(1-\beta)+v_b\beta$.
Go to 2.

- Otherwise, the particle does not move along t_{cur} with g .

5. **TraceOnFlatRegion:** On a horizontal region (perpendicular to g), a particle does not move via gravity; therefore, we assume that particles diffuse concentrically and flow out through the closest point from p_{cur} that can be reached along edges and triangles perpendicular to g . We call such a closest point p_f .

We define E_{perp} as the set of edges that are perpendicular to g and can be reached from p_{cur} only traversing edges and triangles perpendicular to g . We also define T_{perp} as the set of triangles perpendicular to g and incident to edges in E_{perp} . We define V_{cand} and E_{cand} as the vertices and edges where a water particle leaving p_{cur} may flow out through: V_{cand} consists of vertices each of which is an endpoint of E_{perp} and has an incident edge e_i such that $e_i \cdot g > 0$; E_{cand} consists of ridge edges in E_{perp} .

- If V_{cand} and E_{cand} are empty, the particle is trapped at this flat region. We add all the concave vertices incident to the edges in E_{perp} to S_v .
- Otherwise, we find the closest point p_f (section B.2 describes how to find p_f , given p_{cur} and E_{perp} , T_{perp} , V_{cand} , and E_{cand}).
 - If p_f is on a vertex in V_{cand} , set v_{cur} to p_f . **Go to 1.**
 - If p_f is on an edge in E_{cand} , set p_{cur} to p_f and t_{cur} to the triangle incident to the edge and not in T_{perp} . **Go to 3.**

We repeat this procedure for each v until we find all the possible concave vertices (possibly plus “out”) that should be added to each S_v . For each concave vertex $v \in V_c$, we connect the corresponding node to the nodes corresponding to the elements in $S_v(CW)$ by an edge labeled CW and to the nodes corresponding to the elements in $S_v(CCW)$ by an edge labeled CCW .

Note that the ray tracing performance in Procedure 3 will be very expensive for large inputs unless we use a bounding volume hierarchy (BVH) [Samet 2005] to limit the number of triangles tested. We used a kd-tree [Bentley 1975] for the BVH in our implementation.

4.4 Checking Drainability

Now, using the draining graph constructed, we test whether or not a rotation around a given rotation axis can completely drain trapped water. For each concave vertex $v \in V_c$, if there is a path from the corresponding node to the *out* node in the draining graph, we can drain water trapped at v by rotating the input geometry around this rotation axis. Note that when we rotate the geometry clockwise, we can use only edges labeled CW , and when we rotate counterclockwise, we can use only edges labeled CCW .

Algorithm 5 TraceFromVertex

```

Input: vertex  $v_{cur}$ 
if  $v_{cur} + \epsilon g$  is outside of the geometry then
  ParticleDrop( $v_{cur} + \gamma g$ )
else
   $m \leftarrow \arg \max_i (e_i \cdot g)$ 
  if  $e_m \cdot g < 0$  then
    // water particle settles at  $v_{cur}$ 
     $S_v \leftarrow S_v \cup v_{cur}$ 
  else if  $e_m \cdot g > 0$  then
    for  $j = 1$  to  $|E_s|$  do
      TraceFromVertex( $w_j$ )
    end for
    for  $j = 1$  to  $|T_s|$  do
      Solve for  $\alpha$  and  $\beta$  s.t.  $v_{cur} + \text{proj}_g(t_j)\alpha = w_j(1 - \beta) + w_{j+1}\beta$ 
      TraceFromEdge( $\overline{w_j w_{j+1}}$ ,  $w_j(1 - \beta) + w_{j+1}\beta$ )
    end for
  else
    //  $e_m \cdot g = 0$ 
    TraceOnFlatRegion( $v_{cur}$ )
  end if
end if

```

4.4.1 Checking Procedure

Letting $n = |V_c|$, if we take a naive approach, we may have to trace n nodes from each concave vertex $v \in V_c$ in the worst case. Therefore, the total running time becomes $O(n^2)$. However, we observe that if there is a path from one node to the *out* node, it means that there is also a path from the intermediate nodes on this path to the *out* node (because draining is transitive). For example, in Figure 4.3, if we find a path from A through B, C, and D to *out*, we know that there is also a path from B, through C and D to *out*, and so on.

Based on this observation, we can improve the running time through the following procedure. Suppose we rotate the geometry in a clockwise direction. Then, trapped water particles at the concave vertices whose corresponding nodes are directly connected to the *out* node by the edges labeled as *CW* can be drained. Let the set of these nodes be N_d . Next, consider trapped water particles at concave vertices whose corresponding nodes are directly connected to the nodes in N_d by edges labeled as *CW*; these can be drained as well. We add these nodes to N_d , and continue recursively. This recursion stops when all the nodes connecting to at least one of the nodes in N_d by the edges labeled as *CW* are in N_d . Then, after the recursion stops, if $|N_d| = n$, we can guarantee that trapped water particles at all of the concave vertices are completely drained by rotation around the given rotation axis.

Algorithm 6 TraceFromEdge

Input: current edge e_{cur} , current point p_{cur}
if e_{cur} is a ridge edge, and $n_1 \cdot g \geq 0$ or $n_2 \cdot g \geq 0$ **then**
 ParticleDrop($p_{cur} + \gamma g$)
else
 $found1 \leftarrow$ FindNextEdge(t_1, p_{cur})
 $found2 \leftarrow$ FindNextEdge(t_2, p_{cur})
 if $found1 = false$ and $found2 = false$ **then**
 if $v_a \cdot g > v_b \cdot g$ **then**
 TraceFromVertex(v_a)
 else if $v_a \cdot g < v_b \cdot g$ **then**
 TraceFromVertex(v_b)
 else
 // $v_a \cdot g = v_b \cdot g$
 TraceOnFlatRegion(p_{cur})
 end if
 end if
end if

Algorithm 7 ParticleDrop

Input: half-ray h_Ray
if h_Ray does not hit any part of the input geometry **then**
 $S_v \leftarrow S_v \cup out$
else if half-ray h_Ray hits a vertex v_{hit} **then**
 TraceFromVertex(v_{hit})
else if half-ray h_Ray hits an edge e_{hit} **then**
 $p_{hit} \leftarrow$ point where h_Ray hits e_{hit}
 TraceFromEdge(e_{hit}, p_{hit})
else if half-ray h_Ray hits a triangle t_{hit} **then**
 $p_{hit} \leftarrow$ point where h_Ray intersects t_{hit}
 if t_{hit} is not perpendicular to g **then**
 FindNextEdge(t_{hit}, p_{hit})
 else
 TraceOnFlatRegion(t_{hit})
 end if
end if

Through this approach, we do not have to check the same node more than once. Therefore, the time complexity becomes $O(n)$.

Algorithm 8 FindNextEdge

Input: triangle t_{cur} , current point p_{cur}

for all three edges e_i of t_{cur} ($i = 1, 2, 3$) **do**

$v_a \leftarrow$ one endpoint of e_i

$v_b \leftarrow$ other endpoint of e_i

Solve for α and β s.t. $p_{cur} + \text{proj}_g(t_{cur})\alpha = v_a(1 - \beta) + v_b\beta$

if $\alpha > 0$ and $0 \leq \beta \leq 1$ **then**

if $\beta = 0$ **then**

TraceFromVertex(v_a)

else if $\beta = 1$ **then**

TraceFromVertex(v_b)

else if $0 < \beta < 1$ **then**

TraceFromEdge($e_i, v_a(1 - \beta) + v_b\beta$)

end if

return *true*

end if

end for

// particle does not move along t_{cur}

return *false*

Algorithm 9 TraceOnFlatRegion

Input: current point p_{cur}

if V_{cand} and E_{cand} are empty **then**

// flat region is a water trap

$S_v \leftarrow S_v \cup$ (all the concave vertices incident to edges in E_{perp})

else

Find p_f from p_{cur} , E_{perp} , T_{perp} , V_{cand} , and E_{cand} *// (see B.2)*

if p_f is on a vertex in V_{cand} **then**

TraceFromVertex(p_f)

else

// p_f is on an edge in E_{cand}

$t_{cur} \leftarrow$ triangle incident to the edge and not in T_{perp}

FindNextEdge(t_{cur}, p_f)

end if

end if

4.5 Results

We first visualize the analysis output for two sample parts, one simple and one complex, and then discuss the performance.

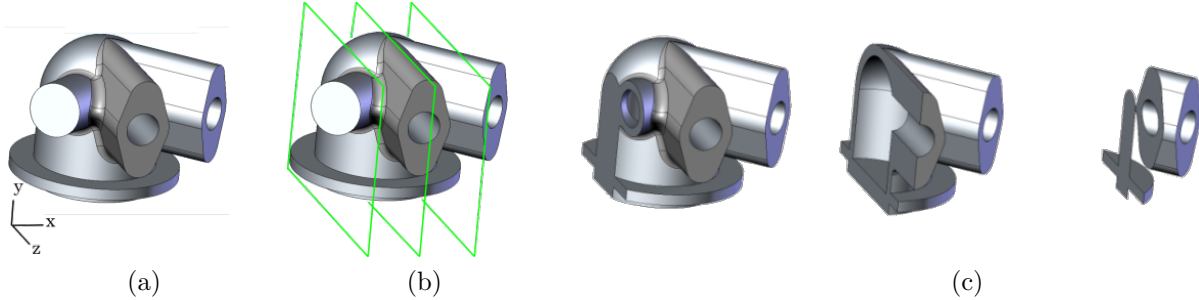


Figure 4.14: We applied our theory to a simple mechanical workpiece shown in (a). The three sections indicated in (b) are shown in (c).

4.5.1 Output

We first show our output graphically for the simple mechanical part shown in Figure 4.14a. Figure 4.14b indicates the locations of the three cross-sections shown in Figure 4.14c, revealing an inside void of the workpiece where water can be trapped. Figure 4.15 (a) and (b) plot whether or not a rotation around a given axis (θ, ϕ) completely drains the workpiece under CW and CCW rotation respectively. To verify these results, we show some representative configurations in Figure 4.15c for the CW case. All these configurations are when $\phi = 0$ and viewed from $+\infty$ on the y-axis. If we fix $\phi = 0$, when $170^\circ \leq \theta \leq 240^\circ$ and $350^\circ \leq \theta \leq 420^\circ (= 60^\circ)$, we cannot drain the workpiece as shown in Figure 4.15a. The four configurations shown in Figure 4.15c are set at these four limits. Notice that the angle between the x-axis and the outlet closer to the x-axis is the same for all four cases; this angle is a threshold for whether or not a given rotation axis works for draining. This shows that our algorithm can capture this threshold.

Figure 4.16 shows representative results of two additional CW cases ((a) $\theta = 0^\circ, \phi = 0^\circ$, (b) $\theta = 230^\circ, \phi = 30^\circ$). For the center and right figures, the vertices shown in blue are concave vertices where a water trap is potentially formed when we rotate the workpiece around the corresponding rotation axis; we cannot drain the workpiece. Figure 4.17 shows a result when $\theta = 30^\circ$ and $\phi = 60^\circ$. This is an example where CW rotation works but CCW rotation does not work (see Figure 4.15 (a) and (b)). Figure 4.18 shows the corresponding transition of a water particle when we rotate (a) clockwise and (b) counterclockwise around this rotation axis.

To get a sense of how sensitive our algorithm is to the coarseness of the triangulation, we compared these results to those on a fine tessellation of the same model. We found only slight shifts in the boundary between the drainable and non-drainable regions (see Figure 4.19).

We also applied our algorithm to a complex automotive model shown in Figure 4.20a. Our

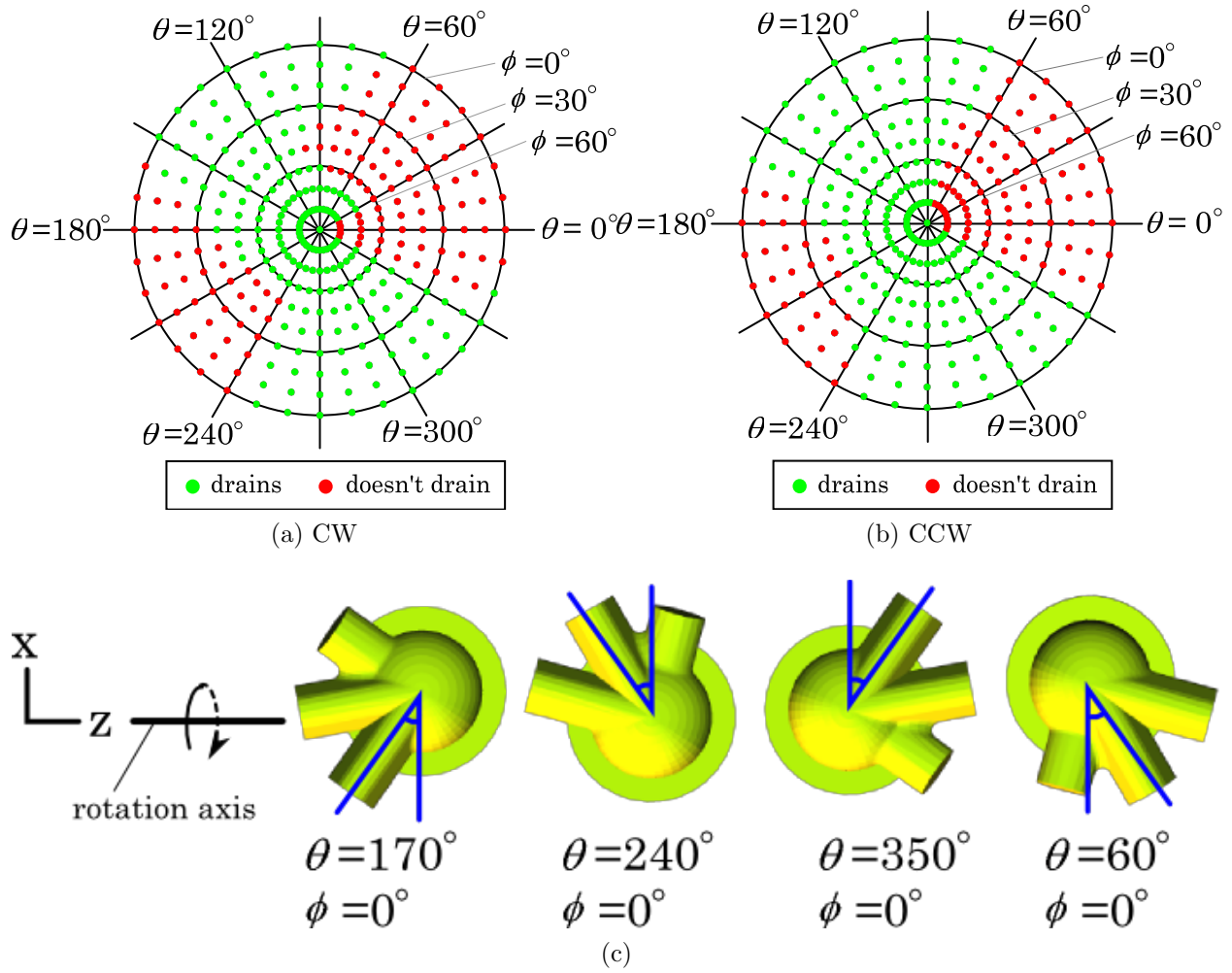


Figure 4.15: Whether or not a rotation around a given axis (θ, ϕ) completely drains the workpiece (a) under CW and (b) CCW rotation. (c) The configurations of the workpiece when $\phi = 0$ and $\theta = 170^\circ, 240^\circ, 350^\circ,$ and 60° that are the limits in the $\phi = 0$ plane of whether the rotation axis drains the workpiece or not. We can see that the angle between the x-axis and the outlet closer to the x-axis is the same for all four cases.

algorithm can quickly compute whether or not a given rotation axis will drain the workpiece even when internal passages (Figure 4.20(b)(c)) are very complex, as in this example. Figure 4.20(d) and (e) plot whether or not the indicated rotation axis drains this model. Figure 4.20 (f) shows concave vertices (colored blue) where a water trap is potentially formed when the rotation axis is set to $\theta = 270^\circ, \phi = 0^\circ$. Figure 4.20 (g) shows the corresponding axis-aligned magnified view.

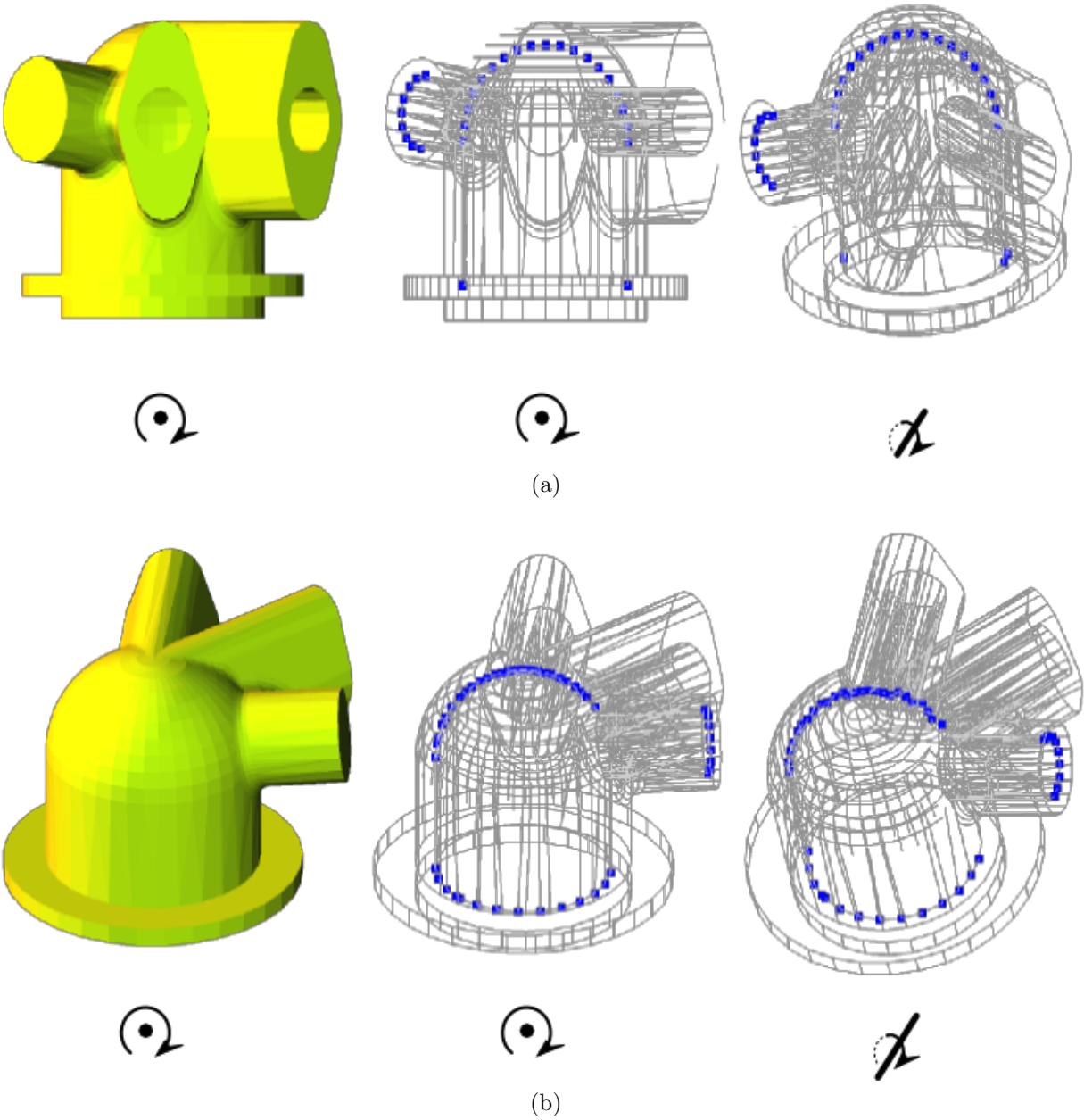


Figure 4.16: Some representative results. (a) $\theta = 0^\circ$, $\phi = 0^\circ$. (b) $\theta = 230^\circ$, $\phi = 30^\circ$. For both (a) and (b), the rotation axis is set perpendicular to the paper for the left and center figures. The right figure is a view from a different angle. For the center and right figures, the indicated vertices are concave vertices such that once a water particle is trapped there, it will never exit the workpiece when we rotate it around the corresponding rotation axis.

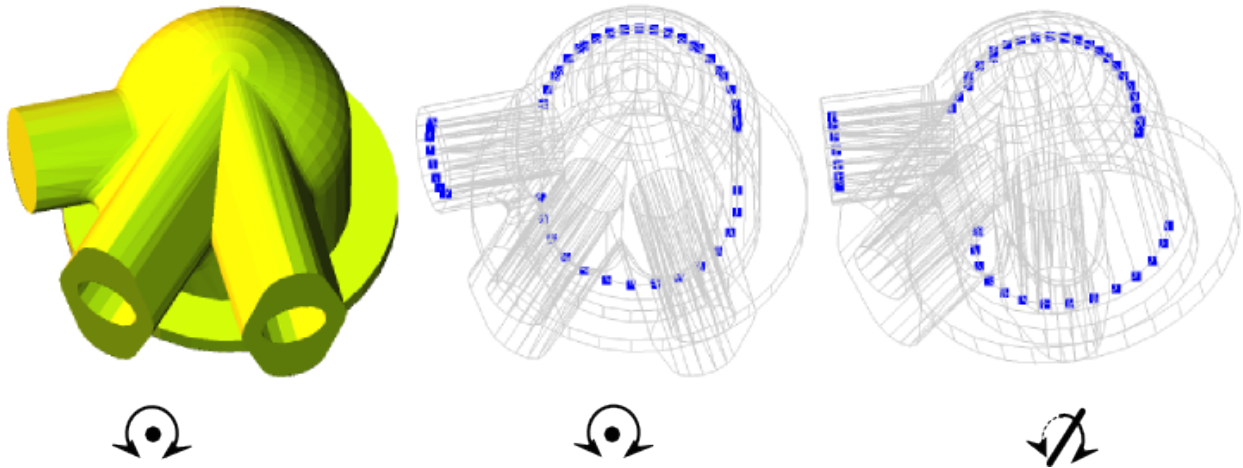


Figure 4.17: $\theta = 30^\circ$, $\phi = 60^\circ$. With this rotation axis, CW rotation drains the workpiece but CCW rotation does not. The rotation axis is set perpendicular to the paper for the left and center figures. The right figure is a view from a different angle.

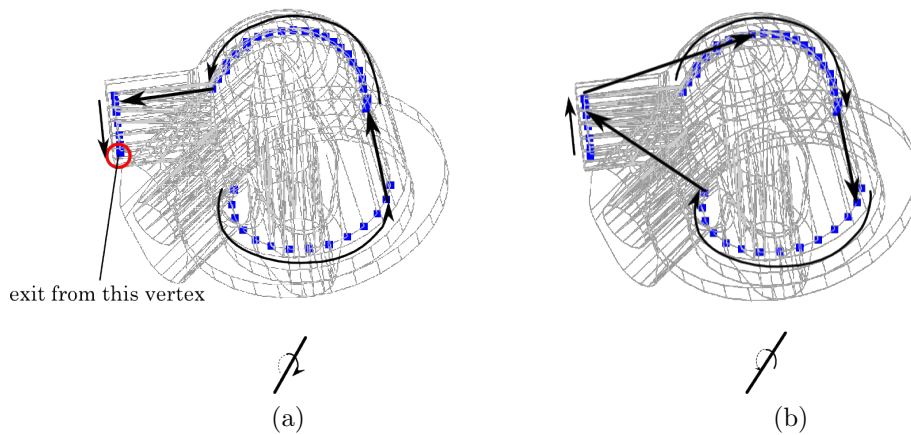


Figure 4.18: The transition of a water particle when we rotate (a) clockwise and (b) counterclockwise around a rotation axis $\theta = 30^\circ$, $\phi = 60^\circ$. CW rotation drains the workpiece but CCW rotation does not.

4.5.2 Performance

Table 5.1 shows the performance of our implementation on a 2.66GHz CPU with 4GB of RAM. The initialization mainly consists of identifying the concave vertices of the input

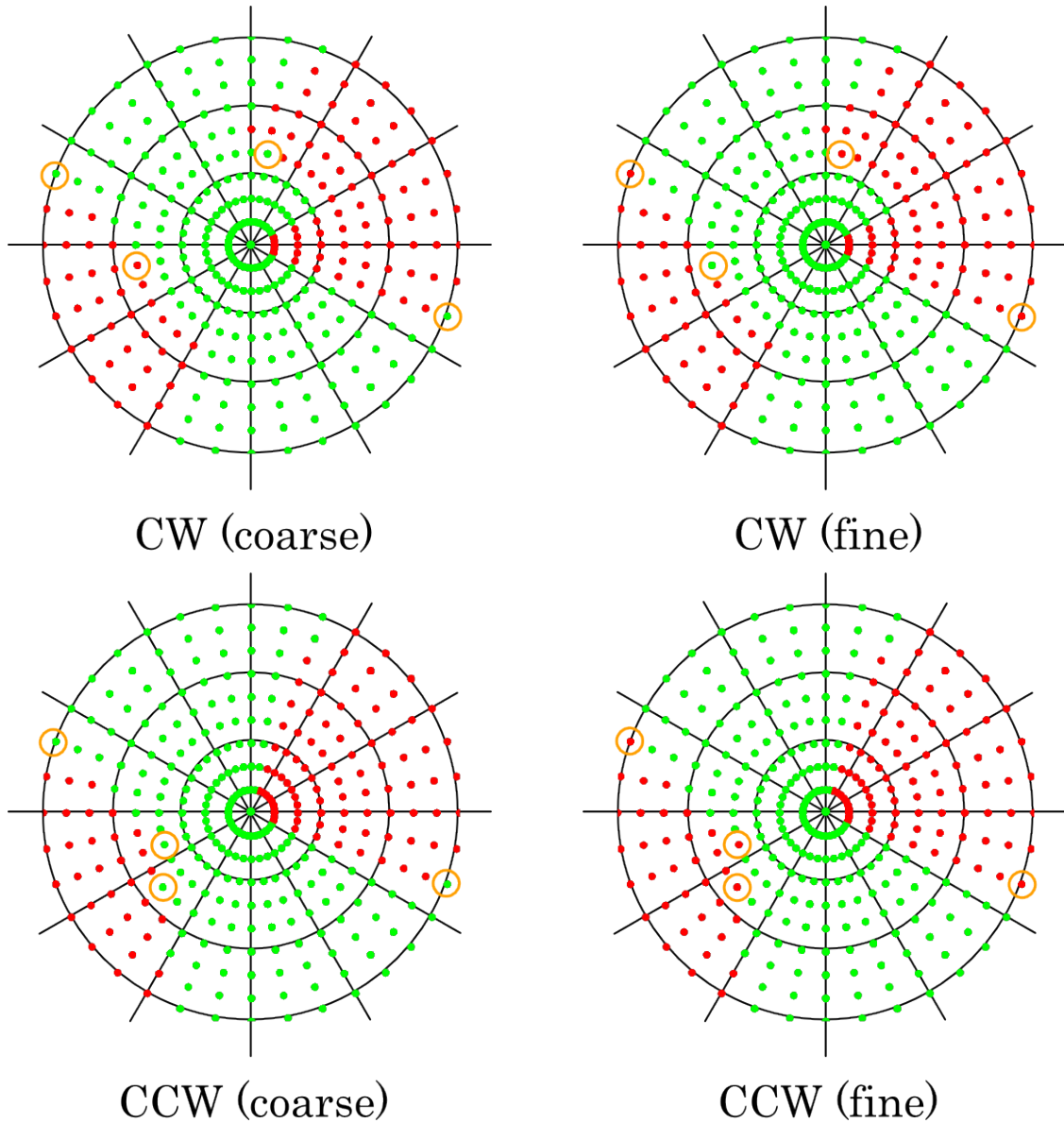


Figure 4.19: Comparison of the results shown in Figure 4.15 (a) and (b) with results for a finer tessellation of the same model with almost five times the number of vertices. The results that differ are circled.

mesh and constructing a BVH for speeding up the ray tracing phase of the geometry. This information can be reused no matter what rotation axis is being considered, and typically more than one possible rotation axis will need to be tested. Then, after the initialization, we tested $36 \times 9 = 324$ (sampled at every 10 degrees in both θ and ϕ directions) rotation axes for each model and report the average and maximum time in Table 5.1. We can see that we

Table 4.1: Time for initialization and to test each additional rotation axis (average and maximum).





		triangles	vertices	concave vertices	initialization time (sec)	average time (sec)	maximum time (sec)
(a)		3,572	1,796	428	0.575	0.006	0.035
(b)		12,0004	59,920	18,203	5.841	0.195	0.278
(c)		160,312	79,982	31,829	9.319	0.360	0.760
(d)		289,956	144,546	57,412	20.742	0.933	5.730

Table 4.2: Detailed timing data, showing the individual timing (average and maximum) for each of three phases to test each additional rotation axis.

	Find $g_{v(CW)}^*$ and $g_{v(CCW)}^*$		Find $S_{v(CW)}$ and $S_{v(CCW)}$		Checking Drainability		Total	
	average(sec)	max(sec)	average(sec)	max(sec)	average(sec)	max(sec)	average(sec)	max(sec)
(a)	0.001	0.011	0.002	0.015	0.004	0.009	0.006	0.035
(b)	0.045	0.064	0.138	0.227	0.013	0.017	0.195	0.278
(c)	0.067	0.106	0.279	0.654	0.014	0.025	0.360	0.760
(d)	0.131	0.169	0.774	5.575	0.028	0.040	0.933	5.730

can test a given rotation axis very quickly and give near-interactive feedback to designers for testing each additional axis. Table 4.2 shows the detailed timing data, showing the individual timing for each of three phases to test a rotation axis, i.e. finding $g_{v(CW)}^*$ and $g_{v(CCW)}^*$ (section 4.3.2.1), finding $S_{v(CW)}$ and $S_{v(CCW)}$ (section 4.3.2.2), and testing drainability by analyzing the draining graph constructed through the preceding two phases (section 4.4). The performance bottleneck of our current implementation is finding $S_{v(CW)}$ and $S_{v(CCW)}$ (i.e. the particle tracing operation in the graph construction phase), so we will investigate offloading some of this work to the GPU in future work. We also measured the number of function calls made to Algorithm 1-5 in determining $S_{v(CW)}$ and $S_{v(CCW)}$ for each concave vertex $v \in V_c$ (shown in Table 4.3). Although the maximum number of calls is higher for the more complex models, the average was low for all models tested.

Table 4.3: The number of function calls (average and maximum) to determine $S_{v(CW)}$ and $S_{v(CCW)}$ for each concave vertex $v \in V_c$.

	TraceFromVertex		TraceFromEdge		ParticleDrop		FindNextEdge		TraceOnFlatRegion		Total	
	average	max	average	max	average	max	average	max	average	max	average	max
(a)	3.02	31	5.42	329	0.26	11	5.72	333	0.01	2	14.43	694
(b)	4.42	1108	6.50	1661	0.44	152	7.37	1772	0.001	2	18.74	4046
(c)	3.35	538	4.97	1338	0.40	235	5.53	1558	0.005	5	14.25	3440
(d)	3.87	14260	8.25	121779	0.59	7392	9.16	134672	0.020	16	21.89	278103

4.6 Complexity Analysis

We now analyze the scalability of our algorithm. In the graph construction phase, for each concave vertex $v \in V_c$, first we compute the gravity directions when the trapped water at v starts to flow out. For each concave vertex v , this takes a constant number of operations equal to the number of edges incident to v , so it is in $O(n)$. For each concave vertex $v \in V_c$, we find the concave vertex into which the trapped water particle flowing out from v settles. In theory, for each v , we have to check all triangles and vertices of the geometry to find the final location in the worst case. Therefore, as Table 4.3 shows, the maximum number of function calls possibly becomes very high. We are still investigating the performance of particle tracing to construct this graph. However, from the fact that a water particle is driven by only a fixed gravity force and the assumption that the input triangles and vertices are uniformly distributed in space, in practice the number of vertices and triangles checked are only a very small fraction of n , reducing worst case $O(n^2)$ growth to close to linear on average in practice; experimental results shown in Table 4.3 support this. Once the graph is constructed, the checking phase runs in $O(n)$ time as described in section 4.

4.7 Conclusion

In this chapter, we presented a new geometric algorithm to test whether a rotation around a given rotation axis can drain an input geometry. Our proof-of-concept implementation can test input meshes of complex industrial parts containing over 100,000 vertices in about a second, a huge improvement compared to using commercial general-purpose simulation packages that can take hours to converge.

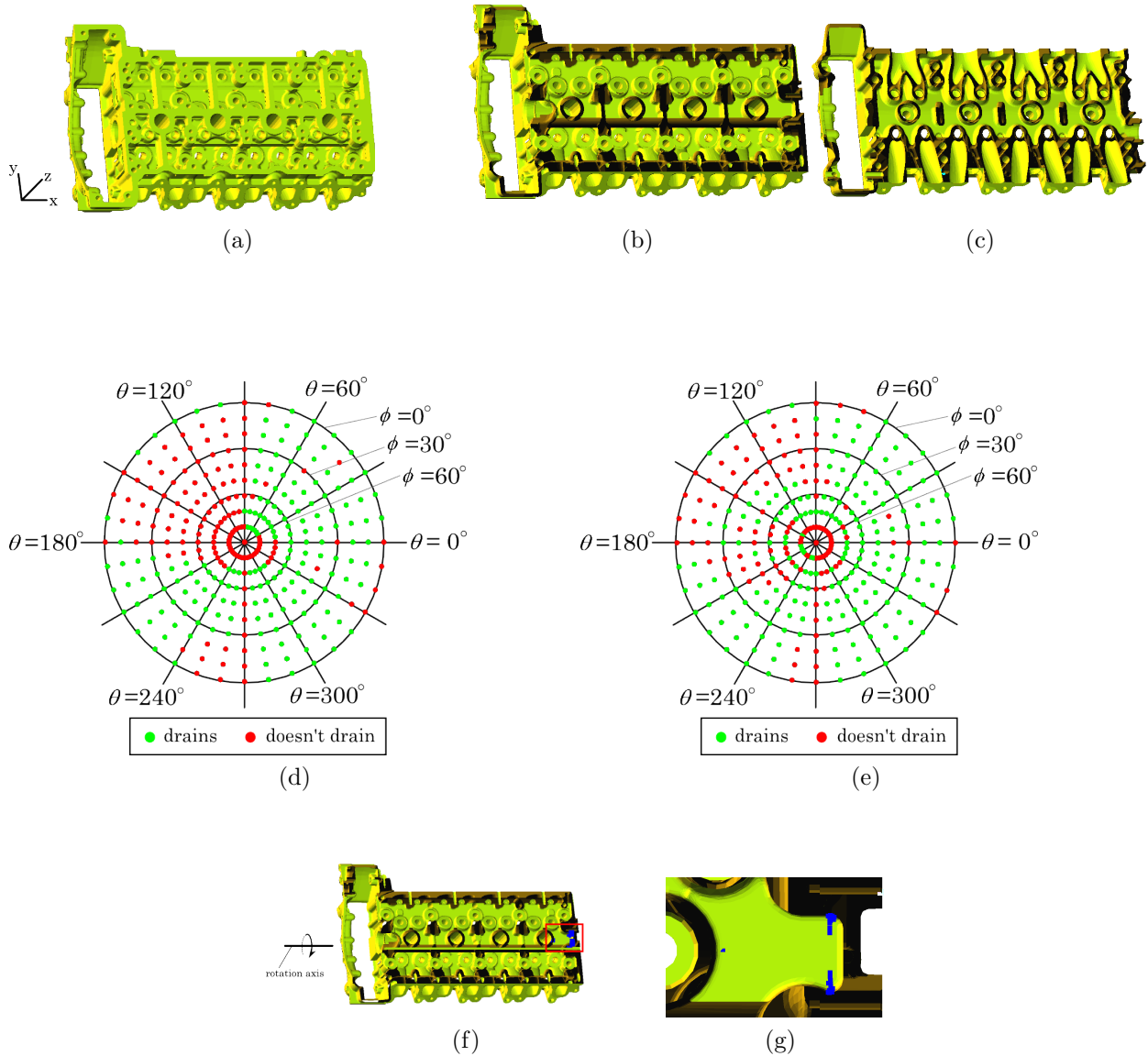


Figure 4.20: (a) Cylinder head model (b)(c) Cross sections revealing the internal passages of the model shown in (a). (d) Plot of whether or not rotation around a given rotation axis completely drains the workpiece under CW rotation and (e) CCW rotation. (f) The concave vertices such that once a water particle is trapped there, it will never exit the workpiece when we rotate it around rotation axis $\theta = 270^\circ$, $\phi = 0^\circ$, which is set horizontally in the plane of the paper. (g) Magnified view of the region indicated in (f).

Chapter 5

Finding a Rotation Axis to Drain a 3D Workpiece

In this chapter, we introduce an algorithm to find the set of *all* rotation axes that would drain a given workpiece geometry (again represented as a triangulated mesh). In chapter 4, we introduced an algorithm to test whether a given rotation axis can drain a given workpiece geometry, showing that the problem can be solved geometrically by constructing and analyzing a directed graph. Suppose that we are given a rotation axis that is found to drain the workpiece using the testing algorithm. In practice, it is also essential to make sure that the rotation axis's nearby rotation axes also drain the workpiece because, although the size of a water particle is finite in the real world, we have assumed that the size of a water particle is infinitesimal in the algorithm. If one of the nearby rotation axes cannot drain the workpiece, the chosen rotation axis might not drain the workpiece in the real world. Taking this into account, if we were to use the previous testing algorithm to find a rotation axis that can drain the workpiece in practice, we may need to test a great many rotation axes to find one that is feasible. This is generally time-consuming and less accurate. Thus, we are motivated to move beyond sampling (testing specified axes) to a configuration space approach (finding all drainable axes).

We introduce a new algorithm to find all rotation axes that drain the workpiece, not just testing a specified rotation axis, by introducing the *extended draining graph* that represents all the possible transitions of water particles considering all the possible rotation axes and rotation directions. We introduce a dual-space stabbing line approach to efficiently analyze the drainability of all possible paths through this graph, for all possible rotation axes. Since we analyze the configuration space, our algorithm can find any possible rotation axis that drains the workpiece.

5.1 Algorithm Overview

5.1.1 Testing a Rotation Axis

Since we develop our algorithm by extending the algorithm to test a given rotation axis proposed in chapter 4, we first briefly describe the algorithm. The key observation to solve the problem geometrically is that a water trap is always formed at a concave vertex, assuming that the geometry of an input workpiece is represented as a triangulated mesh. The definition of a concave vertex for a 3D geometry is as follows:

Concave vertex Given a vertex v , we let w_i be a member of the set of adjacent vertices of v and e_i be the vector from v to w_i (i.e. $e_i = w_i - v$). Then, letting $valence(v)$ be its valence, we check if there is a unit vector d such that, for all the adjacent vertices w_i of v ($i = 1, 2, \dots, valence(v)$), $e_i \cdot d < 0$. If there is such a d and a point $p = v + \epsilon d$ (ϵ a positive infinitesimal number) is inside of the given geometry, we call v a “concave” vertex. Otherwise, v is not a concave vertex.

Based on the observation that a water trap is always formed at a concave vertex, we can represent the transition of water particles in voids of the workpiece using a directed graph called the *draining graph*. In a draining graph, each node corresponds to a concave vertex of the geometry except the one that represents the exterior of the geometry called “out” node, and each edge represents the transition of water particles between concave vertices. Figure 5.1 shows an example of a draining graph.

Since water particles’ movement between concave vertices changes if either the rotation axis or the rotation direction (either clockwise or counterclockwise) changes, a draining graph is constructed for a particular rotation axis and a rotation direction. Given a rotation axis r and a concave vertex v , if a water trap could potentially be formed at v when the workpiece is rotated around r , we add the node $N(v)$ corresponding to v to the draining graph. A directed edge is set from $N(v)$ to the node corresponding to the concave vertex that a water particle from v first moves to when we rotate the workpiece around r in the given rotation direction. If a water particle moves from v to the exterior of the geometry, we add a directed edge from $N(v)$ to the out node. In Figure 5.1, we show two different draining graphs for the geometry shown in Figure 5.1 (a). The draining graph shown in Figure 5.1 (b) is constructed for the rotation axis parallel to the x-axis. The draining graph shown in Figure 5.1 (c) is constructed for the rotation axis parallel to the y-axis. The rotation direction is clockwise when viewed from $+\infty$ on the x-axis and the y-axis, respectively.

To test the drainability of a given workpiece, given a rotation axis and direction, we first construct the corresponding draining graph. Then, we check whether there is a path from each node to the out node in the draining graph. If there is a node that does not have a path to the out node, we cannot drain the workpiece (Figure 5.1 (b)). If there is a path from any node to the out node, we can drain the workpiece with the rotation axis and rotation direction (Figure 5.1 (c)).

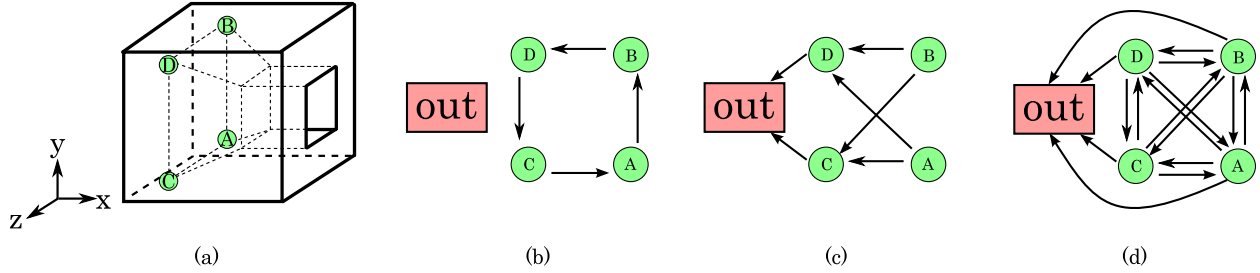


Figure 5.1: (a) Geometry with four concave vertices. (b) The draining graph constructed for clockwise rotation around the x-axis (viewed from $+\infty$ on the axis). Since there is at least one node that does not have a path to the out node, we cannot drain the workpiece. (c) The draining graph constructed for clockwise rotation around the y-axis (viewed from $+\infty$ on the axis). Since there is a path from every node to the out node, we can drain this geometry with this rotation axis. (d) The extended draining graph. This represents all the possible transitions of water particles considering all the possible rotation axes and rotation directions. Therefore, the edges of each draining graph in (b) and (c) are a subset of those of the extended draining graph.

5.1.2 Finding a Rotation Axis

To find (not just test) a rotation axis to drain an input workpiece, we introduce the *extended draining graph* that represents all the possible transitions of water particles considering all the possible rotation axes and rotation directions. Thus, the original draining graph for any given rotation axis and rotation direction is a subgraph of the extended draining graph. Figure 5.1 (d) shows an example of an extended draining graph.

In the extended draining graph, each directed edge E records a set of rotation axes R_E , each of which causes the transition indicated by E . This means, given a rotation axis r and a directed edge E of the extended draining graph, if $r \in R_E$, r causes the transition indicated by E .

Then, the condition to drain a concave vertex v with rotation axis r is either one of the following:

1. A water particle is never trapped at v when the workpiece is rotated around r .
2. $N(v)$ has a path to the out node such that, for every directed edge E in the path, $r \in R_E$.

If r satisfies one of these conditions for every concave vertex of the workpiece, r can drain the workpiece. Therefore, our goal is to find a set of rotation axes that satisfy at least one of the two conditions for all the concave vertices of the workpiece.

In section 5.2, we explain how to find a set of rotation axes that satisfy the first condition for a concave vertex v . In section 5.3, we explain how to construct the extended draining graph. Then, in section 5.4, we explain how to find a set of rotation axes that satisfy the

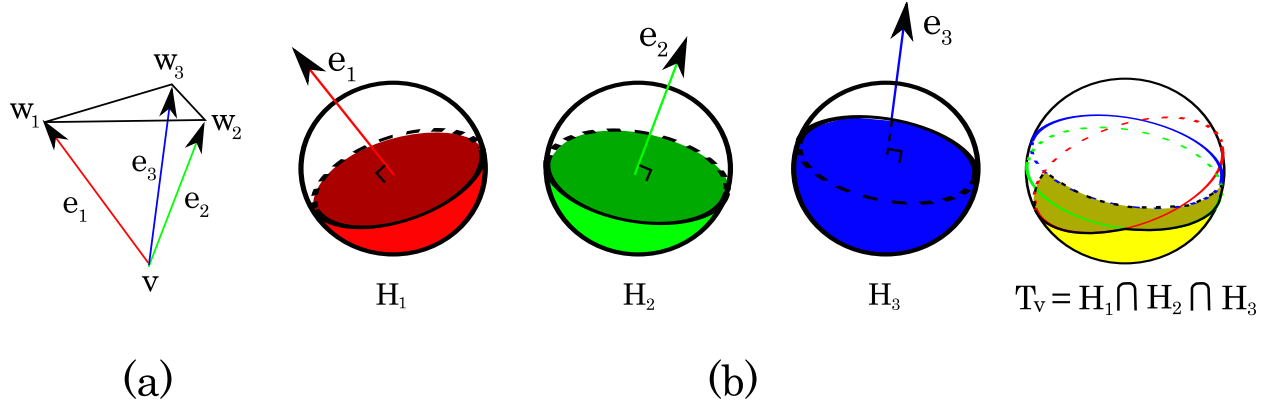


Figure 5.2: (a) Concave vertex v . (b) The corresponding H_i and T_v .

second condition for a concave vertex v using the extended draining graph. Finally, in section 5.5, we describe how to find a rotation axis r that drains the entire workpiece based on the discussion of the two conditions in section 5.2 and 5.4.

5.2 Rotation axes for which a water particle is never trapped at concave vertex v

In this section, we explain how to find a set of rotation axes that satisfy the first condition for a concave vertex v : the set consists of a rotation axis r where a water particle is never trapped at a concave vertex v when the workpiece is rotated around r .

5.2.1 T_v : Gravity directions where a water particle is trapped at concave vertex v

The gravity direction determines whether a water particle is trapped at a concave vertex. Letting V_c be the set of concave vertices of a given workpiece, for each concave vertex $v \in V_c$, we first compute all the gravity directions for which a water particle at v will be trapped. We represent gravity directions as points on the *Gaussian sphere* (a sphere with radius one and center at the origin). Then, we use the notation T_v to describe such water trap gravity directions; when a water particle is trapped at v , the current gravity direction g must be in T_v on the Gaussian sphere.

T_v is calculated as follows. Given a concave vertex v , let w_i be a member of vertices adjacent to v in the input geometry's triangulated mesh and e_i be the vector from v to w_i (i.e. $e_i = w_i - v$). For each e_i , we define a half-space H_i of directions on the Gaussian sphere $H_i = \{p \mid e_i \cdot p \leq 0, \|p\| = 1\}$; $T_v = \bigcap_i H_i$ (see Figure 5.2).

Since the boundary of H_i is a great circle and $T_v = \bigcap_i H_i$, the boundary of T_v , ∂T_v , consists of arcs of great circles, called great arcs (Figure 5.3). We call such a great arc a T_v -

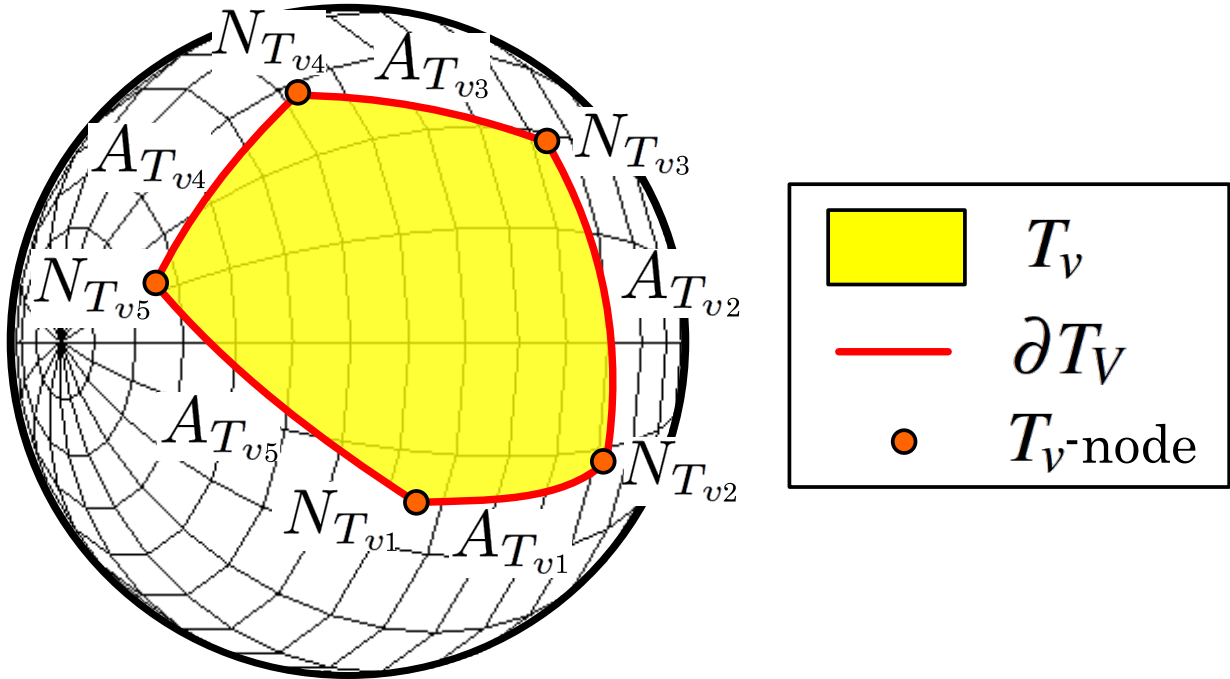


Figure 5.3: The boundary ∂T_v of T_v consists of T_v -arcs $A_{T_{v_j}}$, each of which is a portion of a great circle. T_v -nodes $N_{T_{v_j}}$ and $N_{T_{v_{j+1}}}$ bound T_v -arc $A_{T_{v_j}}$.

arc and a point that bounds a T_v -arc a T_v -node. T_v -arc $A_{T_{v_j}}$ ($j = 1, 2, \dots, |\partial T_v|$) is bounded by two T_v -nodes $N_{T_{v_j}}$ and $N_{T_{v_{j+1}}}$ where $|\partial T_v|$ is the number of T_v -arcs constituting ∂T_v , and $N_{T_{v_{|\partial T_v|+1}}} \equiv N_{T_{v_1}}$. Note that $|\partial T_v|$ is not necessarily equal to $\text{valence}(v)$ because there may be a half-space H_i whose boundary does not contribute to ∂T_v . Appendix C.1 describes how to construct ∂T_v from the set of incident edges e_i .

Next, we consider the set of rotation axes where a water particle is never trapped at a given concave vertex.

5.2.2 Rotation axes for which a water particle is never trapped at a given concave vertex

As we did for gravity directions, we express any rotation axis r as a point (r_x, r_y, r_z) on the Gaussian sphere (point (r_x, r_y, r_z) expresses the rotation axis passing through the origin and the point (r_x, r_y, r_z)). Since point (r_x, r_y, r_z) and point $(-r_x, -r_y, -r_z)$ represent the same rotation axis, we restrict the y -component of r to be non-negative throughout the paper (i.e. $r_y \geq 0$). Further, we assume that $r_y > 0$ in the following discussion unless otherwise stated (refer to section 5.8.1 for how to handle the rotation axes where $r_y = 0$).

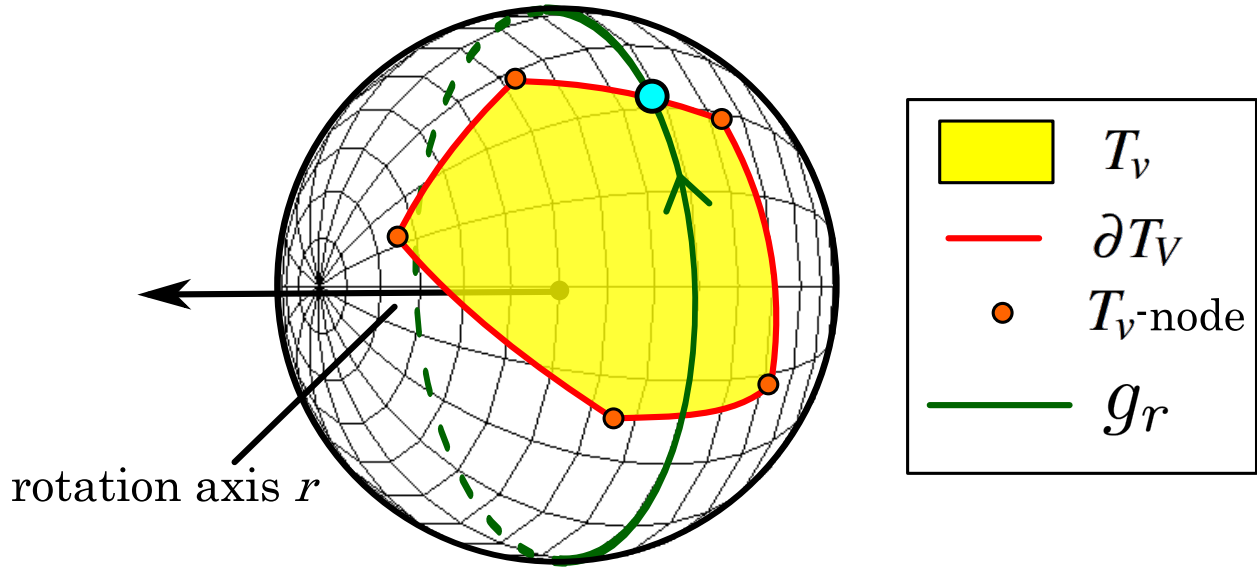


Figure 5.4: The space g_r where the gravity direction g possibly attains becomes a great circle on the Gaussian sphere. This space can be described as $g_r = \{p \mid r \cdot p = 0, \|p\| = 1\}$ where r is the vector representing a rotation axis. If g_r does not intersect T_v , a water particle is never trapped at v when we rotate the geometry around r . On the other hand, if g_r intersects T_v , a water particle is potentially trapped at v . When a water particle is trapped at v , the current gravity direction g must be in T_v . As the workpiece rotates around r , g moves along g_r (the arrow on g_r indicates the direction of g). Then, when g passes through a gravity direction that is on ∂T_v (shown in blue), a water particle trapped at concave vertex v drains from this vertex.

The important observation is, given a rotation axis r , r and a gravity direction g are always orthogonal; therefore, the possible gravity directions passed through while rotating around r , g_r , is a great circle on the Gaussian sphere. These possible configurations can be described as $g_r = \{p \mid r \cdot p = 0, \|p\| = 1\}$ (Figure 5.4).

For a given rotation axis r and a given concave vertex v , if g_r intersects T_v , a water particle is potentially trapped at v . When a water particle is trapped at v , the current gravity direction g must be in T_v . In this case, as the workpiece rotates around rotation axis r , the gravity direction g moves along g_r on the Gaussian sphere, and, when g passes through a point on ∂T_v , the water particle at v moves to another concave vertex (or exits the workpiece).

On the other hand, for a given rotation axis r and a given concave vertex v , if g_r does not intersect T_v , a water particle is never trapped at v when we rotate the workpiece around r . Equivalently, if g_r does not intersect the boundary of T_v , ∂T_v , a water particle is never

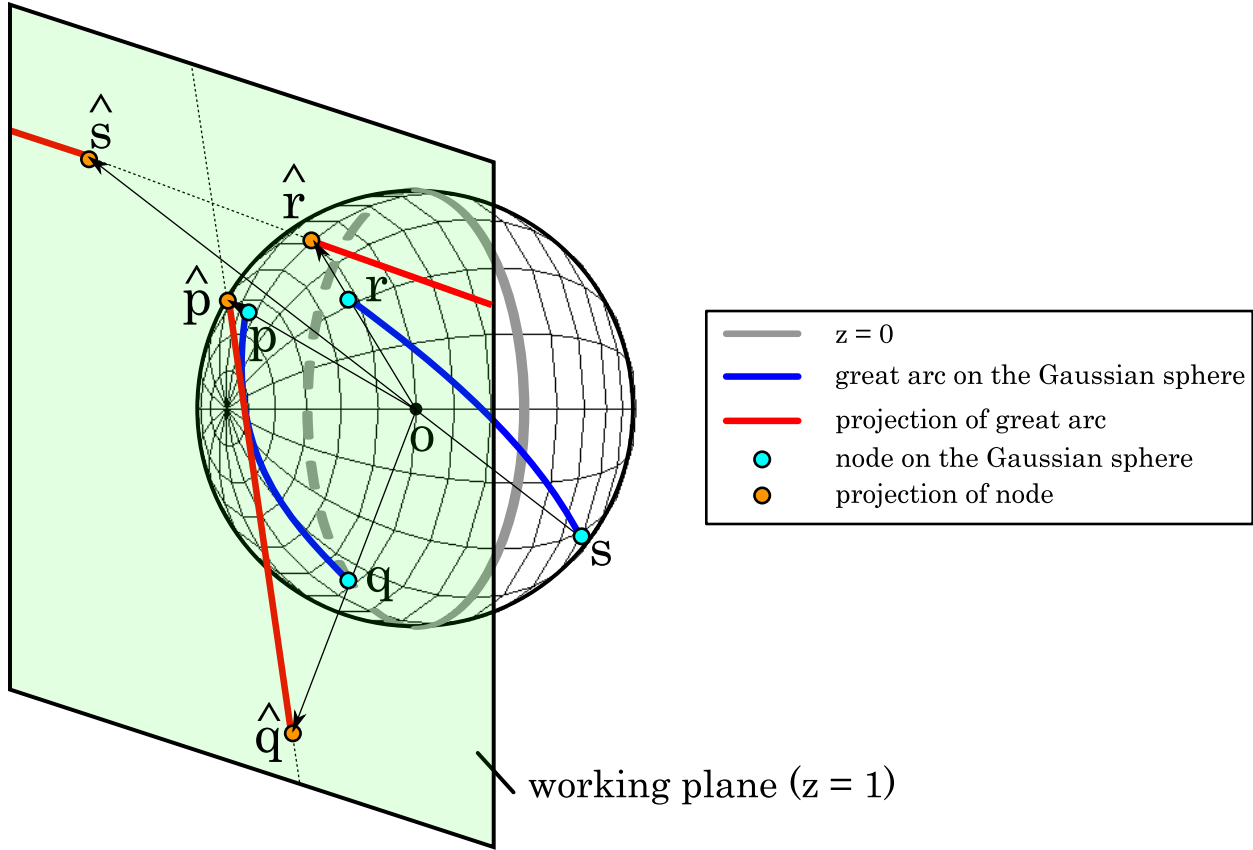


Figure 5.5: A great arc, which is a part of a great circle on the Gaussian sphere, is mapped to a line segment or two half-lines lying on the line obtained by projecting the great circle in the working plane. (The line can be interpreted as the intersection between the working plane and the plane passing through the great circle.) The great arc pq does not pass through any point on the $z = 0$ “equator”; thus, the arc is mapped to the line segment bounded by \hat{p} and \hat{q} . On the other hand, the great arc rs passes through a point at $z = 0$; thus, the arc is mapped to two half-lines, each of which is bounded by \hat{r} and \hat{s} respectively.

trapped at v . We describe how to find the set of rotation axes each of whose corresponding g_r does not intersect ∂T_v in the following subsections.

5.2.2.1 Projecting points from the Gaussian sphere to the 2D plane

We consider the problem of finding the set of all g_r that do not intersect ∂T_v by first projecting from the Gaussian sphere to the 2D “working plane” $z = 1$ using the Gnomonic projection (Figure 5.5). The Gnomonic projection maps the point $p = (p_x, p_y, p_z)$ on the Gaussian sphere to the point $\hat{p} = (p_x/p_z, p_y/p_z)$, which is the intersection point between the working plane and the line passing through the origin and p . Given a set of points P on the Gaussian

sphere, we denote the projection of the set of points as $\Pi(P) = \{\hat{p} \mid p \in P\}$.

The Gnomonic projection maps a great circle to a straight line. Following the projection rule, the great circle c defined by the intersection between the Gaussian sphere and the plane perpendicular to vector $e = (e_x, e_y, e_z)$ is projected to the line $\Pi(c) : e_x x + e_y y + e_z = 0$. (If $e_x = 0$ and $e_y = 0$, $\Pi(c)$ is mapped to infinity.) Therefore, given a rotation axis $r = (r_x, r_y, r_z)$, g_r is projected to the line $\Pi(g_r) : r_x x + r_y y + r_z = 0$.

A great arc a on the Gaussian sphere (e.g. a T_v -arc) is mapped to $\Pi(a)$ in a similar manner. Suppose that a given great arc is a part of the great circle c . As a consequence, the great arc is projected to a line segment or two half-lines lying on the line $\Pi(c)$. Letting $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$ be the great arc's endpoints on the Gaussian sphere, the portion of the line $\Pi(c)$ bounded by \hat{p} and \hat{q} corresponds to the projection of the great arc in the working plane. Figure 5.5 gives an example. Suppose that $e_y \neq 0$ and $\hat{p}_x < \hat{q}_x$, if the great arc pq does not pass through any point at $z = 0$, the line segment is projected to the portion of $\Pi(c)$ where $\hat{p}_x \leq x \leq \hat{q}_x$. If the great arc pq passes through a point at $z = 0$, the line segment is projected to the portion of $\Pi(c)$ where $x \leq \hat{p}_x$ or $x \geq \hat{q}_x$.

A convex spherical polygon bounded by the set of great arcs on the Gaussian spheres is projected to the (possibly unbounded) convex region bounded by the projection of the great arcs. For example, T_v bounded by the set of T_v -arcs $A_{T_{v,j}}$ ($j = 1, 2, \dots, |\partial T_v|$) is mapped to the convex region bounded by $\Pi(A_{T_{v,j}})$ (Figure 5.6 (b)).

5.2.2.2 Duality

To find the set of rotation axes for which a water particle is never trapped at a given concave vertex is to find the set of rotation axes each of whose corresponding g_r does not intersect ∂T_v . Recall that ∂T_v consists of T_v -arcs $A_{T_{v,j}}$. Then, using the Gnomonic projection, the problem is now to find a set of lines in the working plane (each of which corresponds to $\Pi(g_r)$) that do not intersect the line segments (that correspond to T_v -arcs $\Pi(A_{T_{v,j}})$). To solve this problem, we use a method employing a 2D duality transform¹ proposed by Edelsbrunner et al. [Edelsbrunner et al. 1982].

Using this duality transform, a point \hat{p} in the working place (the primal space) is mapped to a line \hat{p}' in the dual space and a line \hat{l} in the primal space is mapped to a point \hat{l}' in the dual space with the following transformation rules:

$$\begin{aligned} \hat{p} : (a, b) &\quad \rightarrow \quad \hat{p}' : y = ax + b \\ \hat{l} : y = kx + d &\quad \rightarrow \quad \hat{l}' : (-k, d) \end{aligned} \tag{5.1}$$

If we apply this rule to a line segment \hat{s} , it is transformed to a (generally wedge-shaped) region $W(\hat{s})$ consisting of the set of lines $W(\hat{s}) = \{\hat{p}' \mid \hat{p} \in \hat{s}\}$ (Figure 5.6 (c)).

When a line \hat{l} intersects a line segment \hat{s} in the primal space, the point \hat{l}' is contained in the region $W(\hat{s})$ in the dual space, and vice versa. Recall that we would like to find a set of lines (i.e. $\Pi(g_r)$) that do not intersect a line segment (i.e. $\Pi(A_{T_{v,j}})$). We can express such

¹The concept of duality transform was first introduced by Brown [Brown 1979].

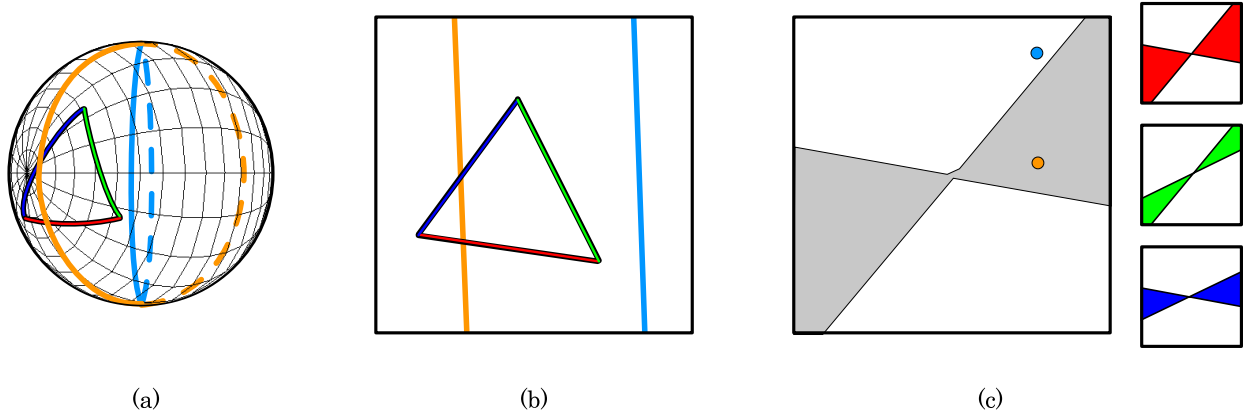


Figure 5.6: Relationships between ∂T_v and great circles (a) on the Gaussian sphere (b) in the working plane (primal space) (c) in dual space (the shaded region corresponds to $\bigcup_j^{|\partial T_v|} W(\Pi(A_{T_{vj}}))$). The subfigures in (c) show each of the T_v -arc's corresponding region $W(\Pi(A_{T_{vj}}))$ in the dual space.

a set of lines in the dual space as the region $[W(\Pi(A_{T_{vj}}))]^c$. Appendix C.2 describes, given an arc a on the Gaussian sphere, how to construct $W(\Pi(a))$. See Figure C.1 for examples of $W(\Pi(a))$.

5.2.2.3 Set of rotation axes where a water particle is never trapped at a concave vertex

Given the great circle representing the gravity directions g_r associated with a rotation axis r , and a concave vertex v , if g_r does not intersect ∂T_v , a water particle will never be trapped at v when we rotate the geometry around r . Since ∂T_v consists of T_v -arcs $A_{T_{vj}}$ ($j = 1, 2, \dots, |\partial T_v|$), we can now express the set of rotation axes where a water particle is never trapped at a given concave vertex v as points in the region $\bigcap_j^{|\partial T_v|} [W(\Pi(A_{T_{vj}}))]^c$, or equivalently $[\bigcup_j^{|\partial T_v|} W(\Pi(A_{T_{vj}}))]^c$. We call this region R_{T_v} . Since there is a one-to-one correspondence between g_r and r for $r_y > 0$, there is no loss of information.

Figure 5.6 shows an example illustrating the relationship among ∂T_v and great circles on the Gaussian sphere, in the working plane, and in the dual space.

5.3 Constructing the extended draining graph

Now, we explain how to find a set of rotation axes that satisfy the second condition for a concave vertex v : the set consists of a rotation axis r for which the corresponding node $N(v)$ in the extended draining graph has a path to the out node such that, for all edges E in the path, $r \in R_E$ (recall that R_E is the set of rotation axes which causes the transition indicated by E).

We describe how to construct the extended draining graph in this section, and how to use it to find such rotation axes in section 5.4.

5.3.1 Directed edges in the extended draining graph

To set directed edges in the extended draining graph, for each concave vertex $v \in V_c$, we first find all the possible destination concave vertices $dest(v)$ where a water particle flowing out from v may settle. Once we find the set $dest(v)$, for each concave vertex in $dest(v)$, we set a directed edge from $N(v)$ to the concave vertex's corresponding node.

5.3.1.1 Finding $dest(v)$

We find $dest(v)$ by tracing the paths a water particle takes from v with a gravity direction that lets the trapped water particle at v move to other concave vertices (or exit the workpiece) using the algorithm described in section 4.3.2.2. In chapter 4, we showed that, even when we approximate a water trap at a concave vertex by multiple water particles, it is sufficient to trace the path of the last particle that leaves the concave vertex (called the *core* particle) to determine whether the part will eventually drain under rotation. Therefore, we constructed the draining graph only by tracing the path of the core particle. We do the same for the extended draining graph.

Since the core particle leaves v when the gravity direction g currently in T_v moves along g_r and passes through a point on ∂T_v , the relevant gravity directions used for tracing the paths are the points along ∂T_v on the Gaussian sphere, which we sample. Then, we perform particle tracing with the gravity direction set to each of the sample points.

We process each T_v -arc individually. Given a T_v -arc $A_{T_{v,j}}$, we can obtain arbitrary sample gravity directions along $A_{T_{v,j}}$ by circularly interpolating between $N_{T_{v,j}}$ and $N_{T_{v,j+1}}$ on the Gaussian sphere. We parameterize $A_{T_{v,j}}$ with variable t ($0 \leq t \leq 1$) and define the sample gravity direction $g_j(t)$ on $A_{T_{v,j}}$ such that $g_j(0) = N_{T_{v,j}}$ and $g_j(1) = N_{T_{v,j+1}}$. Specifically, letting α be the angle between the vector $N_{T_{v,j}}$ and the vector $N_{T_{v,j+1}}$, we define $g_j(t)$ as a vector obtained by rotating the vector $N_{T_{v,j}}$ by $t\alpha$ around the vector defined by $N_{T_{v,j}} \times N_{T_{v,j+1}}$.

Given $g_j(t)$, since the core particle leaves v just after the current gravity direction g passes through a point on ∂T_v (not *at* a point on ∂T_v), we actually trace a water particle with $g_j^*(t)$ defined as the point not in T_v and closest to $g_j(t)$ on the Gaussian sphere. Specifically, we obtain $g_j^*(t)$ by rotating $g_j(t)$ by an infinitesimal angle around the axis defined by the cross product $(g_j(t) \times e_i)$ where e_i is the incident edge of v that defines $A_{T_{v,j}}$ (recall that each T_v -arc constituting ∂T_v comes from the boundary of a halfspace H_i defined by incident edge e_i). We let $S_v(g_j)$ be the concave vertex (or the set of concave vertices if the particle splits) where the core water particle settles when we trace the particle with g_j^* from v .

To find $dest(v)$ efficiently, we first make the following observation. Given two different sample points on $A_{T_{v,j}}$, $g_j(t_1)$ and $g_j(t_2)$ ($t_1 < t_2$), when we trace a water particle under corresponding gravity directions $g_j^*(t_1)$ and $g_j^*(t_2)$, if both settle at the same set of concave vertices by passing over the same sequence of triangle edges without falling through the

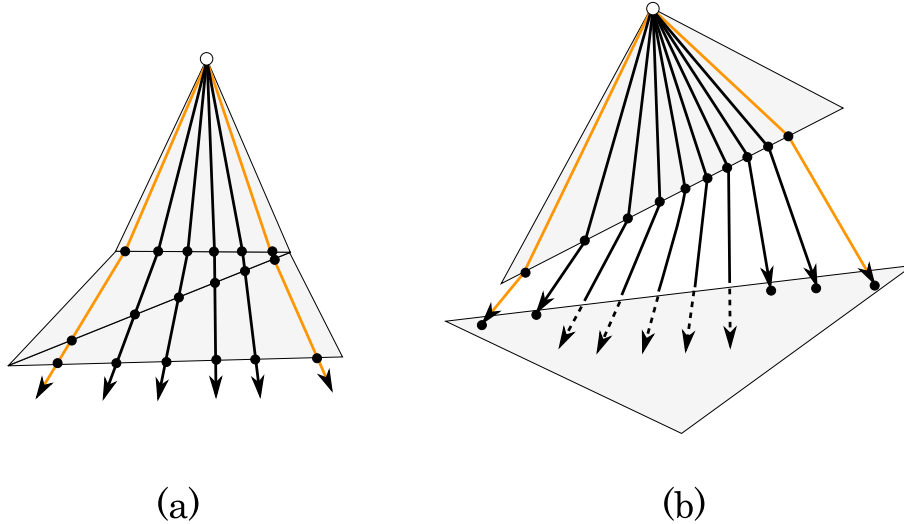


Figure 5.7: (a) Given two different sample points on $A_{T_v j}$, $g_j(t_1)$ and $g_j(t_2)$ ($t_1 < t_2$), when we trace a water particle under corresponding gravity directions $g_j^*(t_1)$ and $g_j^*(t_2)$, if in both cases it passes over the same sequence of triangle edges without leaving the surface (the corresponding paths are shown in orange), any particles traced under intermediate gravity directions $g_j^*(t)$ ($t_1 \leq t \leq t_2$) also pass over the same sequence of triangle edges. (b) However, this is not necessarily true if the water particle falls through the air. (Note that water particles that leave the surface along the same edge of the top triangle pictured, after they fall through the air, will not have colinear intersection points with the plane containing the bottom triangle pictured, because each falls subject to a different gravity direction.)

air (refer to Figure 5.7), any particles traced under gravity directions $g_j^*(t)$ ($t_1 \leq t \leq t_2$) will settle at identical locations, i.e. $S_v(g_j(t))$ contains the same set of concave vertices for $t_1 \leq t \leq t_2$. In this case, we say that $g_j(t_1)$ and $g_j(t_2)$ “set a boundary” on $A_{T_v j}$.

Based on this observation, we find $dest(v)$ by the following procedure (refer to Figure 5.8). For each T_v -arc, $A_{T_v j}$, we initially set $t_{low} = 0$ and $t_{high} = 1$. Then, we trace a water particle under gravity directions $g_j^*(t_{low})$ and $g_j^*(t_{high})$, respectively. If they set a boundary, all the possible destinations when we trace a water particle with $g_j^*(t)$ for $t_{low} \leq t \leq t_{high}$ are found. Otherwise, we trace a water particle under gravity directions $g_j^*(t_{mid})$ where $t_{mid} = (t_{low} + t_{high})/2$. If $g_j^*(t_{low})$ (respectively, $g_j^*(t_{high})$) and $g_j^*(t_{mid})$ set a boundary, we stop. If they do not set a boundary, we take a sample point at $(t_{low} + t_{mid})/2$ (respectively, $(t_{high} + t_{mid})/2$) and perform the same procedure recursively. Our current implementation stops at a minimum user-specified sample distance ² on $A_{T_v j}$. After the above recursion stops, we have multiple sample points on $A_{T_v j}$, $g_j(t_1), \dots, g_j(t_m)$ where m is the number of sample points. Once we perform this for each T_v -arc, we can now express that $dest(v) = \bigcup_{j=1}^{|T_v|} \bigcup_k S_v(g_j(t_k))$.

²We used 0.05 degrees as the minimum to obtain the results shown in section 5.6.

5.3.1.2 Setting directed edges extending from $N(v)$

According to the obtained sample points, we divide T_v -arc $A_{T_v,j}$ into regions G_{jk} such that, when we trace a water particle under gravity directions g^* corresponding to any $g \in G_{jk}$, the location(s) where the water particle settles are the same within the user-specified subdivision limit. Therefore, if there are n different sets of $S_v(g_j(t))$ for $0 \leq t \leq 1$, we divide T_v -arc $A_{T_v,j}$ into n regions, G_{j1}, \dots, G_{jn} . For a portion of $A_{T_v,j}$ bounded by two sample points $g_j(t_p)$ and $g_j(t_{p+1})$ where $S_v(g_j(t_p)) \neq S_v(g_j(t_{p+1}))$, $S_v(g_j(t))$ where $t_p < t < t_{p+1}$ is ambiguous. Since both $S_v(g_j(t)) = S_v(g_j(t_p))$ and $S_v(g_j(t)) = S_v(g_j(t_{p+1}))$ are possible for any $t_p < t < t_{p+1}$, we assume that the portion belongs to both the regions to which $g_j(t_p)$ belongs and to which $g_j(t_{p+1})$ belongs.

Assuming that the sample points $g_j(t_1), \dots, g_j(t_m)$ are sorted such that, for any $k < l$, $t_k < t_l$, we can achieve the divide by scanning $g_j(t_1), \dots, g_j(t_m)$ and, every time we find $g_j(t_p)$ such that $S_v(g_j(t_p)) \neq S_v(g_j(t_{p+1}))$, setting the upper bound of G_{jq} where $g_j(t_p) \in G_{jq}$ at $g_j(t_{p+1})$ and the lower bound of G_{jq+1} where $g_j(t_{p+1}) \in G_{jq+1}$ at $g_j(t_p)$. The lower bound of G_{j1} is set at $g_j(0)$ and the upper bound of G_{jn} is set at $g_j(1)$. Figure 5.8 shows an example of this division. Since $S_v(g)$ is constant for any $g \in G_{jk}$, we define $S_v(G_{jk}) \equiv S_v(g)$ where $g \in G_{jk}$. Then, we can rewrite $dest(v) = \bigcup_{j=1}^{|T_v|} \bigcup_k S_v(G_{jk})$.

Finally, directed edges extending from $N(v)$ are set as follows. For each $\tilde{v} \in S_v(G_{jk})$, we set a directed edge E from $N(v)$ to $N(\tilde{v})$ (or to the “out” node representing the exterior of the part if applicable) in the extended draining graph. At the same time, we assign to E the set of rotation axes R_E that causes the transition of a water particle from $N(v)$ to $N(\tilde{v})$. We explain how to find R_E for G_{jk} in section 5.3.2.

5.3.1.3 Implementation details

Algorithm 10 gives pseudocode for the procedures described above in 5.3.1.1 and 5.3.1.2. In Algorithm 10, subroutine **GetSettleVertices** traces a water particle under a given gravity direction from a given concave vertex and returns the concave vertex (or concave vertices) in which the particle settles (if the particle moves to the exterior of the geometry, the information is also returned) and the path the water particle takes. **IsSamePath** compares given two paths and, if they are different or either involves “falling through the air,” the function returns false. **CollectSamplePoints** shown in Algorithm 11 collect sample points recursively. The subroutine finds the concave vertices where a water particle under the gravity direction corresponding to each sample point settles. Finally, **AddGraphEdge** sets a directed edge in the extended draining graph and assign the set of rotation axes that causes the corresponding transition to the directed edge. The set of rotation axes is determined by the great arc G . In the next subsection, we describe how to find such a set of rotation axes.

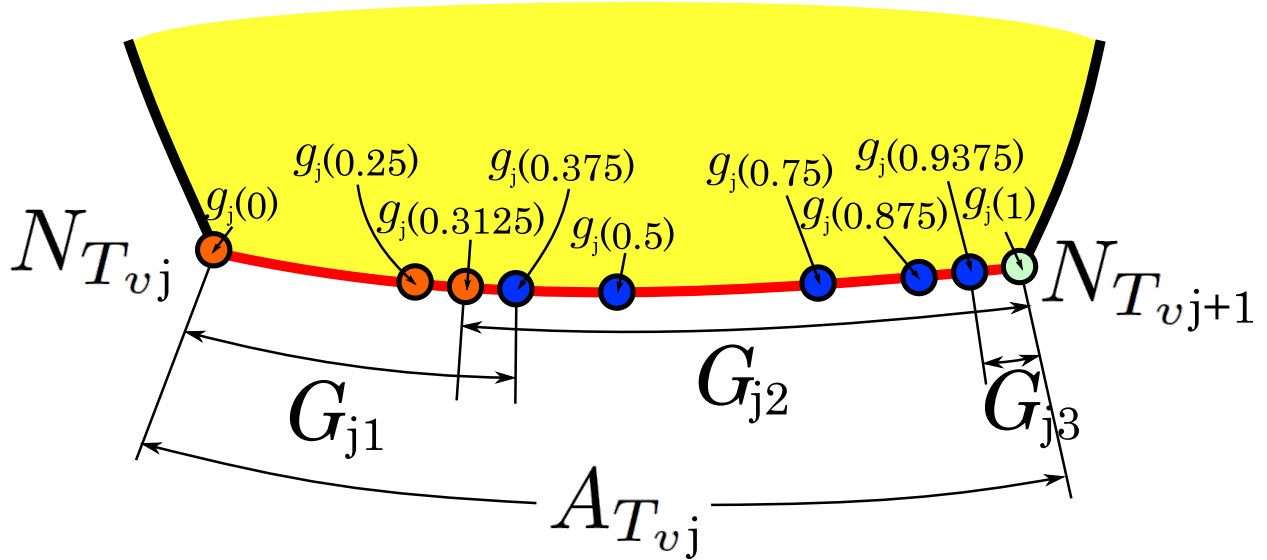


Figure 5.8: T_v -arc $A_{T_{v_j}}$, bounded by two T_v -nodes $N_{T_{v_j}}$ and $N_{T_{v_{j+1}}}$, is divided into multiple regions G_{j_k} such that, when we trace a water particle under gravity directions g^* corresponding to any $g \in G_{j_k}$, the set of concave vertices where the water particle settles is the same. We take sample points $g_j(t)$ along $A_{T_{v_j}}$ recursively, initially setting as $g_j(0) = N_{T_{v_j}}$ and $g_j(1) = N_{T_{v_{j+1}}}$. In the figure, each circle indicates a sample point. Circles with the same color indicate that they have the same set of concave vertices where a water particle settles when we trace a water particle under gravity directions corresponding to each of the sample points. Based on the sample points, $A_{T_{v_j}}$ is divided into three regions G_{j1} , G_{j2} , and G_{j3} in this example.

5.3.2 Set of rotation axes causing water particle transition indicated by directed edge

Letting E be a directed edge in the extended graph, we describe how to find the set of rotation axes R_E that cause the water particle transition indicated by E . Suppose that v is a concave vertex with corresponding node $N(v)$ from which E extends in the extended draining graph. As described in the previous section, directed edge E is set for a particular set of gravity directions G (recall that G is a great arc and portion of ∂T_v on the Gaussian sphere). Geometrically, a rotation axis $r \in R_E$ must have the property that the gravity direction g moving along the corresponding g_r passes through G from the inside to the outside of T_v (refer to Figure 5.4). More specifically, we would like to find a set of great circles (i.e. g_r) each of which intersects a great arc (i.e. G) such that the point (i.e. g) moving along the great circle passes through the great arc from the inside to the outside of a convex spherical polygon (i.e. T_v) lying one side of the great arc.

To consider this problem, we again project geometric features on the Gaussian sphere to the working plane using the Gnomonic projection. In addition, we now have to consider

Algorithm 10 ConstructingExtendedDrainingGraph

```

for each concave vertex  $v \in V_c$  do
  for each  $T_v$ -arc,  $A_{T_v j}$  do
     $g_j(0) \leftarrow N_{T_v j}$ 
     $g_j(1) \leftarrow N_{T_v j+1}$ 
     $[S_v(g_j(0)), path(g_j(0))] \leftarrow \text{GetSettleVertices}(v, g_j(0))$ 
     $[S_v(g_j(1)), path(g_j(1))] \leftarrow \text{GetSettleVertices}(v, g_j(1))$ 
     $P_{sample} \leftarrow g_j(0) \cup g_j(1)$ 
    if  $\text{IsSamePath}(path(g_j(0)), path(g_j(1))) = \text{false}$  then
       $P_{sample} \leftarrow P_{sample} \cup \text{CollectSamplePoints}(v, 0, 1)$ 
    end if
     $m \leftarrow |P_{sample}|$ 
    Sort the sample points  $g_j(t_1), \dots, g_j(t_m) \in P_{sample}$  such that, for any  $k < l$ ,  $t_k < t_l$ 
     $l \leftarrow g_j(0)$ 
    for  $k = 1$  to  $m$  do
      if  $S_v(g_j(t_k)) \neq S_v(g_j(t_{k+1}))$  then
         $u \leftarrow g_j(t_{k+1})$ 
         $G \leftarrow$  great arc on  $A_{T_v j}$  bounded by  $l$  and  $u$ 
        for each concave vertex  $\tilde{v}$  in  $S_v(g_j(t_k))$  do
           $\text{AddGraphEdge}(v, \tilde{v}, G)$ 
        end for
         $l \leftarrow g_j(t_k)$ 
      end if
    end for
     $u \leftarrow g_j(1)$ 
     $G \leftarrow$  great arc on  $A_{T_v j}$  bounded by  $l$  and  $u$ 
    for each concave vertex  $\tilde{v}$  in  $S_v(g_j(t_m))$  do
       $\text{AddGraphEdge}(v, \tilde{v}, G)$ 
    end for
  end for

```

the direction vector $\mathbf{d}(\hat{g})$ in the working plane describing the motion of the projection of the gravity direction \hat{g} along $\Pi(g_r)$ (Figure 5.9). Recalling that we are assuming $r_y > 0$, if the gravity direction g moves clockwise around r along g_r when seen from (r_x, r_y, r_z) , the x-component of $\mathbf{d}(\hat{g})$ becomes negative. If the gravity direction g moves counterclockwise around r along g_r , the x-component of $\mathbf{d}(\hat{g})$ becomes positive. See Appendix C.3 for the proof of this statement.

In the working plane, our problem is now to find a set of lines, each of which corresponds to $\Pi(g_r)$, that intersect a line segment $\Pi(G)$ such that a point \hat{g} moving along $\Pi(g_r)$ passes through $\Pi(G)$ from the inside to the outside of the convex spherical polygon $\Pi(T_v)$ lying to

Algorithm 11 CollectSamplePoints

Input: concave vertex v , the parameters of the lower/upper bounds t_{low} and t_{high} .
Output: set of sample points

if The angle between $g_j(t_{low})$ and $g_j(t_{mid})$ is smaller than the user-specified threshold
then
 return \emptyset
end if

$t_{mid} \leftarrow (t_{low} + t_{high})/2$
 $g_j(t_{mid}) \leftarrow (g_j(t_{low}) + g_j(t_{high}))/2$
 $g_j(t_{mid}) \leftarrow g_j(t_{mid})/\|g_j(t_{mid})\|$
 $[S_v(g_j(t_{mid})), path(g_j(t_{mid}))] \leftarrow \text{GetSettleVertices}(v, g_j(t_{mid}))$
if IsSamePath($path(g_j(t_{low}))$, $path(g_j(t_{mid}))$) = false **then**
 $P_1 \leftarrow \text{CollectSamplePoints}(v, t_{low}, t_{mid})$
else
 $P_1 \leftarrow \emptyset$
end if

if IsSamePath($path(g_j(t_{high}))$, $path(g_j(t_{mid}))$) = false **then**
 $P_2 \leftarrow \text{CollectSamplePoints}(v, t_{mid}, t_{high})$
else
 $P_2 \leftarrow \emptyset$
end if

return $g_j(t_{mid}) \cup P_1 \cup P_2$

one side of the line segment $\Pi(G)$. To solve this problem, we use the same duality transforms as before.

As described in the discussion in 5.2.2.2, the set of lines (i.e. $\Pi(g_r)$) that intersect a line segment (i.e. $\Pi(G)$) in primal space can be expressed as the region $W(\Pi(G))$ in dual space. We define the region $W^+(\Pi(G))$ as the set of lines intersecting $\Pi(G)$ such that the gravity direction \hat{g} moving along $\Pi(G)$ passes through $\Pi(G)$ from the inside to the outside of the corresponding $\Pi(T_v)$ when the x-component of $\mathbf{d}(\hat{g})$ is positive. We define the region $W^-(\Pi(G))$ for the analogous case when the x-component of $\mathbf{d}(\hat{g})$ is negative. Note that $W^+(\Pi(G)) \cup W^-(\Pi(G)) = W(\Pi(G))$ and $W^+(\Pi(G)) \cap W^-(\Pi(G)) = \emptyset$. Appendix C.4 describes, given a great arc G on the Gaussian sphere, how to construct $W^+(\Pi(G))$ and $W^-(\Pi(G))$.

5.3.2.1 Set of rotation axes that cause the water particle transition indicated by directed edge

Given a directed edge E and the set of gravity directions G , we can now express that $R_E = W^+(\Pi(G))$ for the case that we rotate the workpiece clockwise around a rotation axis and $R_E = W^-(\Pi(G))$ for the case that we rotate the workpiece counterclockwise around

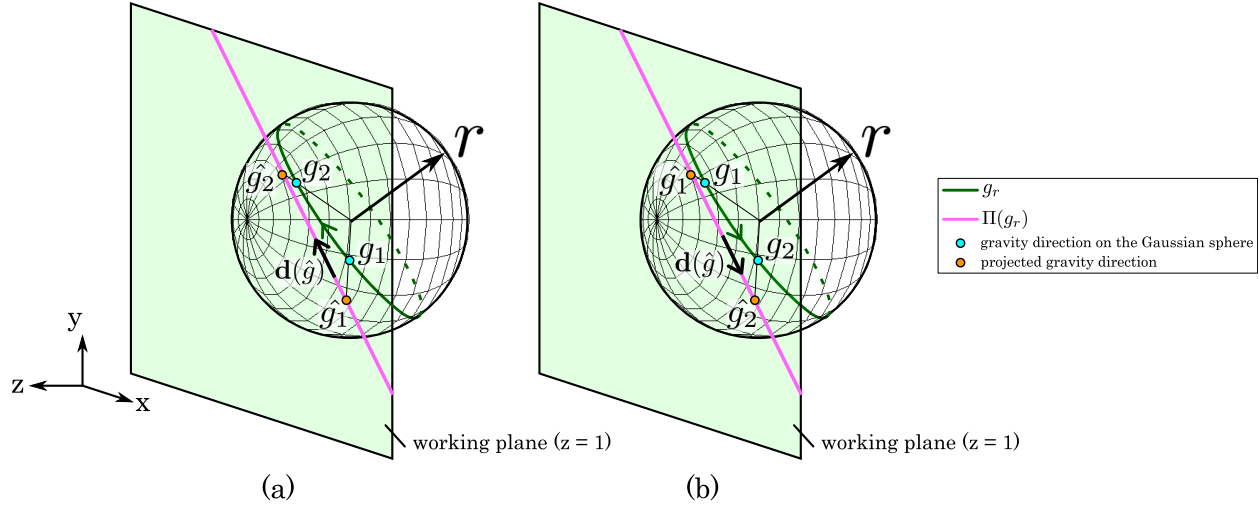


Figure 5.9: The direction vector $\mathbf{d}(\hat{g})$ in the working plane describes the motion of the projection of the gravity direction \hat{g} along $\Pi(g_r)$. As g moves along g_r from g_1 to g_2 , \hat{g} moves along $\Pi(g_r)$ from \hat{g}_1 to \hat{g}_2 . (a) If the gravity direction g moves clockwise around r along g_r when seen from $r = (r_x, r_y, r_z)$, the x-component of $\mathbf{d}(\hat{g})$ becomes negative. (b) If the gravity direction g moves counterclockwise around r along g_r , the x-component of $\mathbf{d}(\hat{g})$ becomes positive.

a rotation axis. Note that when we rotate the workpiece clockwise around r , the gravity direction g rotates around r counterclockwise, and vice versa. Note also that we define R_E as a region in the dual space.

We solve the case of rotating the workpiece clockwise and counterclockwise separately. When we consider the case where we rotate the workpiece clockwise, we set $R_E = W^+(\Pi(G))$; for the other case where we rotate the workpiece counterclockwise, we set $R_E = W^-(\Pi(G))$.

5.4 Rotation axes that drain a trapped water particle at concave vertex v

Given a rotation axis r , even if a water particle is momentarily trapped at concave vertex v , we might be able to drain the water particle by rotating the workpiece around r . This happens when there is a path in the extended draining graph from the corresponding node $N(v)$ to the out node such that, for any directed edge E in the path, $r \in R_E$; we can drain the trapped water particle at v via the sequence of concave vertices corresponding to nodes in this path. We denote the set of such rotation axes as $R_{N(v)}$.

To find $R_{N(v)}$, we find all the paths without cycles from $N(v)$ to the out node where each such path i consists of directed edges E_{ij} such that $\bigcap_j R_{E_{ij}} \neq \emptyset$. Then, we can express $R_{N(v)} = \bigcup_i (\bigcap_j R_{E_{ij}})$. Since we expressed R_E in the dual space, we also express $R_{N(v)}$ in the

dual space (as we did for R_{T_v}).

The cost of finding all the paths without cycles from each node $N(v)$ to the out node is equivalent to the cost of finding all the spanning trees whose root is the out node. The cost is linear in the number of spanning trees of a given graph [Kapoor and Ramesh 2000] and the number of spanning trees can be n^{n-2} for a complete graph where n is the number of nodes in the graph; this is too big to compute in practice. Therefore, we compute $R_{N(v)}$ for each concave vertex v in a greedy manner by traversing the extended draining graph backward from the out node as explained below.

Since computing $R_{N(v)}$ is costly, when we compute it, we only take into account the rotation axes we are interested in³. We define a bounding box B in the dual space, which represents such a subset of rotation axes. Then, we describe how to compute $B \cap R_{N(v)}$ in the following subsection.

5.4.1 Computing $R_{N(v)}$

We compute $B \cap R_{N(v)}$ in a greedy manner as detailed in Algorithm 12. The algorithm takes as its input the bounding box B in the dual space, which limits the set of rotation axes we would like to test. When the algorithm terminates, each $R_{N(v)}$ stores the set of rotation axes in B that can drain concave vertex v . Given a rotation axis r , if the corresponding point lies outside of B in the dual space, r will not be stored in $R_{N(v)}$ even if r can drain a trapped water particle at v .

In the algorithm, we use a priority queue Q each of whose entry stores a node and a set of rotation axes to be propagated through directed edges incoming to the node. Q is first initialized with an entry consisting of the out node and B .

The sets of rotation axes that drain concave vertices are propagated as follows. Given an entry at the front of Q , we let N be the node and R be the set of rotation axes, respectively. Then, we perform the following procedures. For each edge E_j incoming to N and E_j 's source node N_j , we define the set of rotation axes propagated backwards via E_j , R_j , as the intersection of R and the set of rotation axes assigned to E_j i.e. $R_j = R \cap R_{E_j}$. If the intersection is not empty and R_j is not already a subset of the rotation axes that have been assigned to N_j (that is, R_{N_j}), R_j is added to R_{N_j} and (N_j, R_j) is inserted as an entry of Q . We repeat this procedure for each entry while Q is not empty.

In the priority queue Q , the entries are stored in descending order of the area of the polygon (in the dual space) that represents the set of rotation axes. Experimentally, we observed that reducing the number of entries inserted into Q in each iteration is the key to terminate the algorithm as early as possible. Given a set of rotation axes R_j to be propagated to node N_j , if R_j had already been a subset of the rotation axes R_{N_j} , we do not have to propagate R_j further; therefore, we do not have to insert a new entry (N_j, R_j) to Q . Based on this observation, our algorithm tries to propagate a large set of rotation axes as early

³We might not be able to use some rotation axes in practice because of physical constraints of the device.

as possible using the priority queue so that we can reduce the number of entries inserted in subsequent iterations.

Note that $R_{N(v)}$ is different depending on whether we rotate a workpiece clockwise or counterclockwise. We have to perform Algorithm 12 for each case separately. Since the only difference is the definition of R_E discussed in section 5.3.2.1, which does not affect the other parts of the discussion in this paper, when we say $R_{N(v)}$, we do not specify whether it is for the clockwise or counterclockwise case.

5.4.1.1 Implementation note

Suppose that we are given a bounding box B defined as a simple polygon in the dual space. Our strategy to compute $B \cap R_{N(v)}$ in practice is to first perform a decomposition, i.e. decompose simple polygon B into a set of convex polygons B_i such that $\bigcup_i B_i = B$ and $B_i \cap B_j = \emptyset$ for any $i \neq j$. Then, setting each of the convex polygon as a bounding box, we compute $B_i \cap R_{N(v)}$ individually. Finally, we obtain $B \cap R_{N(v)}$ by taking the union $\bigcup_i (B_i \cap R_{N(v)})$.

This approach has several advantages. Firstly, this guarantees that any bounding box is convex; this is important to accelerate the computation of $B \cap R_{N(v)}$ because the intersection of two convex polygons always yields a convex polygon. This makes any intersection computation in Algorithm 12 a convex/convex intersection, which is much simpler and faster than performing a general polygon/polygon intersection computation. In addition, given two different bounding boxes B_i and B_j , since the computation of $B_i \cap R_{N(v)}$ and $B_j \cap R_{N(v)}$ can be done independently, we can parallelize the computation of $B_i \cap R_{N(v)}$ and $B_j \cap R_{N(v)}$; this is faster than computing $(B_i \cup B_j) \cap R_{N(v)}$ serially. Finally, recalling $B \cap R_{N(v)}$ is defined as a union of convex polygons, we incrementally store each convex polygon defining $B \cap R_{N(v)}$ in main memory. If the size of the bounding box B is large (i.e. we take into account many rotation axes at once), the memory size might not be enough to store all the convex polygons. We can avoid this problem by decomposing B into several smaller sub-convex polygons B_i until we can store all the convex polygon defining $B_i \cap R_{N(v)}$ for each node $N(v)$ in main memory. We did some experiments to find a better decomposition with respect to the performance and the memory issue (refer to 5.6.2).

5.5 Finding rotation axes

We perform the following procedure completely independently for clockwise and counterclockwise rotation. Note that whereas $R_{\overline{T}_v}$ is common for both the cases, $R_{N(v)}$ is different for the two cases since R_E is defined differently as discussed in section 5.3.2.1.

5.5.1 Condition to Drain Concave Vertex v

In section 5.1.2, we considered the two conditions under which a concave vertex v will drain with rotation axis r . To drain v , r must satisfy one of these conditions. Based on the

Algorithm 12 Compute $B \cap R_{N(v)}$

Input: Region B in the dual space corresponding to the set of rotation axes we would like to test.

Each entry stored in Q consists of a node and a set of rotation axes. The entries in Q are sorted in descending order of the area of the region defining the corresponding set of rotation axes in the dual space.

for each node N in the extended draining graph **do**

$R_N \leftarrow \emptyset$

end for

Insert the entry (N_{out}, B) into Q // N_{out} is the out node

while Q is not empty **do**

$(N, R) \leftarrow$ the entry at the front of Q

Delete the entry from Q

for each edge E_j incoming to node N **do**

$N_j \leftarrow$ source node of E_j

$R_j \leftarrow R \cap R_{E_j}$

if $R_j \not\subseteq R_{N_j}$ **then**

$R_{N_j} \leftarrow R_{N_j} \cup R_j$

Insert the entry (N_j, R_j) into Q

end if

end for

end while

discussion in section 5.2 and section 5.4, we can rewrite the conditions as follows.

1. The dual transformation of $\Pi(g_r)$ is in $R_{\overline{T_v}}$.
2. The dual transformation of $\Pi(g_r)$ is in $R_{N(v)}$.

Recall that $R_{\overline{T_v}}$ is the set of rotation axes where a water particle is never trapped at a given concave vertex v and $R_{N(v)}$ is the the set of rotation axes that can drain the trapped water particle at v via other concave vertices.

By combining these two conditions, we can express the set of rotation axes to drain concave vertex v as $R_{\overline{T_v}} \cup R_{N(v)}$.

5.5.2 A rotation axis to drain the entire workpiece

Given a rotation axis r , if $\Pi(g_r)$ is in $R_{\overline{T_v}} \cup R_{N(v)}$ for each concave vertex $v \in V_c$, we can drain the workpiece by rotating it around the rotation axis r . Then, we can express the set of rotation axes to drain the workpiece as $\bigcap_{v \in V_c} (R_{\overline{T_v}} \cup R_{N(v)})$. If this becomes empty, there is no rotation axis to drain the workpiece. If the result is not empty, any point inside the region corresponds to a rotation axis that will drain the workpiece.

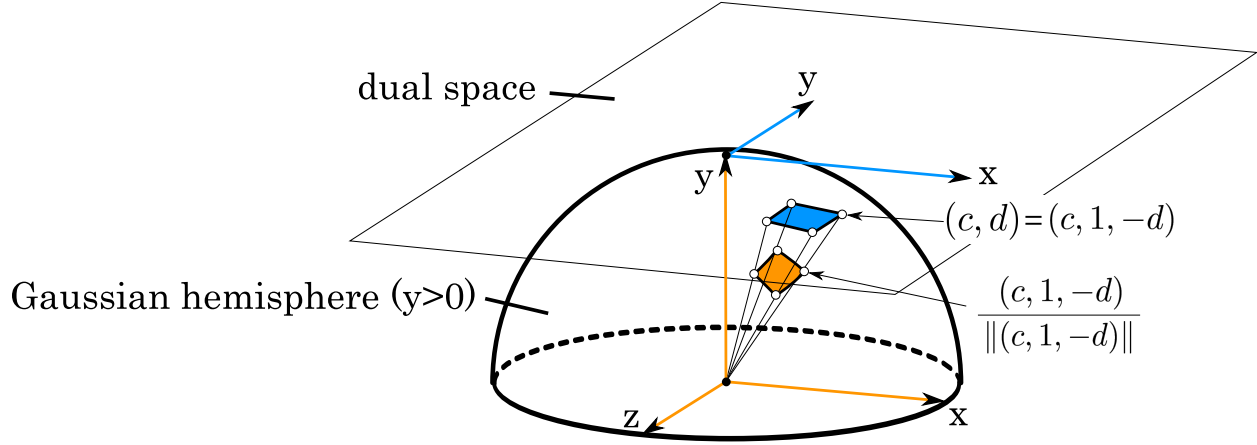


Figure 5.10: We imagine that the dual space is lying such that its origin is at $(0, 1, 0)$ (i.e. the pole of the Gaussian hemisphere), the x-axis of the dual space is parallel to the x-axis of the global coordinate system and pointing in the same direction, and the y-axis of the dual space is parallel to the z-axis of the global coordinate system but pointing in the opposite direction. In this configuration, for any ray emanating from the origin, the corresponding intersection point with the Gaussian sphere and the corresponding intersection point with the dual space represent the same rotation axis. From this relationship, we can observe that the set of rotation axes represented by a continuous region on the Gaussian sphere (the orange region) is also represented by a continuous region in the dual space (the blue region).

Once we compute $\bigcap_{v \in V_c} (R_{T_v}^- \cup R_{N(v)})$, we can check whether the given rotation axis $r = (r_x, r_y, r_z)$ drains the workpiece as follows. Suppose that we are given a point (c, d) in the dual space. The line in the working plane transformed into (c, d) is $y = -cx + d$, that is, $cx + y - d = 0$. For rotation axis $r = (r_x, r_y, r_z)$, g_r on the Gaussian sphere is projected to the line $\Pi(g_r)$: $r_x x + r_y y + r_z = 0$ in the working plane. By comparing each term of these two equations, we find that the point (c, d) in the dual space corresponds to the rotation axis $r = (c, 1, -d) / \|(c, 1, -d)\|$ on the Gaussian sphere (recall that we restrict the y-component of the rotation axis to be positive). Based on this relationship, we can decide whether $r = (r_x, r_y, r_z)$ can drain the workpiece by checking whether the point $(r_x/r_y, -r_z/r_y)$ is contained in $\bigcap_{v \in V_c} (R_{T_v}^- \cup R_{N(v)})$ in the dual space.

Interestingly, we can interpret the relationship geometrically as follows (Figure 5.10). Let us call the coordinate system where the Gaussian sphere is lying the global coordinate system. We can imagine that the dual space is lying in the global coordinate such that its origin is at $(0, 1, 0)$, the x-axis of the dual space is parallel to the x-axis of the global coordinate system and pointing in the same direction, and the y-axis of the dual space is parallel to the z-axis of the global coordinate system but pointing in the opposite direction.

In this layout, the coordinate (c, d) in the dual space corresponds to the coordinate $(c, 1, -d)$ in the global coordinate system. As a result, for any ray emanating from the origin, its intersection point with the Gaussian sphere and its intersection point with the dual space

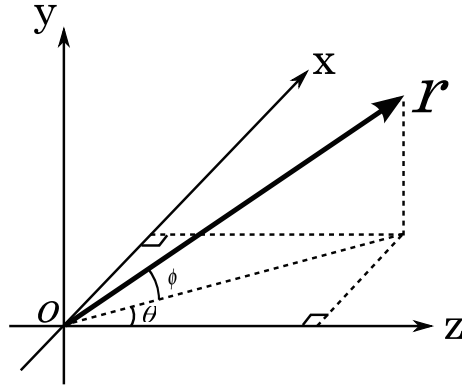


Figure 5.11: We describe any rotation axis relative to workpiece geometry $r = (r_x, r_y, r_z)$ by two variables θ ($0^\circ \leq \theta < 360^\circ$) and ϕ ($0^\circ \leq \phi \leq 90^\circ$) where θ is the azimuthal angle in the xz -plane from the z -axis and ϕ is the polar angle from the xz -plane.

represent the same rotation axis. To see this, imagine that the ray emanating from the origin and passing through the point (c, d) in the dual space, that is, $(c, 1, -d)$ in the global coordinate system. Then, the global coordinate of the intersection point between the ray and the Gaussian sphere is $(c, 1, -d)/\|(c, 1, -d)\|$; this is the coordinate of the corresponding rotation axis expressed on the Gaussian sphere.

From this result, we can observe that there is a one-to-one correspondence between a rotation axis represented as a point on the positive Gaussian hemisphere and a rotation axis represented as a point in the dual space. In addition, the set of rotation axes represented by a continuous region on the positive Gaussian hemisphere is also represented by a continuous region in the dual space (Figure 5.10).

We can also express rotation axes on the positive Gaussian hemisphere by two variables θ ($0^\circ \leq \theta < 360^\circ$) and ϕ ($0^\circ \leq \phi \leq 90^\circ$) where θ is the azimuthal angle in the xz -plane from the z -axis and ϕ is the polar angle from the xz -plane as shown in Figure 5.11, the components of r can be expressed as $r_x = \cos \phi \sin \theta$, $r_y = \sin \phi$, and $r_z = \cos \phi \cos \theta$. In this notation, we can decide whether $r[\theta, \phi]$ can drain the workpiece by checking whether the point $(\sin \theta / \tan \phi, -\cos \theta / \tan \phi)$ is inside of $\bigcap_{v \in V_c} (R_{T_v}^- \cup R_{N(v)})$ in the dual space. Conversely, a point (c, d) in the dual space corresponds to the rotation axis $r[\theta, \phi] = r[\tan^{-1}(-\frac{c}{d}), \tan^{-1}(\frac{1}{\sqrt{c^2+d^2}})]$.

5.5.3 Visualization of Results

Defining the drainable region as the set of points in the dual space corresponding to the rotation axes that can drain the workpiece, we visualize the result of our algorithm by rendering the non-drainable region. Since the drainable region is expressed as $\bigcap_{v \in V_c} (R_{T_v}^- \cup R_{N(v)})$, the non-drainable region can be expressed as its complement, rewritten as follows:

$$[\bigcap_{v \in V_c} (R_{\overline{T}_v} \cup R_{N(v)})]^c = \bigcup_{v \in V_c} [(R_{\overline{T}_v} \cup R_{N(v)})]^c = \bigcup_{v \in V_c} ([R_{\overline{T}_v}]^c \cap [R_{N(v)}]^c). \quad (5.2)$$

Defining $R_{T_v} = [R_{\overline{T}_v}]^c$ and using the relationship $S \cap T^c = S \setminus T$,

$$(5.2) = \bigcup_{v \in V_c} (R_{T_v} \cap [R_{N(v)}]^c) = \bigcup_{v \in V_c} (R_{T_v} \setminus R_{N(v)}). \quad (5.3)$$

Suppose we choose a convex polygon as the bounding box B . Then, we can express the non-drainable region bounded by B using the relationship $[\bigcap_{v \in V_c} (R_{\overline{T}_v} \cup R_{N(v)})]^c = \bigcup_{v \in V_c} (R_{T_v} \setminus R_{N(v)})$ derived from (5.2) and (5.3), the distributive law, and the relationship $S \cap (T \setminus V) = (S \cap T) \setminus (S \cap V)$ ⁴ as follows:

$$\begin{aligned} B \cap [\bigcap_{v \in V_c} (R_{\overline{T}_v} \cup R_{N(v)})]^c &= B \cap [\bigcup_{v \in V_c} (R_{T_v} \setminus R_{N(v)})] = \bigcup_{v \in V_c} [B \cap (R_{T_v} \setminus R_{N(v)})] \\ &= \bigcup_{v \in V_c} [(B \cap R_{T_v}) \setminus (B \cap R_{N(v)})]. \end{aligned} \quad (5.4)$$

Based on (5.4), we can render the non-drainable region bounded by B as follows. We first define the viewport covering B as the frame buffer and initially set the color of each pixel in the color buffer to the color of the drainable region (e.g. green). Then, for each concave vertex v , we clear stencil buffer S and then render $B \cap R_{N(v)}$ to S . Next, we render $B \cap R_{T_v}$ to the portion of the color buffer where not masked by S with the color of the non-drainable region (e.g. red); hence, the pixels not covered by $B \cap R_{N(v)}$ but covered by $B \cap R_{T_v}$ become red. After performing this series of operation for all the concave vertices, the set of green pixels represent the drainable region and the set of red pixels represent the non-drainable region. We take this approach because both $B \cap R_{N(v)}$ and $B \cap R_{T_v}$ can be rendered easily and cheaply since they are expressed as the union of convex polygons.

For $B \cap R_{N(v)}$, we have discussed in section 5.4.1.1 that, when B is a convex polygon, the intersection computation step in Algorithm 12 ($R_j \leftarrow R \cap R_{E_j}$), because it intersects convex polygons, always yields a convex polygon. The final result $B \cap R_{N(v)}$ is the union of such convex polygons (corresponding to $R_N \leftarrow R_N \cup R$). Since each convex polygon that constitutes $B \cap R_{N(v)}$ is rendered to set a mask and setting a mask to a pixel that is already masked does not cause a problem, it does not matter even if multiple polygons overlap each other. As a result, we do not have to explicitly compute the union of multiple convex polygons.

For $B \cap R_{T_v}$, we have $(B \cap R_{T_v}) = B \cap [\bigcup_{j=1}^{|\partial T_v|} W(\Pi(A_{T_v,j}))] = \bigcup_{j=1}^{|\partial T_v|} [B \cap W(\Pi(A_{T_v,j}))]$. We always choose a convex polygon as B and $W(\Pi(A_{T_v,j}))$ is an area bounded by two lines possibly crossing each other; therefore, the number of vertices defining $B \cap W(\Pi(A_{T_v,j}))$ becomes at most the number of vertices of B plus five. Hence, we can compute $B \cap W(\Pi(A_{T_v,j}))$

⁴ $S \cap (T \setminus V) = [S \cap (T \setminus V)] \cup (S \cap S^c \cap T) = (S \cap T \cap V^c) \cup (S \cap T \cap S^c) = (S \cap T) \cap (V^c \cup S^c) = (S \cap T) \cap (S \cap V)^c = (S \cap T) \setminus (S \cap V)$.

Algorithm 13 VisualizeResult

Input: Convex bounding box B
 Define the viewport covering B
 Clear the color buffer with the color of the drainable region (e.g. green)
for each concave vertex $v \in V_c$ **do**
 Clear stencil buffer S
 Render each convex polygon constituting $B \cap R_{N(v)}$ to S
 for $j = 1 \rightarrow |\partial T_v|$ **do**
 Compute $B \cap W(\Pi(A_{T_v,j}))$ (consisting of at most two convex polygons) and render them to the portion of the color buffer where not masked by S with the color of the non-drainable region (e.g. red)
 end for
end for

Algorithm 14 FindingRotationAxes

Input: Bounding Box B
for each concave vertex $v \in V_c$ **do**
 Construct ∂T_v
end for
 ConstructingExtendedDrainingGraph()
 Decompose B into convex polygons B_i
for each i **do**
 Compute $B \cap R_{N(v)}(B_i)$ (* computing $B_i \cap R_{N(v)}$ *)
 VisualizeResult(B_i) (* rendering $\bigcup_{v \in V_c} [(B_i \cap R_{T_v}) \setminus (B_i \cap R_{N(v)})]$ *)
 Store the rendering result C_i in graphics memory.
end for
 VisualizeFinalResultOnGaussianSphere($\bigcup_i C_i$)

in constant time and the result consists of at most two convex polygons. Since each convex polygon(s) $B \cap W(\Pi(A_{T_v,j}))$ is rendered with the same color, it does not matter if multiple polygons overlap each other; again, we do not have to explicitly compute the union of multiple convex polygons.

Algorithm 13 shows the pseudocode for the visualization of results discussed in this subsection.

5.5.4 Implementation

Algorithm 14 shows the overall procedure of our algorithm. We first compute ∂T_v for each $v \in V_c$ (Appendix C.1). Then, we construct the extended draining graph (**ConstructingExtendedDrainingGraph** shown in Algorithm 10). Next, as discussed in section 5.4.1.1, we decompose the input bounding box B into several convex polygons

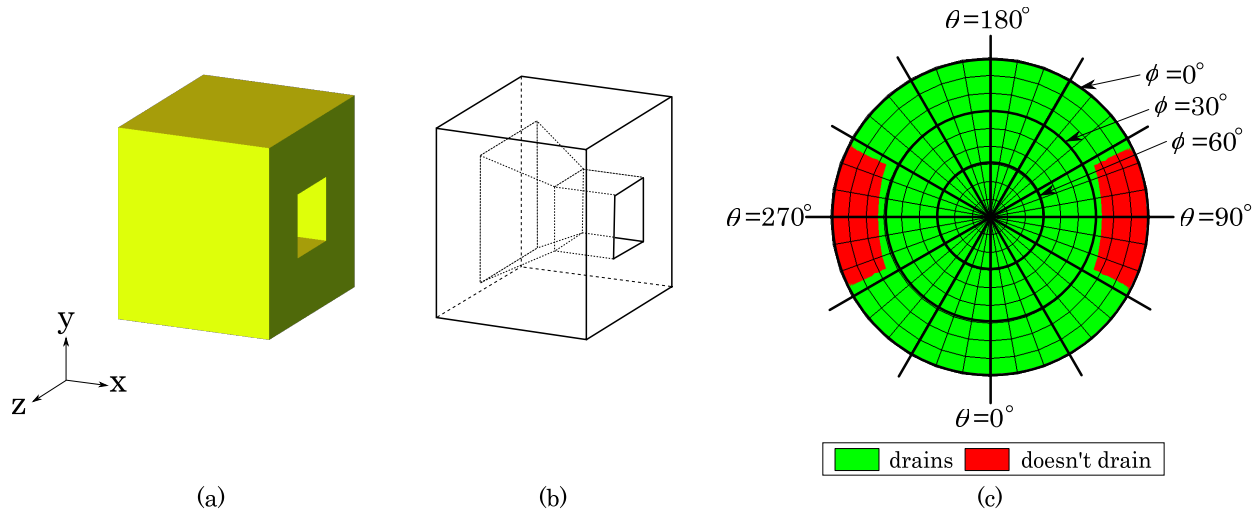


Figure 5.12: (a) Simple test geometry (b) The interior of the geometry. (c) Plot of whether or not rotation around a given rotation axis completely drains the workpiece.

B_i , and compute $B_i \cap R_{N(v)}$ (**Compute** $B \cap R_{N(v)}$ shown in Algorithm 12) and visualize $\bigcup_{v \in V_c} [(B_i \cap R_{T_v}) \setminus (B_i \cap R_{N(v)})]$ (**VisualizeResult** shown in Algorithm 13) individually for each B_i . As also suggested in section 5.4.1.1, we can parallelize this loop since $B_i \cap R_{N(v)}$ and $B_j \cap R_{N(v)}$ ($i \neq j$) can be computed independently of each other. The rendering result is stored in graphics memory (remember that the rendering result is the one in the dual space). Once we perform this for each convex polygon B_i , we can visualize the result, for example, on the hemisphere corresponding to the portion $y > 0$ of the Gaussian sphere (**VisualizeFinalResultOnGaussianSphere**). As discussed in section 5.5.2, a continuous region on the Gaussian sphere corresponds to a continuous region in the dual space. Therefore, we can easily map each of the rendering results in the dual space to the corresponding portion of the Gaussian sphere.

5.6 Results

We first visualize the analysis output for two sample parts, one simple and one complex, and then discuss the performance. For the following results, we set the minimum user-specified distance between two samples on an T_v -arc to 0.05 degrees when we constructed the extended draining graph.

5.6.1 Output

Figure 5.12 shows the result of our algorithm applied to a simple geometry (Figure 5.12 (a) shows the the exterior and (b) shows the interior). For this simple geometry, the set of rotation axes that can drain the geometry is the same whether we rotate the geometry

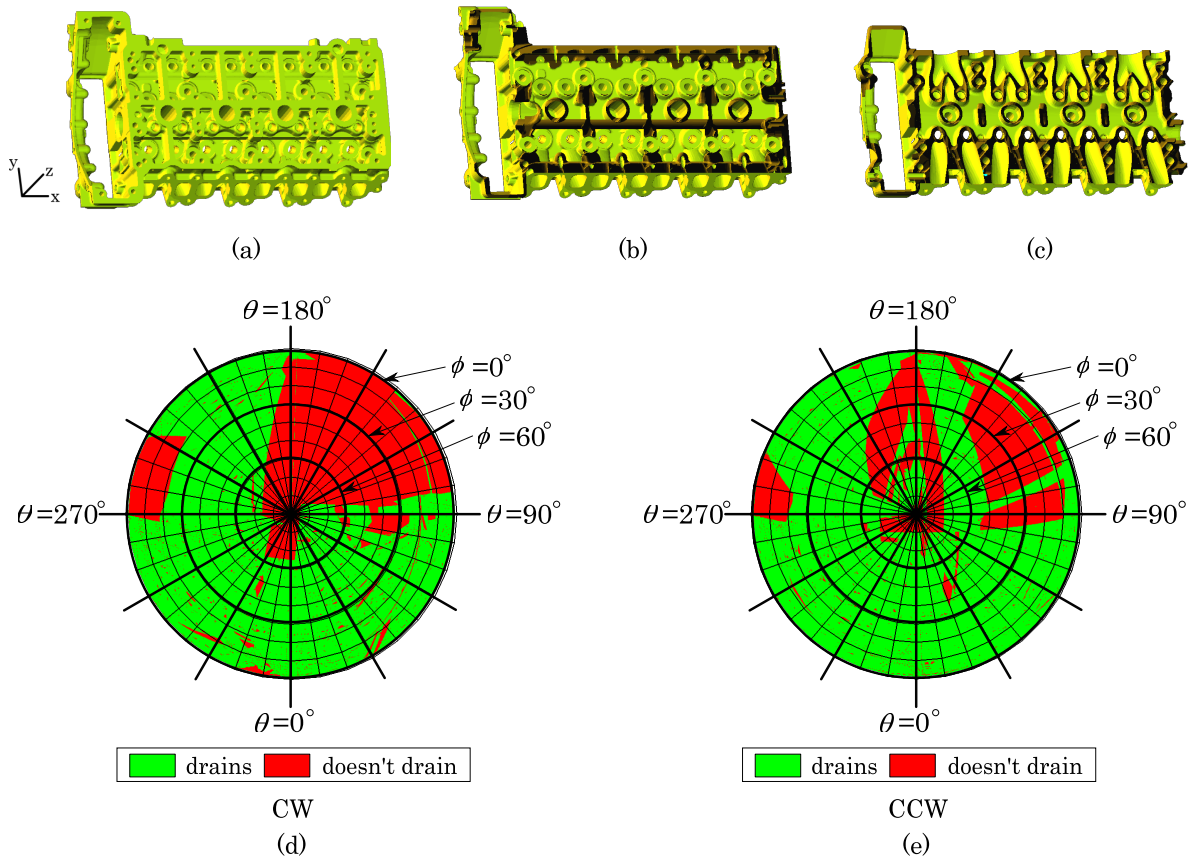






Figure 5.13: (a) Cylinder head model (b)(c) Cross sections revealing the internal passages of the model shown in (a). (d) Plot of whether or not rotation around a given rotation axis completely drains the workpiece under CW rotation and (e) CCW rotation.

clockwise or counterclockwise; therefore, we only show the result of the clockwise case in Figure 5.12 (c).

To examine the result, we plot it on the plane where the $y > 0$ portion of the Gaussian sphere is projected. Each point on the projection corresponds to a unique rotation axis. Given a point on the projection, to determine the drainability of the corresponding rotation axis, we look up the corresponding point in the dual space. If the point has the color of a drainable region (or a non-drainable region), we assigned it to the original point on the projection. In this example, as expected, the results show that rotation axes with relatively large $|r_x|$ cannot drain this geometry.

We also applied our algorithm to the high level-of-detail cylinder head model shown in Figure 5.13 where internal passages (Figure 5.13 (b), (c)) are very complex. Figure 5.13(d) and (e) plot whether or not the indicated rotation axis drains this model.

Table 5.1: Time for computing

	Input characteristics			Time in seconds			
	triangles	vertices	concave vertices	Constructing ∂T_v	Constructing graph	Computing $R_{N(v)}$ and VisualizeResult:	
						total time (average per B_i)	
						CW	CCW
	3,572	1,796	428	< 0.1(sec)	0.3(sec)	24.1 (< 0.1)(sec)	25.5 (< 0.1)(sec)
	12,0004	59,920	18,203	0.9(sec)	36.0(sec)	1694.0 (0.9)(sec)	1638.9 (0.9)(sec)
	160,312	79,982	31,829	2.1(sec)	74.5(sec)	1618.7 (0.9)(sec)	1503.6 (0.8)(sec)
	289,956	144,546	57,412	5.5(sec)	176.8(sec)	5064.7 (2.8)(sec)	5592.3 (3.1)(sec)

5.6.2 Performance

Table 5.1 shows the performance of our implementation running on a quad core 2.66 GHz CPU with 4 GB of RAM. For each geometry, we broke down the timing data between the three steps: constructing ∂T_v for all $v \in V_c$, constructing the extended draining graph, and computing $B \cap R_{N(v)}$ and rendering $\bigcup_{v \in V_c} [(B \cap R_{T_v}) \setminus (B \cap R_{N(v)})]$ (refer to Algorithm 14). Implementation notes for the three steps follow.

5.6.2.1 Implementation note

Constructing ∂T_v : For each concave vertex $v \in V_c$, we construct the corresponding ∂T_v . Since each vertex can be processed independently, we parallelized this operation.

Constructing the extended draining graph: We constructed the extended draining graph by following the procedure described in section 5.3. For each concave vertex $v \in V_c$, we find all the possible concave vertices $dest(v)$ where a water particle flowing out from v may settle. Once we find $dest(v)$, we set a directed edge from node $N(v)$ to each node $N(\tilde{v})$ in $\tilde{v} \in dest(v)$. Since each vertex can be processed independently, we also parallelized this operation. We set the minimum user-specified distance between two samples on a T_v -arc to 0.05 degrees in our implementation.

Computing $B \cap R_{N(v)}$ and rendering $\bigcup_{v \in V_c} [(B \cap R_{T_v}) \setminus (B \cap R_{N(v)})]$: The data shown in Table 5.1 is the time required to test any rotation axes where $0^\circ \leq \theta < 360^\circ$ and $1^\circ \leq \phi < 90^\circ$ (i.e. this set of rotation axes corresponds to B) and visualize the result.

We obtained the timing data by first splitting the portion $y > 0$ of the Gaussian sphere into 36 sections every 10 degrees in the θ direction. Specifically, section i ($1 \leq i \leq 36$) is defined by the three great arcs bounded by $(0, 1, 0)$, $(\cos 1^\circ \sin(10i)^\circ, \sin 1^\circ, \cos 1^\circ \cos(10i)^\circ)$, and $(\cos 1^\circ \sin(10(i+1))^\circ, \sin 1^\circ, \cos 1^\circ \cos(10(i+1))^\circ)$. We further partition each section into 50 patches in the ϕ direction such that the area of each patch is equal on the Gaussian sphere⁵. Therefore, portion $y > 0$ of the Gaussian sphere is ultimately split into $36 * 50 = 1800$ patches, each of which is defined by four vertices on the Gaussian sphere. (For a patch touching the pole $(0, 1, 0)$, the patch is degenerate; it is defined by three vertices since two vertices coincide at the pole.) Notice that each patch is mapped to a convex polygon in the dual space. Then, we process each patch individually, by setting the convex polygon corresponding to the patch in the dual space as a bounding box B_i i.e. we compute $B_i \cap R_{N(v)}$ and render $\bigcup_{v \in V_c} [(B_i \cap R_{T_v}) \setminus (B_i \cap R_{N(v)})]$.

Table 5.1 shows the time to compute $B_i \cap R_{N(v)}$ and render $\bigcup_{v \in V_c} [(B_i \cap R_{T_v}) \setminus (B_i \cap R_{N(v)})]$ for all the 1800 patches (i.e. time to compute $\bigcup_{v \in V_c} [(B \cap R_{T_v}) \setminus (B \cap R_{N(v)})]$). The average time to process one patch is also shown in parentheses next to the corresponding total time in Table 5.1. Since the CPU has four cores, we processed four patches in parallel. We chose to split into 50 patches in the ϕ direction after experimenting with splitting each section into 25 pieces and 100 pieces, for which the computation time was approximately 8% and 14% longer, respectively.

5.7 Complexity Analysis

Letting n be the number of concave vertices, we analyze the complexity of our algorithm.

5.7.1 Constructing ∂T_v

We first consider the complexity of constructing ∂T_v for all the concave vertices $v \in V_c$. For each concave vertex v , if we construct ∂T_v by following the procedure described in Appendix C.1, the complexity is determined by constructing the spherical convex hull of points, corresponding to the incident edge orientations. The complexity to construct the spherical convex hull of m points is $O(m \log m)$ [Chen and Woo 1992]. Since $m = \text{valence}(v)$ and the maximum $\text{valence}(v)$ of any vertex can be taken as constant in practice, and other operations needed to construct ∂T_v are linear in $\text{valence}(v)$, the complexity to construct the corresponding spherical convex hull is also constant. That is, the complexity to construct ∂T_v is constant. Then, the overall complexity to construct ∂T_v for all vertices $v \in V_c$ becomes $O(n)$ in practice.

⁵The proportion of the area of the section's portion bounded by $(0, 1, 0)$, $(\cos \phi \sin(10i)^\circ, \sin \phi, \cos \phi \cos(10i)^\circ)$, and $(\cos \phi \sin(10(i+1))^\circ, \sin \phi, \cos \phi \cos(10(i+1))^\circ)$ to the area of the original section is $(\pi/2 - \phi)^2$ to $(\pi/2)^2$. Using this, we can split the section into several patches such that the area of each patch is equal on the Gaussian sphere.

5.7.2 Constructing the extended draining graph

We next consider the complexity of constructing the extended draining graph.

For each concave vertex v , we set directed edges extending from $N(v)$ by taking multiple samples on the T_v -arcs and tracing the path a water particle takes from v with the gravity direction corresponding to each of the samples. For each trace, since we have to potentially check all the vertices and faces, the complexity of the trace becomes linear in n (assuming that the number of vertices and triangles are proportional to the number of concave vertices). However, we have experimentally shown in chapter 4, that in practice the complexity of tracing is not proportional to n (that is, we can regard the complexity as constant), taking into account the fact that a water particle is driven by only a fixed gravity force and the assumption that the input triangles and vertices are uniformly distributed in space. Therefore, the complexity to find all the edges extending from the corresponding node $N(v)$ is determined by the number of samples we take for that concave vertex. Letting τ be the angle in radians representing the minimum user-specified distance between two samples on a T_v -arc, the number of samples we have to take is at most $2\pi/\tau$, because the sum of the T_v -arcs' length is at most 2π . The average and maximum number of samples we take in practice is shown in the first column of Table 5.2 and 5.3.

Since for each concave vertex v the complexity to find the directed edges extending from $N(v)$ is $O(2\pi/\tau)$, the overall complexity to construct the extended draining graph is bounded by $O((2\pi/\tau)n)$.

5.7.3 Computing $B_i \cap R_{N(v)}$





Finally, we analyze the complexity of computing $B_i \cap R_{N(v)}$ by Algorithm 12.

We first initialize $R_{N(v)}$ to the empty set for each concave vertex $v \in V_c$. This takes $O(n)$.

Then, for each iteration in the while loop, we process entries in Q one by one. Retrieving the entry with the highest priority and deleting the entry from Q takes $O(\log |Q|)$ time where $|Q|$ is the number of entries in Q . Given an entry retrieved from Q , for each directed edge E_j incoming to the node in the entry, we must compute the intersection of two convex polygons in the dual space (i.e. $R \cap R_{E_j}$). Since R is always a convex polygon and R_{E_j} is a region defined by at most two lines, this is bounded by $O(m)$ where m is the number of vertices of the convex polygon representing R . Once we compute the intersection, we perform the containment test $R_j \not\subseteq R_{N_j}$ by testing, for each convex polygon constituting R_{N_j} , whether R_j is contained in the convex polygon. We test whether one convex polygon is completely contained inside of the other convex polygon using a sweep line algorithm in time linear in the sum of the number of vertices constituting each convex polygon [de Berg et al. 2008]. Therefore, letting the number of vertices of each convex polygon constituting R_{N_j} be m_i , the complexity is $O(m + m_i)$. We can now state that the overall complexity to process one directed edge extending from the node in the pair becomes $O(\sum_i(m + m_i))$.

Letting D be the number of directed edges incoming to the node in an entry, we can now state that the complexity to process one entry is $O(D \sum_i(m + m_i))$.

Table 5.2: Detailed complexity data (average)

	concave vertices	#samples	Q		m		$\sum_i(m + m_i)$		D		I	
			CW	CCW	CW	CCW	CW	CCW	CW	CCW	CW	CCW
	428	106.1	99.3	109.4	4.1	4.1	192.0	177.5	7.8	7.8	1,236.2	1,178.4
	18,203	188.9	5,830.8	5,833.9	4.1	4.1	458.8	479.4	8.3	8.3	95,860.9	99,747.2
	31,829	214.0	7,820.3	7,302.0	4.1	4.1	151.8	134.3	8.2	8.2	104,826	96,253.8
	57,412	198.9	21,250.8	22,142.4	4.2	4.2	443.3	534.3	8.8	8.8	323,282	346,544

We do not explicitly compute the union $R_N \cup R$ in our implementation; we just store the convex polygon R in main memory. This process is bounded by the number of vertices defining R , that is, $O(m)$. Since this is contained in $O(D \sum_i(m + m_i))$, this does not affect the overall complexity.

Letting I be the number of iterations, we can now state that the overall complexity to compute $B_i \cap R_{N(v)}$ is $O(n + I[\log |Q| + D \sum_i(m + m_i)])$. The experimental average and maximum number for each of the variables discussed in this subsection are shown in Tables 5.2 and 5.3.





5.8 Discussion

5.8.1 For rotation axes where $r_y = 0$

We have presented/described our algorithm assuming that $r_y > 0$. In this subsection, we briefly consider how to evaluate the rotation axes where $r_y = 0$. When $r_y = 0$, since point $(r_x, 0, r_z)$ and point $(-r_x, 0, -r_z)$ represent the same rotation axis, we restrict the x-component of r to be non-negative.

As you might have noticed, any rotation axes where $r_y = 0$ are mapped to points at infinity in the dual space; therefore, we cannot directly evaluate these axes using our algorithm. Although we can approximately evaluate such a rotation axis by setting $r_y = \epsilon$, a simple and more exact method is to rotate the input geometry, for example, 90 degrees around the z-axis. Then, the rotation axes where $r_y = 0$ in the original configuration are expressed as the rotation axes where $r_x = 0$ in the new configuration. Hence, we can apply our algorithm to determine the drainability of these rotation axes. We compute the drainability of the

Table 5.3: Detailed complexity data (maximum)

	concave vertices	#samples	Q		m		$\sum_i(m + m_i)$		D		I	
			CW	CCW	CW	CCW	CW	CCW	CW	CCW	CW	CCW
	428	4,172	367	1,086	12	11	2,345	2,828	36	36	7,658	10,148
	18,203	6,153	30,593	31,632	13	11	14,463	11,675	141	141	289,636	264,143
	31,829	4,634	41,192	36,250	12	11	7,660	8,597	116	116	340,787	304,655
	57,412	5,129	136,769	165,904	13	13	22,826	23,884	392	392	1,907,760	2,632,350

rotation axes by defining a bounding box that is large enough to cover the rotation axes where $r_x = 0$. Once we obtain the result, we can visualize the result on the portion $y = 0$ of the Gaussian sphere.

We can also evaluate the rotation axis $(0, 0, \pm 1)$ in an analogous manner, for example, by first rotating the input geometry such that $(0, 0, \pm 1)$ coincides with point $(0, 1, 0)$.

5.8.2 Reuse of results

Although we explained how to visualize the results on a computer screen, we might want to use the results as an input to other applications. These other applications might need the boundaries of drainable regions (or non-drainable regions) to be represented by a set of spherical polygons, not just a set of pixels.

We could compute such boundaries by directly evaluating equation (5.4). The running time of constructing the subdivision induced by a set of m lines, called the arrangement, is $O(m^2)$ [de Berg et al. 2008]. Since the arrangement representing the portion $\bigcup_{v \in V_c} (B \cap R_{T_v})$ in equation (5.4) is induced by $\bigcup_{v \in V_c} (\text{valence}(v))$ lines, m could be $\Omega(n)$ where n is the number of concave vertices. Therefore, the running time of directly evaluating equation (5.4) could be $\Omega(n^2)$; this might be costly to compute in practice.

An alternative approach might be to take advantage of our visualization results. Since our result is represented by a set of pixels, we could extract such boundaries from the pixels using the Marching cubes algorithm [Lorensen and Cline 1987] in the dual space. Since the cost of performing the Marching cubes algorithm is merely determined by the number of processed pixels, the cost is independent from the complexity of input geometry and the algorithm runs very fast in practice.

Once we extract polygons representing drainable (respectively, non-drainable) regions in the dual space, we can also convert each of them to the corresponding spherical polygon on the Gaussian sphere by finding, for each vertex and edge defining the polygons, the corresponding point and great arc on the Gaussian sphere.

5.8.3 Performance improvement

We showed that we can find all the possible rotation axes that can drain the workpiece in a reasonable amount of time for manufacturing planning purposes even when the the geometry of the workpiece is complicated; however, since we would like to give interactive feedback to designers in the future, the performance should be much better ideally. Both theoretically and experimentally, we have seen that although we can construct ∂T_v and the extended draining graph relatively quickly, the computation of $B \cap R_{N(v)}$ is a big bottle neck in the computation.

In terms of the performance of computing $B \cap R_{N(v)}$, one encouraging fact is that we do not always have to take into account all the possible rotation axes; because of physical constraints of the device, we might not be able to use some rotation axes in practice. We can compute any local set of rotation axes by setting the bounding box in the dual space that exactly covers the corresponding set, and skip the computation of other sets of rotation axes. If the corresponding region is much smaller than the entire portion $y > 0$ of the Gaussian sphere, we can obtain the result much faster. For example, the time required to process only one patch (corresponding to 1/1800 of all the possible rotation axes) for the most complex geometry is about 3 seconds on average as shown in Table 5.1. Another encouraging fact is that our algorithm to compute $B \cap R_{N(v)}$ is easily parallelized. If we can utilize more cores at the same time, we can solve the problem faster.

Although we pointed out some encouraging facts, let us still think about accelerating the computation of $B \cap R_{N(v)}$ itself. We have shown that the complexity of computing $B \cap R_{N(v)}$ is $O(N + I[\log |Q| + D \sum_i (m + m_i)])$. As Table 5.2 and 5.3 shows, it is largely determined by the number of iterations I . Therefore, reducing the number of iterations is the key to accelerate the computation.

One way to reduce the number of iterations might be reducing the size of the bounding box. Given a bounding box B , suppose that we split B into four smaller bounding boxes B_i ($1 \leq i \leq 4$) and computed $B_i \cap R_{N(v)}$ in parallel on a quad-core CPU. Then, consider further splitting each B_i into four pieces B_{ij} ($1 \leq j \leq 4$) and compute $B_{ij} \cap R_{N(v)}$ in parallel. Although the required number of iterations to compute $B_{ij} \cap R_{N(v)}$ is almost always less than the number of iterations to compute $B_i \cap R_{N(v)}$, unless the former is less than one fourth of the latter, the total number of iterations to compute $B \cap R_{N(v)}$ actually increases. Therefore, although this approach might help, it is not guaranteed to.

We can also take another approach. Recall that we discussed that we can terminate the algorithm early by reducing the number of entries inserted into Q in each iteration in section 5.4.1. One possible approach could be, in Algorithm 12, when we insert the entry (N_j, R_j) into Q , to require not only that the intersection is not empty and that R_j has not already

assigned to N_j via other edges outgoing from N_j , but also require that the area of R_j be larger than some user-defined parameter α . If we set $\alpha = 0$, we can compute $B \cap R_{N(v)}$ precisely. If we set α to some non-zero value, small sets of rotation axes will not be propagated further. As a result, although the region $B \cap R_{N(v)}$ could become smaller than its actual size and some drainable regions might be classified as non-drainable, this approach should definitely reduce the number of iterations. The set of “drainable” rotation axes found for some non-zero α value could even be the same as the set found for $\alpha = 0$. If we only need to find a single rotation axis that can drain the workpiece, this approach could be quite useful.

5.9 Conclusion

In this chapter, we introduced a new approach to find the set of all rotation axes that drains a given workpiece geometry. If it does not exist, our algorithm can also detect that. To the best of our knowledge, this is the first work to tackle the draining problem and to give a reasonable algorithm to solve the problem.

To the best of our knowledge, our work is also the first to establish the theoretical foundation to find great circles intersecting a given great arc when we take into account the direction of points moving along the great circle. We have shown that we can involve this concept while using Gnomonic projection and duality transformation.

Although our algorithm can output results in a reasonable amount of time for manufacturing planning purposes even if the input geometry is complex, we are still looking for an algorithm to solve the problem running, ideally, in real-time. We hope that we can establish such an algorithm based on the findings proposed in this chapter.

Chapter 6

Conclusions and Future Work

In this thesis, we discussed geometric algorithms for cleanability in manufacturing. We believe that our algorithms will not only help manufacturers to investigate an optimal way of cleaning and draining a workpiece with complex geometry, but also help designers to design products whose geometry is compatible with cleaning using high-pressure water jets and draining by rotation. Both from the aspect of design and manufacturing, we expect that our algorithms will improve the efficiency of product design in industry.

Specifically, we proposed the following two problems:

- An efficient algorithm to predict water trap regions of a workpiece in a given orientation.
- A new algorithm to find a rotation axis to fully drain a 3D workpiece.

We wrap up our thesis by briefly summarizing and discussing the future research directions of each of the algorithms.

6.1 Predicting water trap regions using pool segmentation

In chapter 3, we proposed a new pool segmentation data structure based on topological changes of 2D slices with respect to a gravity direction and introduced an algorithm to generate the corresponding segmentation directly from a polygonal mesh. We showed that we can efficiently predict potential water trap regions of a given geometry by analyzing the directed graph representing the gravity-driven water flow movement among the segmented pools. We believe that our algorithm provides an intuitive way for manufacturers to investigate an orientation of the workpiece that minimizes the potential water traps during the cleaning of a workpiece using high-pressure water jets.

6.1.1 Future research directions

We believe that our pool segmentation data structure may also prove useful for other applications analyzing fluid flow inside complex geometries. As future research, let us suggest the possibility of utilizing this data structure to accelerate physics-based simulation, given an inflow location, to track the filling state inside mechanical parts with complex geometries.

Suppose that we consider the fluid flow simulation using smoothed particle hydrodynamics (SPH). To perform highly accurate simulation, we need a lot of particles; however, if we use more particles, it slows the performance of SPH. Although real-time fluid flow simulation employing several tens of thousands of particles on GPUs is now possible in a simple computational domain, performing such simulation in a complex domain in real time is still challenging. Noting that we only need to know the filling-state of water, not the behavior of each water particle, if we use our pool segmentation as an underlying data structure, we believe that we can reduce the size of the computational domain and, as a result, the number of particles we need for the filling state simulation.

Recall that our directed graph represents the direction of water flow driven by gravity. Given a water particle and pool p where the particle is currently located, the destination of the particle driven by gravity must be one of the pools whose corresponding nodes can be reached by following the directed graph from the node corresponding to p . This means that if we cannot reach any of the nodes corresponding to potential water trap regions from the node corresponding to p , we can exclude pool p from the computational domain, since the particle in p will not settle in any pools corresponding to potential water trap regions.

Thus, the computational domain should only consist of pools that have corresponding nodes from which we can reach a node that is a potential water trap region. We can find such pools by traversing the graph in the opposite direction of the graph edges from the node corresponding to each pool found to be a potential water trap region. Since we can exclude other pools from the computational domain, this approach should dramatically reduce the size of the computational domain, especially when the internal passage of the workpiece is complex.

We can further reduce the size of the computational domain based on the current filling state. Recall that we can compute the volume of each pool. Given a particle, once we find the pool where the particle ultimately settles, we store the volume of the particle to the pool. Given a pool that is a potential water trap region, based on the total volume of particles in the pool, we can decide whether it is full or not. Once the pool is full, we delete the corresponding node from the directed graph. As a result, we might be able to further narrow down the computational domain since a pool corresponding to a potential water trap region is removed; the pools categorized as a part of the computational domain because a particle in the pools might settle in the deleted pool could be excluded from the computational domain.

In addition, we can also model air traps formed at concave regions of the voids of complex geometry using our pool segmentation data structure. Determining in which portion of the workpiece air traps are likely to form is important to estimate the cleaning efficiency. The procedure might be as follows. Given a pool p , when p becomes full, we can check whether

air is likely trapped in pools above p by traversing the directed graph from each of those pool's corresponding nodes regardless of the direction of each directed edge (because air particles are not gravity-driven). If we cannot reach the bottommost pool (which represents the exterior) without passing through any node whose corresponding pool is full, there is no path by which the air in the pool could escape. This means air is likely trapped at the pool above p . We could remove such pools from the computational domain of SPH simulation and also delete such pools' corresponding nodes from the directed graph. Since such pools are removed from the computational domain of SPH simulation, water particles never enter the pools. In this way, we can model the air traps formed inside of mechanical parts; we do not have to explicitly model air, for example, with air particles in addition to water particles while performing SPH simulation.

Additionally, we might be able to incorporate solid particles into the model in addition to the water and the air, and provide information about what can and cannot be cleaned based on the flow patterns. Material properties of the substrate and chips, and adhesion strength between the chips and the substrate, could also be incorporated.

6.2 Finding a rotation axis to drain a 3D workpiece

To the best of our knowledge, our work is the first work to tackle the rotational draining problem.

In chapter 4, we presented a new geometric algorithm to test whether rotation around a given rotation axis can drain an input geometry. Observing that all water traps contain a concave vertex, we introduced a draining graph whose nodes correspond to concave vertices of the geometry and whose edges are set according to the transition of trapped water when we rotate the workpiece around the given axis. We showed that we can test a given rotation axis by constructing and analyzing a draining graph in about one second for complex objects.

Based on this algorithm, in chapter 5, we proposed a new approach to find a rotation axis to drain an input geometry. If such a rotation axis does not exist, our algorithm can also detect that. Since we take a configuration space approach, our algorithm can exactly classify all the possible rotation axes. We extended the draining graph such that it takes into account all the possible transition of trapped water. We also introduced a dual-space stabbing line approach to analyze the drainability of all possible rotation axes.

From a theoretical point of view, to the best of our knowledge, our work is the first to establish the theoretical foundation to find great circles intersecting a given set of great arcs while taking into account from which side of a great arc a point moving along the great circle intersects with the great arc. We showed that the approach employing the Gnomonic projection and duality transformations can not only just find a great circle intersecting a given set of great arcs but also distinguish the side from which a point moving along the great circle intersect with the great arcs.

6.2.1 Future research directions

To solve our problem geometrically, we have made a number of simplifying assumptions to make the problem more tractable. The impact of some of our assumptions, such as ignoring the effect of viscosity and friction, must be tested experimentally. These assumptions should be tested and/or relaxed when more sophisticated variations of our algorithm are built on this work.

We have shown theoretically that a rotation axis that drains all the concave vertices must eventually drain the entire workpiece by introducing the concept of a core particle; however, our existing algorithm cannot decide how many rotations are required to drain the workpiece. Given two rotation axes found to drain the workpiece, the number of rotation needed to drain the workpiece might be different. Considering that this information is also useful to minimize the time and energy to drain the workpiece, we would need some modifications to calculate the number of rotations to drain the workpiece.

In terms of the performance of our algorithm, although we have shown that our algorithm can output results in a reasonable amount of time for manufacturing planning purposes even if the input geometry is complex, we are still looking for an algorithm running much faster. We briefly discussed some approaches to improve the performance in 5.8.3. Let us further consider other possibilities to improve the performance here.

We are motivated to improve the performance since, ultimately, we would like to provide real-time monitoring of the design in a CAD system, not just checking the design after it is completed. Responding to the dramatic improvement of CPUs and GPUs, we expect that such functionality will be an essential part of CAD system in the near future. In such a system, undesirable features of a design should be identified and the designer notified of the issue in realtime during design. Therefore, it is important to develop an algorithm that is not only reliable, but also runs fast.

Recall that the performance of our algorithm is largely determined by the number of concave vertices of the workpiece (refer to section 5.6.2). The above-mentioned algorithm, predicting potential water trap regions, might be helpful to reduce the number of concave vertices we have to take into account. If we consider the inflow location and actual filling state, we expect that we would be able to narrow down the computational domain (i.e. the number of concave vertices we take into account) in a similar manner to that discussed in the previous subsection.

In addition, we believe that the most promising approach may be the following. Suppose we are given a concave vertex v all of whose adjacent vertices (in the mesh) are concave vertices and we consider draining a core particle trapped at v by rotating the workpiece. When the workpiece is rotated, a core particle leaving v always momentarily settles at one of its adjacent vertices. This implies that the drainability of v is totally determined by the drainability of v 's adjacent vertices.

This observation suggests that we do not have to explicitly consider the drainability of v because the drainability of v is indirectly determined by the drainability of v 's adjacent vertices. Therefore, we believe that, in the extended draining graph, we can merge the node

corresponding to v into one of the nodes corresponding to v 's adjacent vertices. Since this observation is applicable to any concave vertex all of whose adjacent vertices are concave vertices, the number of nodes in the extended draining graph can be much smaller. (For the models shown in section 5.6, from 32% to 37% of the concave vertices have only concave vertices adjacent to them.) We expect that this approach can improve the performance dramatically while not sacrificing the accuracy of the computation of drainability.

Appendix A

We manage boundary cycles by modifying the data structure introduced by McMains' sweep plane slicing algorithm [McMains and Séquin 1999].

A.1 Vertex classification implementation

We implemented the algorithm using a half-edge data structure [Muller and Preparata 1978]. We represent each edge in \mathcal{W} by two halfedges that are mated. For each vertex v , we define a set of halfedges extending from v , called the disk cycle $disk(v)$, in which the halfedges are ordered clockwise when viewed from the exterior of \mathcal{W} and connected in the form of a circular double-linked list, as shown in Figure A.1.

Each halfedge has a status that is either *beginning* or *ending*. Initially, all halfedges in \mathcal{W} are set as beginning. The status of a halfedge changes to ending halfedge when the destination vertex of the halfedge is processed during sweeping.

When we process a vertex v during sweeping, the type of the vertex is determined by the status of the halfedges in $disk(v)$ (Figure A.2). If all the halfedges are beginning halfedges, the vertex is a *beginning vertex*; if all halfedges are ending halfedges, the vertex is an *ending vertex*. When a vertex has both beginning halfedges and ending halfedges, if all beginning halfedges appear consecutively in its disk cycle (and thus is, all ending halfedges also appear consecutively), the vertex is a *no-change vertex*; otherwise, the vertex is a *merge/split vertex*.

A.2 Boundary cycles implementation

As explained in 3.2.1, a boundary cycle consists of a set of triangles. In our implementation, halfedges are also included in boundary cycles. Given a halfedge, when its origin vertex is processed, if the halfedge's mate is not in any boundary cycles, the halfedge is inserted into a boundary cycle. If the halfedge's mate is in a boundary cycle already, the mate of the halfedge is deleted from that boundary cycle. From the viewpoint of vertex v , we are inserting beginning halfedges in $disk(v)$ into boundary cycles and deleting the mate of each ending halfedge in $disk(v)$ from the corresponding boundary cycle.

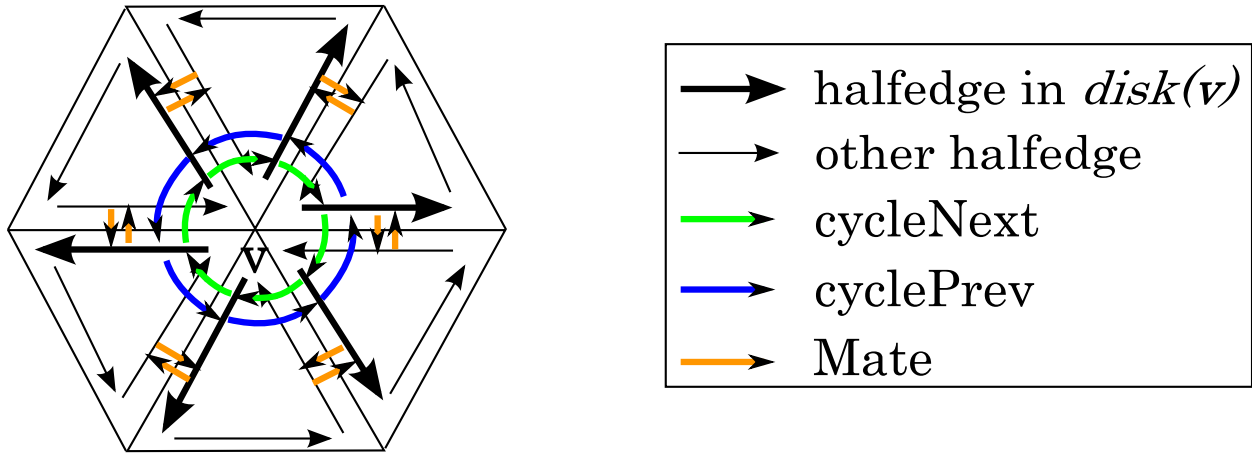


Figure A.1: Given a vertex v , the disk cycle $disk(v)$ is a set of halfedges extending from v . The halfedges in $disk(v)$ are ordered clockwise when viewed from the exterior of \mathcal{W} and connected in the form of circular double-linked list.

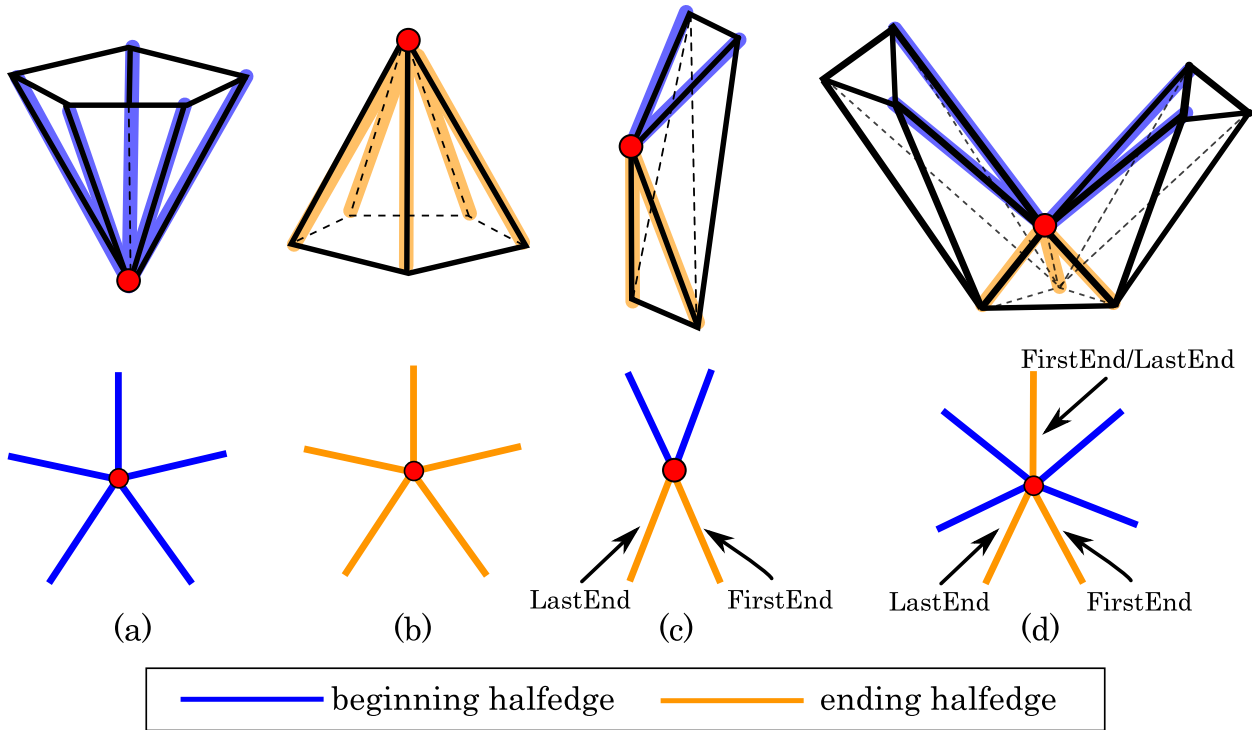


Figure A.2: (a) Beginning Vertex (b) ending vertex (c) no-change vertex (d) merge/split vertex. The bottom row shows the configuration of halfedges in the corresponding disk cycle. We call the first and last ending halfedge of consecutive ending halfedges in the disk cycle *FirstEnd* and *LastEnd*.

Since we process each vertex in ascending order of z-coordinate, halfedges currently in boundary cycles are always intersecting with the sweep plane just as triangles currently in boundary cycles do. Halfedges in each boundary cycle are ordered and connected in the form of a circular double-linked list such that, if we compute the intersection points between these halfedges and $p_{sweep}(z)$ and connect them in that order, we can obtain a closed polygonal chain representing the same slice polygon boundary that the triangles in the same boundary cycle represent at z where $V(z) = \emptyset$.

When we process each vertex in \mathcal{W} , we first replace halfedges in the corresponding boundary cycles. Then, according to the replacement, we replace the triangles in those boundary cycles. We generate, complete, and update boundary cycles depending on the vertex type we encounter during sweeping as follows.

A.2.1 Beginning Vertex

At a beginning vertex v , we generate a new boundary cycle. The halfedges in $disk(v)$ are directly treated as the halfedges in the new boundary cycle (recall that both halfedges in a disk cycle and halfedges in a boundary cycle are connected in the form of circular double-linked lists). Then, triangles adjacent to each halfedge in $disk(v)$ are inserted into the new boundary cycle.

A.2.2 Ending Vertex

At an ending vertex v , we complete the existing boundary cycle to which the mates of halfedges in $disk(v)$ belong.

A.2.3 No-change Vertex

At a no-change vertex v , we update halfedges and triangles in the existing boundary cycle. First, we identify *FirstEnd* and *LastEnd*, which are the first and last ending halfedge of the consecutive ending halfedges in $disk(v)$ (Figure A.2 (c)). Then, we update pointers as follows.

```

FirstEnd->cyclePrev->cycleNext ← FirstEnd->Mate->cycleNext
LastEnd->Mate->cyclePrev->cycleNext ← LastEnd->cycleNext
FirstEnd->Mate->cycleNext->cyclePrev ← FirstEnd->cyclePrev
LastEnd->cycleNext->cyclePrev ← LastEnd->Mate->cyclePrev

```

After this pointer update, we pick up one of the beginning halfedges in $disk(v)$ and traverse the circular linked-list from this beginning halfedge until we revisit it. The halfedges we have visited during the traversal are the set of appropriately ordered halfedges in the boundary cycle after processing v .

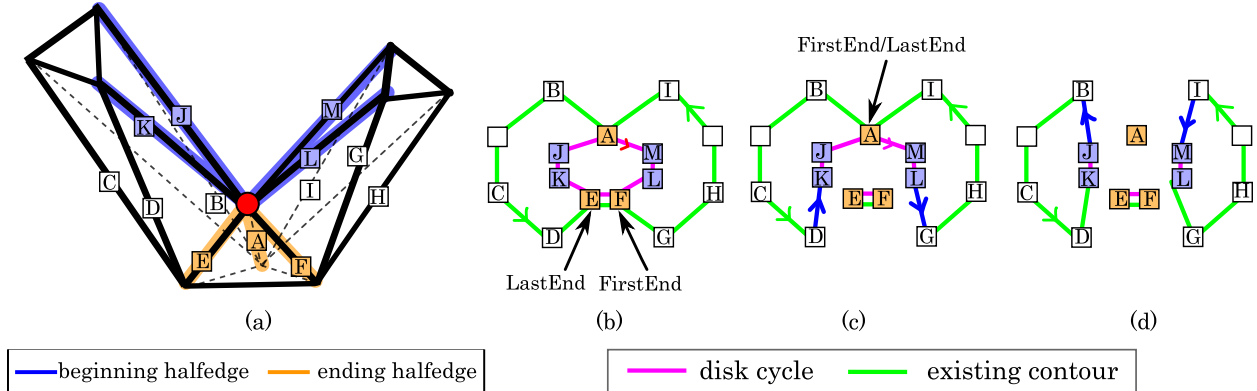


Figure A.3: (a) Model with merge/split vertex v (indicated vertex). Blue edges indicate beginning halfedges and orange edges indicate ending halfedges in $disk(v)$. (b) The halfedges in $disk(v)$ (connected with magenta lines) and the halfedges in the existing boundary cycle (connected with green lines). (c) The change of the pointers after processing the first pair of FirstEnd and LastEnd. (d) The change of the pointers after processing the second pair of FirstEnd and LastEnd (note that FirstEnd and LastEnd are the same halfedge here in this example). After both sets of pointer changes, there are two new boundary cycles: one whose halfedges we can traverse from beginning halfedge J or K belong and one whose halfedges we can traverse from beginning halfedge L or M belong. The mates of halfedges A , E , and, F are no longer in any boundary cycle.

Finally, we check each triangle incident to v . If the triangle is visited for the first time, the triangle is inserted into the boundary cycle. If the triangle is visited for the third time, the triangle is deleted from the boundary cycle.

A.2.4 Merge/split Vertex

At a merge/split vertex v , we first complete all the existing boundary cycle(s) to which the mates of ending halfedges in $disk(v)$ belong. A disk cycle of a merge/split vertex has multiple sets of consecutive ending halfedges (Figure A.2 (d)). First, we identify the FirstEnd and LastEnd of each set. Then, we update pointers in the same manner as for a no-change vertex for each FirstEnd/LastEnd pair (Figure A.3).

After the pointer updates, we pick up one of the beginning halfedges in $disk(v)$ and traverse the circular linked-list from the beginning halfedge until we revisit it. The halfedges we have visited during the traversal are an ordered set of halfedges that belong to a new boundary cycle after processing v . If there is a beginning halfedge in $disk(v)$ that we did not visit during this traversal, then we traverse the circular linked-list from this beginning halfedge until we revisit it. The halfedges we visited during the subsequent traversal are an ordered set of halfedges that belong to another new boundary cycle after processing v . We repeat this procedure until we determine the boundary cycles to which all the beginning

halfedges in $disk(v)$ belong.

Finally, for each new boundary cycle, triangles adjacent to each halfedge in the boundary cycle are inserted into the corresponding boundary cycle.

A.3 Finding boundary cycles generated, completed, and, updated

Finally, we show the pseudocode **ProcessVertices**($V(z)$) that takes $V(z)$ as an input and returns a set of boundary cycles generated, completed, and updated at z in Algorithm 15. In the pseudocode, **ProcessBeginningVertex**(v) generates the new boundary cycle b and returns it. **ProcessEndingVertex**(v) returns the existing boundary cycle b completed at v . **ProcessMergeSplitVertex**(v) changes the pointers of the halfedges in $disk(v)$ (as detailed in A.2.4). According to the result, we assign new triangles to each of the generated boundary cycles, and the set of completed boundary cycles B_C and the set of generated boundary cycles B_G at v are returned. **ProcessNoChangeVertex**(v) changes the pointers of the halfedges in $disk(v)$ and updates the triangles in the existing boundary cycle b accordingly. After the updates, b and triangles inserted into b at v are returned.

Note that there might be boundary cycles generated at one vertex in $V(z)$ and completed at another vertex also in $V(z)$. We call this type of boundary cycle a “degenerate” boundary cycle. Since a degenerate boundary does not define a pool, we remove degenerate boundary cycles from the set of boundary cycles returned by the function. Similarly, there might be boundary cycles generated at one vertex in $V(z)$ and updated at another vertex in $V(z)$. We treat such boundary cycles as generated boundary cycles, not as updated boundary cycles. In a similar manner, we treat boundary cycles updated at a vertex in $V(z)$ and completed at another vertex in $V(z)$ as completed boundary cycles, not as updated boundary cycles. Therefore, U , the set of boundary cycles updated at z , are the ones neither generated nor completed at z .

Notice that no triangle parallel to $p_{sweep}(z)$ is in any boundary cycles returned by the function since a triangle is deleted from the boundary cycle once the triangle is visited three times; all vertices of such a triangle are processed in $V(z)$. Therefore, triangles parallel to $p_{sweep}(z)$ whose corresponding boundary cycles are generated or completed at z do not constitute the boundary of the pool, since such triangles will overlap the bottom face or the top face of the pool. Whereas such triangles are redundant to define the pool boundary, on the other hand, triangles (introduced at no-change vertices) that are parallel to $p_{sweep}(z)$ whose corresponding boundary cycles are *not* generated nor completed at z do constitute the pool boundary.

Algorithm 15 ProcessVertices($V(z)$)

Input: $V(z)$ set of vertices whose z -coordinate is z

Output: G set of boundary cycles generated at z , U set of boundary cycles updated at z ,

C set of boundary cycles completed at z

$G \leftarrow \emptyset$

$C \leftarrow \emptyset$

$U \leftarrow \emptyset$

for each $v \in V(z)$ **do**

if v is a beginning vertex **then**

$b \leftarrow \mathbf{ProcessBeginningVertex}(v)$

$G \leftarrow G \cup b.$

else if v is an ending vertex **then**

$b \leftarrow \mathbf{ProcessEndingVertex}(v)$

$C \leftarrow C \cup b.$

else if v is a merge/split vertex **then**

$(B_C, B_G) \leftarrow \mathbf{ProcessMergeSplitVertex}(v)$

$C \leftarrow C \cup B_C.$

$G \leftarrow G \cup B_G.$

else if v is a no-change vertex **then**

$(b, T_{inserted}) \leftarrow \mathbf{ProcessNoChangeVertex}(v)$

 // assuming initially $(b \rightarrow T_{new}(z)) = \emptyset$

$b \rightarrow T_{new}(z) \leftarrow (b \rightarrow T_{new}(z)) \cup T_{inserted}$

$U \leftarrow U \cup b.$

end if

end for

$D \leftarrow G \cap C$ // D : set of degenerate boundary cycles

$G \leftarrow G \setminus D$

$C \leftarrow C \setminus D$

$U \leftarrow U \setminus (G \cup C \cup D)$

return (G, C, U)

Appendix B

B.1 Boundary of $H_i(xy)$

The boundary of $H_i(xy)$ is defined by the intersection points between the boundary of H_i and the xy-plane Gaussian circle. Let the intersection point be $I = (I_x, I_y, 0)$. Since it is confined on the Gaussian circle, $I_x^2 + I_y^2 = 1$. From the definition, the boundary of H_i is defined by the plane perpendicular to e_i . Letting $e_i = ((e_i)_x, (e_i)_y, (e_i)_z)$, this plane is expressed as $(e_i)_x x + (e_i)_y y + (e_i)_z z = 0$. Then, assuming $(e_i)_x \neq 0$, we can solve for I_x ,

$$\begin{aligned} (e_i)_x(I_x) + (e_i)_y(I_y) + (e_i)_z(0) &= 0 \\ I_x &= -\frac{(e_i)_y}{(e_i)_x} I_y \quad ((e_i)_x \neq 0) \end{aligned}$$

Substituting into $I_x^2 + I_y^2 = 1$,

$$\begin{aligned} \left(\frac{(e_i)_y}{(e_i)_x}\right)^2 I_y^2 + I_y^2 &= 1 \\ \left(\frac{(e_i)_y}{(e_i)_x}\right)^2 + 1) I_y^2 &= 1 \\ I_y &= \pm \sqrt{\frac{1}{\left(\frac{(e_i)_y}{(e_i)_x}\right)^2 + 1}} \quad ((e_i)_x \neq 0) \end{aligned}$$

Note that the boundary of H_i and the Gaussian circle intersect at two points. When $(e_i)_x = 0$, if $(e_i)_y \neq 0$, $I_x = \pm 1$ and $I_y = 0$, and if $(e_i)_y = 0$ as well, the entire xy-plane Gaussian circle defines the boundary of H_i .

B.2 Finding a closest point on a flat region

Among vertices in V_{cand} and points on edges in E_{cand} (V_{cand} and E_{cand} are the candidate vertices and edges where a water particle leaving p_{cur} may flow out through), we find the point p_f that is closest to p_{cur} along edges in E_{perp} and triangles in T_{perp} . Since this problem can be NP-hard [Hershberger and Snoeyink 1994], we solve the problem using a modification of the approximation method proposed by Kallmann [Kallmann 2005]. First, we define a

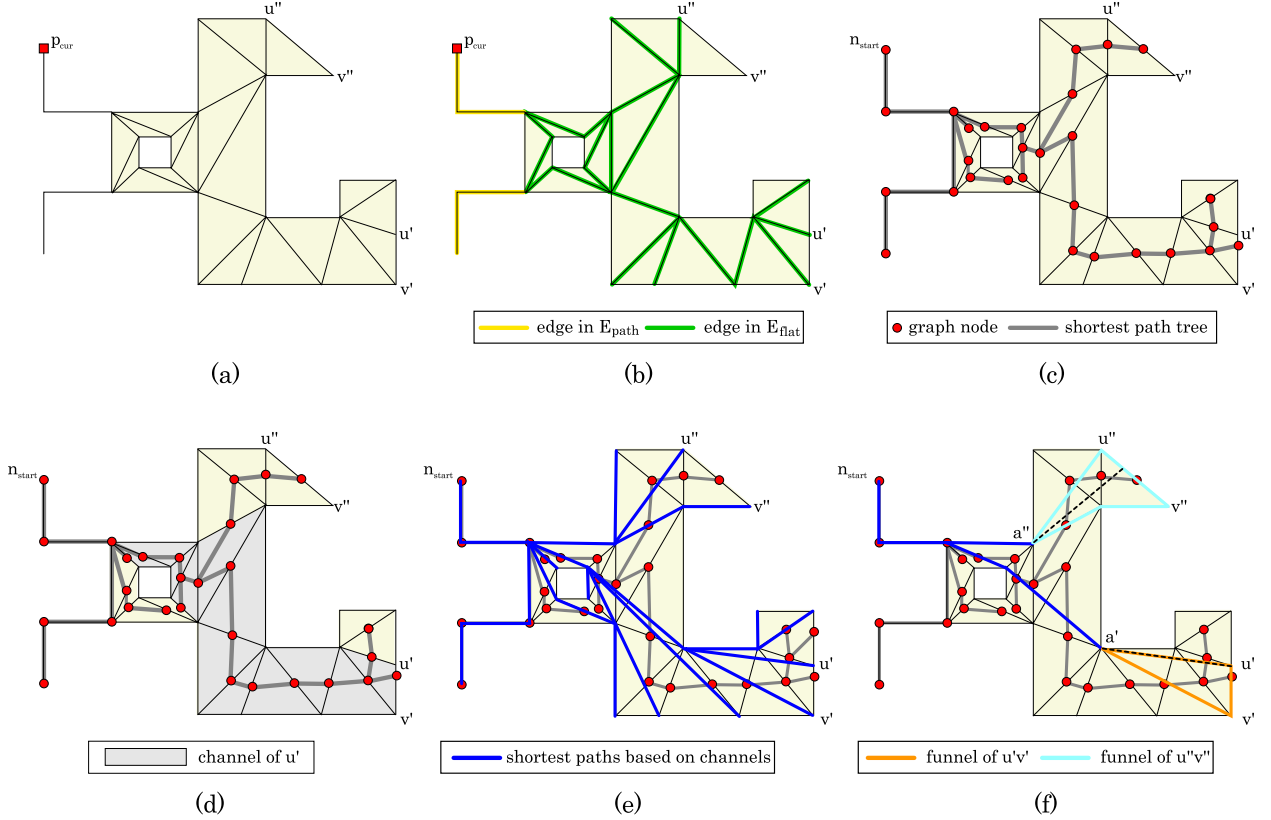


Figure B.1: (a) p_{cur} , edges in E_{perp} , and triangles in T_{perp} . Suppose $E_{cand} = \{\overline{u'v'}, \overline{u''v''}\}$. (b) Edges in E_{path} (yellow) and edges in E_{flat} (green) are highlighted. (c) Shortest path tree from n_{start} (corresponding to p_{cur}) on the graph whose nodes consist of p_{cur} , vertices incident to E_{path} , and midpoints of edges in E_{flat} and E_{cand} . Edges of the graph are those in E_{path} plus edges between any pair of nodes on the same triangle in T_{perp} . (d) Channel of u' . (e) Shortest path from n_{start} to each vertex along the channels. (f) The shortest path from n_{start} on edge $\overline{u'v'}$ (respectively, $\overline{u''v''}$) is the shortest path from the apex of the corresponding funnel a' (respectively, a''). The shortest path from n_{start} to each edge lies on the dashed lines.

set E_{path} , the set of edges in E_{perp} both of whose incident triangles are not in T_{perp} . We also define a set E_{flat} , the set of edges in E_{perp} both of whose incident triangles are in T_{perp} (Figure B.1 (b)). Then, we consider a graph whose nodes consist of p_{cur} , vertices incident to E_{path} , and midpoints of edges in E_{flat} and E_{cand} . The graph's edges are those in E_{path} plus edges between any pair of nodes on the same triangle in T_{perp} . Letting n_{start} be a node corresponding to p_{cur} in the graph, we construct a shortest path tree from n_{start} on the graph using, for example, Dijkstra's algorithm (Figure B.1 (c)). At this point, the minimum distance to each vertex incident to edges in E_{path} is determined; therefore, if T_{perp} is empty (note that E_{cand} is also empty in this case), a vertex in V_{cand} corresponding to a graph node

with a minimum distance on the shortest path tree is p_f . Otherwise, we find the minimum distances to other vertices using the funnel algorithm [Lee and Preparata 1984; Chazelle 1982]. The funnel algorithm finds the shortest path to each vertex inside a *channel*, a chain of triangles along the shortest path tree (Figure B.1 (d)(e)). For an explanation of how the funnel algorithm works, refer to [Hershberger and Snoeyink 1994].

We have to find the shortest path from n_{start} to each edge in E_{cand} , since p_f may be located on some edge in E_{cand} . As shown in Figure B.1 (f), the shortest path from n_{start} to the edge's two endpoints u and v on the corresponding channel travel together and diverge at a vertex a (called the *apex*). The region bounded by edge \overline{uv} and concave chains from u and v to a is called the *funnel* [Hershberger and Snoeyink 1994]. The shortest path from n_{start} to edge \overline{uv} passes through a ; therefore, we can find the shortest path from n_{start} by finding the shortest path from a . If \overline{uv} and a half-line extending from a and perpendicular to \overline{uv} intersect, the line segment between a and the intersection point is the shortest path from a to \overline{uv} (as for a'' and $\overline{u''v''}$ in Figure B.1 (f)). Otherwise, the line segment between a and an intersection point between \overline{uv} and a tangent line of the funnel extending from a is the shortest path. There are two candidates, so we pick the shorter one (an example is the line from a' to u' in Figure B.1 (f)). Finally, the minimum distance from n_{start} to \overline{uv} along edges in E_{perp} and triangles in T_{perp} is the minimum distance from n_{start} to a plus the length of the line segment from the apex.

Now, we find the minimum distance from n_{start} to each vertex in V_{cand} and point in E_{cand} along edges in E_{perp} and triangles in T_{perp} . The candidate vertex or point on a candidate edge that has the minimum distance from n_{start} is p_f .

Appendix C

C.1 Constructing ∂T_v from edges e_i incident to v

Given a concave vertex v , let w_i be a member of the set of adjacent vertices of v in the input geometry's triangulated mesh and $e_i = w_i - v$ ($i = 1, 2, \dots, \text{valence}(v)$). Then, we define $e_i^* = e_i / \|e_i\|$. The points e_i^* are on the Gaussian sphere; the spherical convex hull of these points $SCH(e_1^*, e_2^*, \dots, e_{\text{valence}(v)}^*)$ is the boundary of the smallest convex set on the Gaussian sphere containing $e_1^*, e_2^*, \dots, e_{\text{valence}(v)}^*$ [Preparata and Shamos 1985]. We call each vertex of the spherical convex hull s_j . Chen and Woo showed that each boundary arc of the intersection of the halfspaces $H_1, H_2, \dots, H_{\text{valence}(v)}$ on the Gaussian sphere (that is, each T_v -arc) is a portion of a great circle $g_j = \{p \mid s_j \cdot p = 0, \|p\| = 1\}$ and, if boundary arc A is induced by s_j , each of A 's adjacent boundary arcs is induced by s_{j-1} and s_{j+1} , respectively [Chen and Woo 1992]. That is, if T_v -arc $T_v A_j$ is induced by s_j , $T_v A_{j-1}$ is induced by s_{j-1} and $T_v A_{j+1}$ is induced by s_{j+1} . This implies that the number of vertices of the spherical convex hull is equal to $|\partial T_v|$.

Based on this, we can find T_v -nodes defining ∂T_v by the following procedures. Given a concave vertex v , for each e_i , we first compute e_i^* . Then, we compute the spherical convex hull of the points $e_1^*, e_2^*, \dots, e_{\text{valence}(v)}^*$ (with the procedure described by Chen and Woo. [Chen and Woo 1992]). Then, we compute the centroid of the the spherical convex polygon's vertices and find the intersection point c between the vector passing through the centroid and the Gaussian sphere. We can guarantee that the intersection point c is in the interior of the spherical convex polygon.

Now, each T_v -node $T_v N_j$ is computed as follows. Suppose that T_v -arc $T_v A_j$ corresponds to a vertex of the spherical convex polygon s_j . Then, $T_v A_j$ consists of points p such that $p \cdot s_j = 0$; Since T_v -node $T_v N_j$ is an intersection point between $T_v A_j$ and $T_v A_{j-1}$, $T_v N_j$ must hold $T_v N_j \cdot s_{j-1} = 0$ and $T_v N_j \cdot s_j = 0$. That means, $T_v N_j$ must be a point expressed as either $n_1 = (s_{j-1} \times s_j) / \|s_{j-1} \times s_j\|$ or $n_2 = -(s_{j-1} \times s_j) / \|s_{j-1} \times s_j\|$; the one that has a negative dot product with the interior point c corresponds to $T_v N_j$. That is, $T_v N_j = n_1$ if $n_1 \cdot c < 0$. Otherwise, $T_v N_j = n_2$.

C.2 Converting a great arc on the Gaussian Sphere to a region in the dual space

In this section, we explain how to convert a great arc on the Gaussian sphere into the corresponding region in the dual space.

A point $p = (p_x, p_y, p_z)$ on the Gaussian sphere is transformed into the line \hat{p}' in the dual space (unless $p = (0, 1, 0)$ or $p = (0, -1, 0)$) as follows:

$$p : (p_x, p_y, p_z) \quad \rightarrow \quad \hat{p}' : p_x x - p_z y + p_y = 0 \quad (\text{C.1})$$

This is because the point p on the Gaussian sphere is first projected to the point $\hat{p} = (p_x/p_z, p_y/p_z)$ in the working plane, and then, by the duality transform, \hat{p} is transformed into the line $\hat{p}' : y = (p_x/p_z)x + (p_y/p_z)$, that is, $p_x x - p_z y + p_y = 0$. For the points $(0, 1, 0)$ and $(0, -1, 0)$ on the Gaussian sphere, we define that they are transformed into infinity in the dual space.

Given a great arc a on the Gaussian sphere, since a great arc is a set of points and each of these points is transformed into a line in the dual space, $W(\Pi(a))$ is a region consisting of a set of lines. Call a 's bounding points on the Gaussian sphere $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$. Following the relationship in (C.1), these points are transformed into the lines $\hat{p}' : p_x x - p_z y + p_y = 0$ and $\hat{q}' : q_x x - q_z y + q_y = 0$. Then, $W(\Pi(a))$ becomes the region bounded by \hat{p}' and \hat{q}' as shown in Figure C.1, assuming that the arc length of a is always shorter than π (since we do not have to consider a great arc longer than π in our algorithm, we assume this in the following discussion).

Any point on a is perpendicular to the vector e defined by $e = p \times q$. When the y -component of e , $e_y = p_z q_x - p_x q_z$, is zero, a possibly passes through the point $(0, 1, 0)$ or $(0, -1, 0)$ on the Gaussian sphere that is transformed into infinity in the dual space. Therefore, we consider the case when $e_y \neq 0$ and $e_y = 0$ separately.

C.2.1 Case: $e_y \neq 0$

When $e_y \neq 0$, since $p_z q_x \neq p_x q_z$, \hat{p}' and \hat{q}' are not parallel. Therefore, the dual space is divided into four wedges by \hat{p}' and \hat{q}' . $W(\Pi(a))$ consists of two opposed wedges (called a double wedge [Edelsbrunner et al. 1982]). Notice that a point whose z -coordinate is zero on the Gaussian sphere is transformed into a vertical line in the dual space (see the relationship (C.1)). When p_z and q_z have the same sign, a does not pass through any point where $z = 0$ (recall that a is a part of the great circle and is always shorter than π). Therefore, $W(\Pi(a))$ is a double wedge whose union does not contain a vertical line in its interior (Figure C.1 (a)). On the other hand, if p_z and q_z have different signs, a passes through a point where $z = 0$; therefore, $W(\Pi(a))$ is a double wedge whose union contains a vertical line in its interior (Figure C.1 (b)) in this case. When $p_z = 0$ or $q_z = 0$, we transform an arbitrary point between p and q on a into the line in the dual space; the double wedge containing the line corresponds to $W(\Pi(a))$.

C.2.2 Case: $e_y = 0$

When $e_y = 0$, since $p_z q_x = p_x q_z$, \hat{p}' and \hat{q}' are parallel. $W(\Pi(a))$ consists of a region or regions bounded by the parallel lines \hat{p}' and \hat{q}' . When p_z and q_z have the same sign, a passes through neither the point $(0, 1, 0)$ nor $(0, -1, 0)$. Therefore, $W(\Pi(a))$ should not contain points at infinity; $W(\Pi(a))$ is a region bounded by \hat{p}' and \hat{q}' (Figure C.1 (c)). When p_z and q_z have different signs, a passes through either the point $(0, 1, 0)$ or $(0, -1, 0)$. $W(\Pi(a))$ contains a point transformed into infinity in the dual space; $W(\Pi(a))$ consists of two disjoint regions each of which is bounded by \hat{p}' and \hat{q}' , respectively (Figure C.1 (d)). When $p_z = 0$ (or $q_z = 0$), we transform an arbitrary point between p and q on a into the corresponding line in the dual space; the side of \hat{q}' (\hat{p}' , respectively) containing the line corresponds to $W(\Pi(a))$.

When $p_z = q_z = 0$ (i.e. $e_x = e_y = 0$), we compare the sign of p_x and q_x instead of p_z and q_z since, when the sign of p_x and q_x are different, a passes through either the point $(0, 1, 0)$ or $(0, -1, 0)$. Then, an analogous argument holds for this case (Figure C.1 (e)(f)).

C.3 The characteristic of the gravity direction $\mathbf{d}(\hat{g})$ in the working plane

We prove that, for $r_y > 0$, if the gravity direction g moves clockwise around r along g_r when seen from (r_x, r_y, r_z) , the x-component of $\mathbf{d}(\hat{g})$ is always negative. To prove this, let us consider a point $p = (p_x, p_y, p_z)$ on g_r such that the point $q = (q_x, q_y, q_z)$ that can be obtained by rotating p by an infinitesimal angle clockwise around r has a z-coordinate with the same sign i.e. $p_z q_z > 0$ (note that q is also on g_r). Then, the vector defined by the cross product $(p \times q)$ will be $(-r_x, -r_y, -r_z)$. Since we have assumed that $r_y > 0$, this means that the y-component of $(p \times q)$ must be negative, i.e. $p_z q_x - p_x q_z < 0$. By dividing the each term in the inequality by $p_z q_z$ (recall that $p_z q_z > 0$), we obtain $(q_x/q_z) - (p_x/p_z) < 0$, or $\hat{q}_x - \hat{p}_x < 0$ where \hat{p}_x and \hat{q}_x are the x-component of \hat{p} and \hat{q} , respectively.

Since the vector $\mathbf{d}(\hat{g})$ can be expressed as $\mathbf{d}(\hat{g}) = (\hat{g} - \hat{p})k$ for some $k > 0$, the sign of the x-component of $\mathbf{d}(\hat{g})$ is the same as the sign of $\hat{q}_x - \hat{p}_x$. Since we know that $\hat{q}_x - \hat{p}_x < 0$, the x-component of $\mathbf{d}(\hat{g})$ is negative.

In a similar manner, we can show that, when the gravity direction moves counterclockwise around r along g_r , the x-component of $\mathbf{d}(\hat{g})$ is positive.

C.4 $W^+(\Pi(G))$ and $W^-(\Pi(G))$ in the dual space for great arc G

In this section, given a great arc G on the Gaussian sphere, we explain how to compute the regions $W^+(\Pi(G))$ and $W^-(\Pi(G))$ in the dual space. Recall that the region $W^+(\Pi(G))$ represents the set of lines intersecting $\Pi(G)$ such that the gravity direction \hat{g} moving along $\Pi(G)$ passes through $\Pi(G)$ from the inside to the outside of the corresponding $\Pi(T_v)$ when the

x-component of $\mathbf{d}(\hat{g})$ is positive. The region $W^-(\Pi(G))$ is for the case when the x-component of $\mathbf{d}(\hat{g})$ is negative. Note that $W^+(\Pi(G)) \cup W^-(\Pi(G)) = W(\Pi(G))$ and $W^+(\Pi(G)) \cap W^-(\Pi(G)) = \emptyset$.

Given great arc G , call its bounding points $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$. Figure C.2 shows the possible relationships between $\Pi(G)$ in the working plane and the regions $W^+(\Pi(G))$ and $W^-(\Pi(G))$ in the dual space. In addition, we let $e_i = (e_x, e_y, e_z)$ be the vector in the direction of the incident edge that defines the half-space H_i inducing T_v -arc $A_{T_v j}$, part of which is G .

C.4.1 Case A: $e_y \neq 0$

First, we consider the case when $e_y \neq 0$. When $p_z e_y < 0$ and $q_z e_y < 0$ (Figure C.2 (a)), $\Pi(T_v)$ lies above $\Pi(G)$ in the working plane (refer to Appendix C.5 for the proof). We let $k(s)$ be the slope of the line containing line segment s in the working plane. If the x-component of $\mathbf{d}(\hat{g})$ is negative, only when the slope of $\Pi(g_r)$ is larger than $k(\Pi(G))$, \hat{g} passes through $\Pi(G)$ moving from the inside to the outside of $\Pi(T_v)$ (Figure C.3 (a)). On the other hand, if the x-component of $\mathbf{d}(\hat{g})$ is positive, only when the slope of $\Pi(g_r)$ is smaller than $k(\Pi(G))$, \hat{g} passes through $\Pi(G)$ moving from the inside to the outside of $\Pi(T_v)$ (Figure C.3 (b)).

Points in the dual space whose x-coordinate is smaller than $-k(\Pi(G))$ correspond to lines in the working plane whose slope is larger than $k(\Pi(G))$; points in the dual space whose x-coordinate is larger than $-k(\Pi(G))$ corresponds to lines in the working plane whose slope is smaller than $k(\Pi(G))$. Therefore, the portion of the double wedge $W(\Pi(G))$ lying to the left of $-k(\Pi(G))$ corresponds to $W^-(\Pi(G))$ and the portion of $W(\Pi(G))$ lying to the right of $-k(\Pi(G))$ corresponds to $W^+(\Pi(G))$. Since the x-coordinate of the intersection point of the double wedge $W(\Pi(G))$ is $-k(\Pi(G))$, the disjoint portions of the double wedge correspond to $W^-(\Pi(G))$ and $W^+(\Pi(G))$.

In a similar manner, when $p_z e_y > 0$ and $q_z e_y > 0$ (Figure C.2 (b)), $\Pi(T_v)$ lies below $\Pi(G)$ in the working plane. Then, the portion of $W(\Pi(G))$ lying to the left of $-k(\Pi(G))$ corresponds to $W^+(\Pi(G))$ and the portion of $W(\Pi(G))$ lying to the right of $-k(\Pi(G))$ corresponds to $W^-(\Pi(G))$.

The former two cases hold only when the signs of p_z and q_z are the same (i.e. G does not pass through a point where $z = 0$). Now, we consider the case when the sign of p_z and q_z are different (i.e. G passes through a point where $z = 0$) (Figure C.2 (c)(d)). Letting the point on G whose z-coordinate is zero be $m = (m_x, m_y, 0)$, we define G_1 as the portion of G bounded by p and m and G_2 as the portion of G bounded by q and m (note $G_1 \cup G_2 = G$ and $G_1 \cap G_2 = \emptyset$).

The set of great circles intersecting with G_1 on the Gaussian sphere can be expressed, in the dual space, as a region bounded by $p_x x - p_z y + p_y = 0$ and the vertical line $m_x x + m_y = 0$. When $p_z e_y < 0$, then $\Pi(T_v)$ lies above $\Pi(G_1)$. Therefore, the portion of $W(\Pi(G_1))$ lying to the left of $-k(\Pi(G_1))$ corresponds to $W^-(\Pi(G_1))$ and the portion of $W(\Pi(G_1))$ lying to the right of $-k(\Pi(G_1))$ corresponds to $W^+(\Pi(G_1))$.

In a similar manner, the set of great circles intersecting G_2 on the Gaussian sphere can be expressed, in the dual space, as a region bounded by $q_x x - q_z y + q_y = 0$ and the vertical line $m_x x + m_r y = 0$. Since the signs of p_z and q_z are different, $q_z e_y > 0$ and therefore $\Pi(T_v)$ lies below $\Pi(G_2)$. Thus, the portion of $W(\Pi(G_2))$ lying to the left of $-k(\Pi(G_2))$ corresponds to $W^+(\Pi(G_2))$ and the portion of $W(\Pi(G_2))$ lying to the right of $-k(\Pi(G_2))$ corresponds to $W^-(\Pi(G_2))$.

Since $W^+(\Pi(G)) = W^+(\Pi(G_1)) \cup W^+(\Pi(G_2))$ and $W^-(\Pi(G)) = W^-(\Pi(G_1)) \cup W^-(\Pi(G_2))$, and $k(\Pi(G)) = k(\Pi(G_1)) = k(\Pi(G_2))$, ultimately, each disjoint portion of the double wedge $W(\Pi(G))$ corresponds to $W^+(\Pi(G))$ and $W^-(\Pi(G))$, respectively (Figure C.2 (c)).

For the case when $p_z e_y > 0$ and $q_z e_y < 0$, an analogous argument holds (Figure C.2 (d)).

C.4.2 Case B: $e_y = 0$

Now, we consider the other case, $e_y = 0$, which we further divide into subcases $e_x \neq 0$ and $e_x = 0$.

C.4.2.1 Sub-Case 1: $e_y = 0$ and $e_x \neq 0$

When $e_y = 0$ and $e_x \neq 0$, if $p_z e_x < 0$ and $q_z e_x < 0$, then $\Pi(T_v)$ lies to the right of $\Pi(G)$ in the working plane (refer to Appendix C.5). Then, only when the x-component of $\mathbf{d}(\hat{g})$ is negative, \hat{g} passes through $\Pi(G)$ from the inside to the outside of $\Pi(T_v)$. Therefore, $W^-(\Pi(G)) = W(\Pi(G))$ and $W^+(\Pi(G)) = \emptyset$ (Figure C.2 (e)). On the other hand, when $p_z e_x > 0$ and $q_z e_x > 0$, $\Pi(T_v)$ lies to the left of $\Pi(G)$ in the working plane. Thus, only when the x-component of $\mathbf{d}(\hat{g})$ is positive, \hat{g} passes through $\Pi(G)$ from the inside to the outside of $\Pi(T_v)$. Therefore, $W^+(\Pi(G)) = W(\Pi(G))$ and $W^-(\Pi(G)) = \emptyset$ (Figure C.2 (f)).

The former two cases discussed in the previous paragraph hold only when the signs of p_z and q_z are the same. Now, we consider the case when the signs of p_z and q_z are different. Letting the point on G whose z-coordinate is zero be m , we define G_1 as the part of G bounded by p and m and G_2 as the part of G bounded by q and m . (Notice that when $e_y = 0$, $m = (0, 1, 0)$ or $(0, -1, 0)$.)

When $p_z e_x < 0$, $\Pi(T_v)$ lies to the right of $\Pi(G_1)$ in the working plane. Therefore, $W^-(\Pi(G_1)) = W(\Pi(G_1))$ and $W^+(\Pi(G_1)) = \emptyset$. Since the signs of p_z and q_z are different, $q_z e_x > 0$ and therefore $\Pi(T_v)$ lies to the left of $\Pi(G_2)$ in the working plane. Therefore, $W^+(\Pi(G_2)) = W(\Pi(G_2))$ and $W^-(\Pi(G_2)) = \emptyset$.

Since $W^+(\Pi(G)) = W^+(\Pi(G_1)) \cup W^+(\Pi(G_2))$, $W^+(\Pi(G)) = W(\Pi(G_2))$. Since $W^-(\Pi(G)) = W^-(\Pi(G_1)) \cup W^-(\Pi(G_2))$, $W^-(\Pi(G)) = W(\Pi(G_1))$ (Figure C.2 (g)). For the case when $p_z e_x > 0$ and $q_z e_x < 0$, an analogous argument holds (Figure C.2 (h)).

C.4.2.2 Sub-Case 2: $e_y = 0$ and $e_x = 0$

When $e_y = 0$ and $e_x = 0$, $p_z = q_z = 0$. To take advantage of the observation in the previous subsection, we consider the problem by rotating $p = (p_x, p_y, 0)$, $q = (q_x, q_y, 0)$, and, $e = (0, 0, e_z)$ by an infinitesimal angle ϵ counterclockwise around the y-axis, obtaining p^* , q^* , and e^* , respectively (where $p_z^* = -p_x \sin(\epsilon)$, $q_z^* = -q_x \sin(\epsilon)$, and $e_x^* = e_z \sin(\epsilon)$). Then, by checking the signs of $p_z^* e_x^*$ and $q_z^* e_x^*$, we can decide which portion of $W(\Pi(G))$ corresponds to $W^+(\Pi(G))$ and which to $W^-(\Pi(G))$.

Checking the sign of $p_z^* e_x^*$ (respectively, $q_z^* e_x^*$) is equivalent to checking the sign of $-p_x e_z$ (respectively, $-q_x e_z$). If $p_x e_z > 0$ and $q_x e_z > 0$, $W^-(\Pi(G)) = W(\Pi(G))$ and $W^+(\Pi(G)) = \emptyset$. On the other hand, when $p_x e_z < 0$ and $q_x e_z < 0$, $W^+(\Pi(G)) = W(\Pi(G))$ and $W^-(\Pi(G)) = \emptyset$.

The former two cases discussed in the previous paragraph hold only when the signs of p_x and q_x are the same. For the case when the signs of p_x and q_x are different, letting the point on G whose x-coordinate is zero be m , we define G_1 as the portion of G bounded by p and m and G_2 as the portion of G bounded by q and m . (Notice again that when $e_y = 0$, $m = (0, 1, 0)$ or $(0, -1, 0)$.) Then, when $p_x e_z > 0$ and $q_x e_z < 0$, $W^-(\Pi(G_1)) = W(\Pi(G_1))$ and $W^+(\Pi(G_1)) = \emptyset$ and $W^+(\Pi(G_2)) = W(\Pi(G_2))$ and $W^-(\Pi(G_2)) = \emptyset$. Finally, we can write $W^+(\Pi(G)) = W(\Pi(G_2))$ and $W^-(\Pi(G)) = W(\Pi(G_1))$. For the case when $p_x e_z > 0$ and $q_x e_z < 0$, an analogous argument holds as well.

C.5 Determining the side of a T_v -arc where T_v lies in the working plane

In this section, we explain, for each T_v -arc $A_{T_v,j}$, how to determine on which side of $\Pi(A_{T_v,j})$ the corresponding $\Pi(T_v)$ lies in the working plane. We let $e_i = (e_x, e_y, e_z)$ be the vector in the direction of the incident edge that defines the half-space H_i inducing $A_{T_v,j}$ on the Gaussian sphere. Given a T_v -arc, the side of $\Pi(A_{T_v,j})$ the corresponding $\Pi(T_v)$ lies changes when T_v straddles a point where $z = 0$. Thus, we consider the portion of the T_v where $z > 0$ and the portion of the T_v where $z < 0$, separately. Suppose we are given a (portion of) T_v completely lying on one hemisphere ($z > 0$ or $z < 0$) of the Gaussian sphere. we let $t = (t_x, t_y, t_z)$ be an arbitrary point inside of the portion of T_v (not on the boundary of T_v) i.e. $t_z > 0$ or $t_z < 0$ (from the definition of a concave vertex, T_v always has a non-zero area and thus an interior point t where $t_z \neq 0$ always exists). Since $H_i = \{p \mid e_i \cdot p \leq 0, \|p\| = 1\}$ and $T_v = \bigcap_i H_i$, it follows that $t \cdot e_i < 0$. Using the Gnomonic projection, t is projected to the point $\hat{t} = (t_x/t_z, t_y/t_z)$ in the working plane.

We first consider the case where $e_y \neq 0$. Using the Gnomonic projection, $\Pi(A_{T_v,j})$ is a portion of the line $e_x x + e_y y + e_z = 0$ in the working plane. Since $\hat{t} \in \Pi(T_v)$, $\Pi(T_v)$ lies on the same side of the line as \hat{t} does in the working plane. A 2D vector perpendicular to the line in the working plane can be represented as $\hat{n} = (e_x/e_y, 1)$. Since $e_y \neq 0$, the line always passes through the point $\hat{a} = (0, -e_z/e_y)$. Now, we consider the following 2D dot product in the

working plane (denoted by the symbol \circ): $\hat{n} \circ (\hat{t} - \hat{a}) = (e_x/e_y)(t_x/t_z - 0) + (1)(t_y/t_z + e_z/e_y) = (t_x e_x + t_y e_y + t_z e_z)/(e_y t_z) = (t \cdot e_i)/(e_y t_z)$. Since $t \cdot e_i < 0$, $\hat{n} \circ (\hat{t} - \hat{a}) > 0$ if $e_y t_z < 0$ and $\hat{n} \circ (\hat{t} - \hat{a}) < 0$ if $e_y t_z > 0$. Recalling that $\hat{n}_y = 1$, $\Pi(T_v)$ lies above $\Pi(A_{T_v,j})$ for the portion of $\Pi(T_v)$ where $e_y t_z < 0$ and, $\Pi(T_v)$ lies below $\Pi(A_{T_v,j})$ for the portion of $\Pi(T_v)$ where $e_y t_z > 0$. Note that the sign of t_z indicates on which hemisphere ($z > 0$ or $z < 0$) of the Gaussian sphere the corresponding T_v lies.

For the case where $e_y = 0$, $\Pi(A_{T_v,j})$ is a portion of the line $e_x x + e_z = 0$ in the working plane. A 2D vector perpendicular to this line can be represented as $\hat{n} = (1, 0)$. From the line equation, it always passes through the point $\hat{b} = (-e_z/e_x, 0)$. Then, we consider the 2D dot product in the working plane: $\hat{n} \circ (\hat{t} - \hat{b}) = t_x/t_z - (-e_z/e_x) = (t_x e_x + t_z e_z)/(e_x t_z) = (t \cdot e_i)/(e_x t_z)$ (recall that $e_y = 0$). Since $t \cdot e_i < 0$, $\hat{n} \circ (\hat{t} - \hat{b}) > 0$ if $e_x t_z < 0$ and $\hat{n} \circ (\hat{t} - \hat{b}) < 0$ if $e_x t_z > 0$. Since $\hat{n}_x = 1$, $\Pi(T_v)$ lies to the right of $\Pi(A_{T_v,j})$ for the portion of $\Pi(T_v)$ where $e_x t_z < 0$ and $\Pi(T_v)$ lies to the left of $\Pi(A_{T_v,j})$ for the portion of $\Pi(A_{T_v,j})$ where $e_x t_z > 0$.

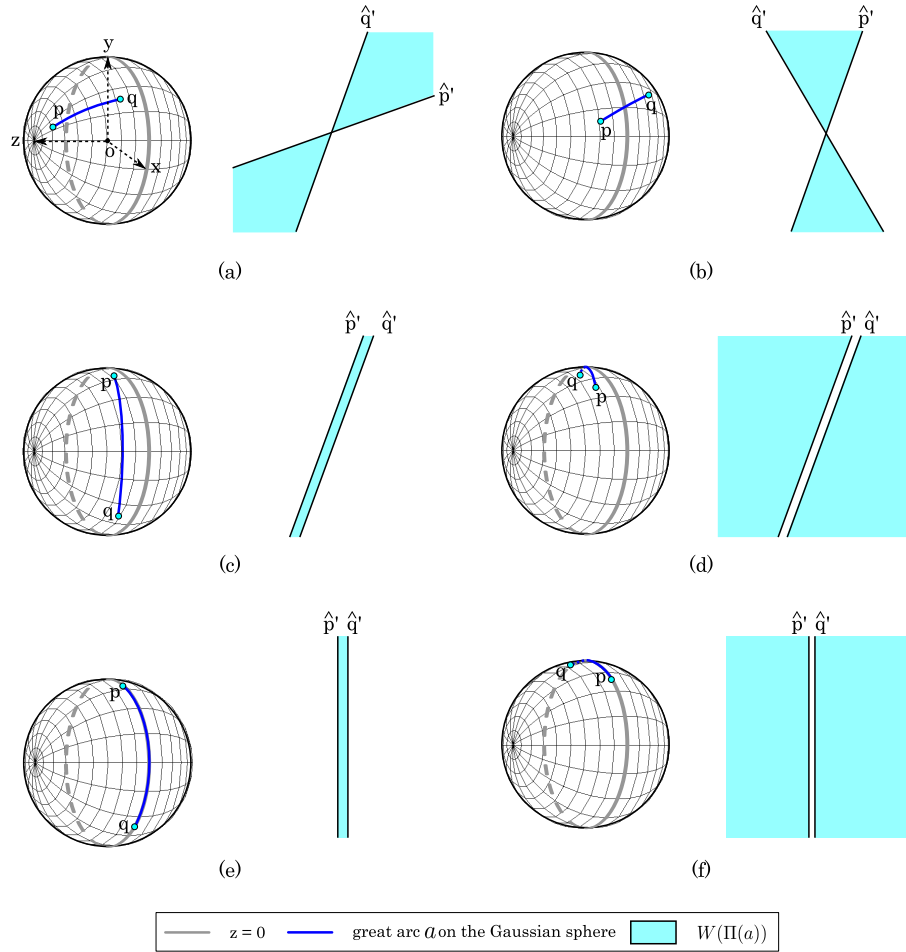


Figure C.1: Given a great arc a , letting its bounding points on the Gaussian sphere be $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$, each point is transformed into the line $\hat{p}' : p_x x - p_z y + p_y = 0$ and $\hat{q}' : q_x x - q_z y + q_y = 0$ in the dual space. Then, the set of great arcs intersecting with a , $W(\Pi(a))$, is expressed as the region bounded by \hat{p}' and \hat{q}' in the dual space. When $e_y \neq 0$, $W(\Pi(a))$ consists of two opposed wedges (called a double wedge [Edelsbrunner et al. 1982]). For such double wedges, when p_z and q_z have the same sign, $W(\Pi(a))$ is a double wedge whose union does not contain a vertical line in its interior (shown in (a)). On the other hand, when p_z and q_z have different signs, $W(\Pi(a))$ is a double wedge whose union contains a vertical line in its interior (shown in (b)). When $e_y = 0$, $W(\Pi(a))$ consists of a region or regions bounded by the parallel lines \hat{p}' and \hat{q}' . When p_z and q_z have the same sign, $W(\Pi(a))$ is a region bounded by \hat{p}' and \hat{q}' (shown in (c)). When p_z and q_z have different signs, $W(\Pi(a))$ consists of two disjoint regions each of which is bounded by \hat{p}' and \hat{q}' , respectively (shown in (d)). When $p_z = q_z = 0$ (i.e. $e_x = e_y = 0$), \hat{p}' and \hat{q}' become vertical lines. In this case, we compare the sign of p_x and q_x instead of p_z and q_z . If the signs match, we have one connected region (shown in (e)); if not, we have two disjoint regions (shown in (f)).

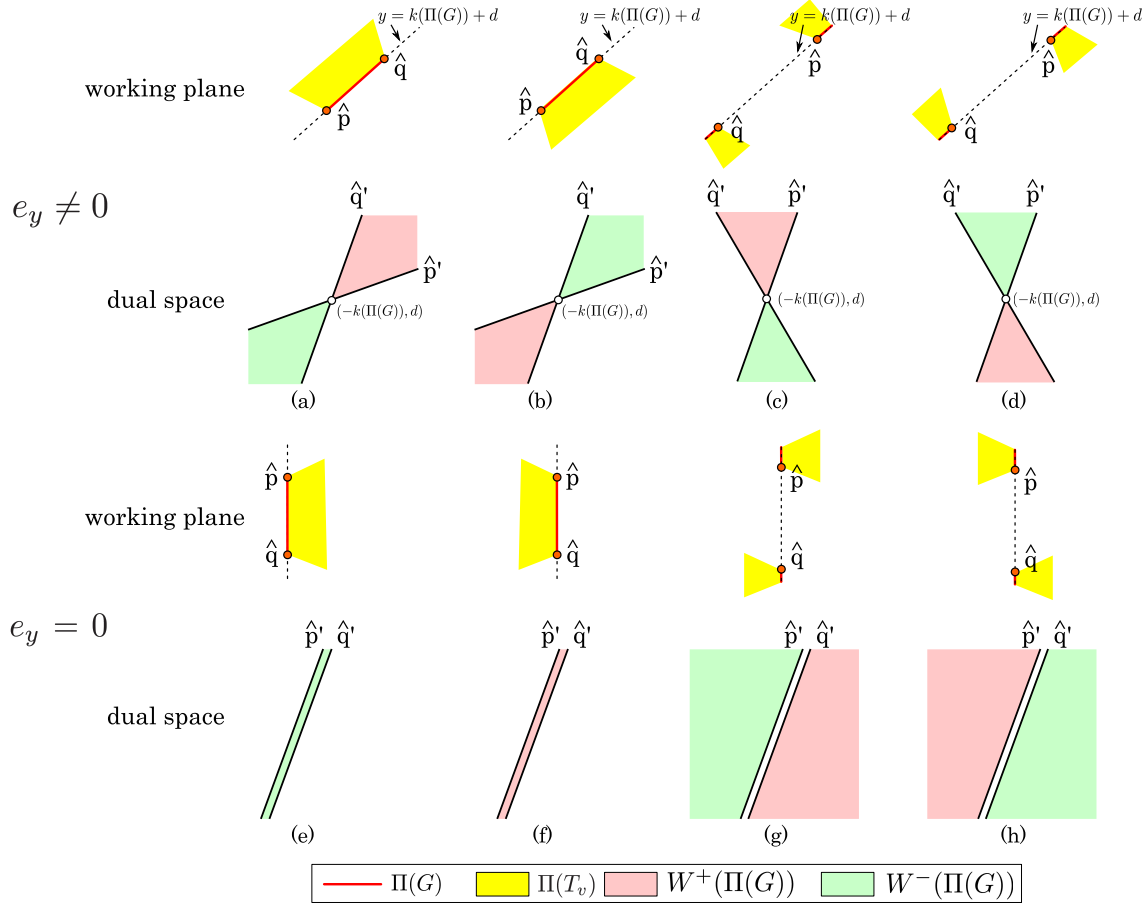


Figure C.2: Possible relationships between $\Pi(G)$ and $\Pi(T_v)$ in the working plane and $W^+(\Pi(G))$ and $W^-(\Pi(G))$ in the dual space. Letting $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$ be the points bounding G on the Gaussian sphere, and $e = (e_x, e_y, e_z)$ be the vector in the direction of the incident edge that defines the half-space H_i inducing T_v -arc $A_{T_v,j}$, portion of which is G , then figures (a)-(d) show the cases when $e_y \neq 0$ and either (a) $p_z e_y < 0$ and $q_z e_y < 0$, (b) $p_z e_y > 0$ and $q_z e_y > 0$, (c) $p_z e_y < 0$ and $q_z e_y > 0$, or (d) $p_z e_y > 0$ and $q_z e_y < 0$. Figures (e)-(f) shows the cases when $e_y = 0$, $e_x \neq 0$ and either (e) $p_z e_x < 0$ and $q_z e_x < 0$, (f) $p_z e_x > 0$ and $q_z e_x > 0$, (g) $p_z e_x < 0$ and $q_z e_x > 0$, or (h) $p_z e_x > 0$ and $q_z e_x < 0$. Recall that $\hat{p} = (p_x/p_z, p_y/p_z)$ and $\hat{q} = (q_x/q_z, q_y/q_z)$.

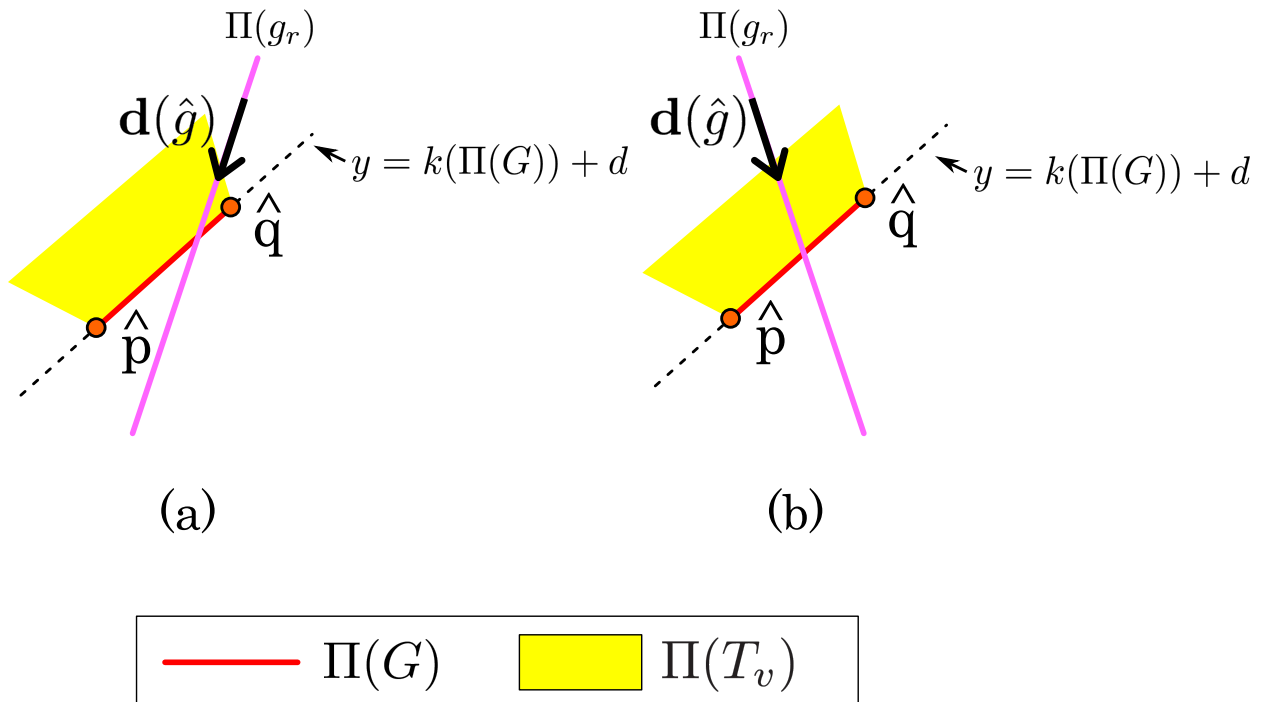


Figure C.3: When $\Pi(T_v)$ lies above $\Pi(G)$ in the working plane, (a) if the x-component of $\mathbf{d}(\hat{g})$ is negative, only when the slope of $\Pi(g_r)$ is larger than $k(\Pi(G))$, \hat{g} passes through $\Pi(G)$ moving from the inside to the outside of $\Pi(T_v)$. (b) If the x-component of $\mathbf{d}(\hat{g})$ is positive, only when the slope of $\Pi(g_r)$ is smaller than $k(\Pi(G))$, \hat{g} passes through $\Pi(G)$ moving from the inside to the outside of $\Pi(T_v)$.

Bibliography

- Aloupis, Greg, Jean Cardinal, Sebastien Collette, Ferran Hurtado, Stefan Langerman, and Joseph O'Rourke (2008). "Draining a Polygon – or – Rolling a Ball out of a Polygon." In: *CCCG*. URL: <http://dblp.uni-trier.de/db/conf/cccg/cccg2008.html>.
- Arbelaez, D., M. Avila, A. Krishnamurthy, W. Li, Y. Yasui, D. Dornfeld, and S. McMains (2008). "Cleanability of mechanical components." In: *Proceedings of 2008 NSF Engineering Research and Innovation Conference*. Knoxville, Tennessee.
- Avila, M., C. Reich-Weiser, D. Dornfeld, and S. McMains (2006). "Design and manufacturing for cleanability in high performance cutting." In: *Proceeding of 2nd International High Performance Cutting Conference*. CIRP, Vancouver, BC.
- Avila, Miguel C., Joel D. Gardner, Corinne Reich-Weiser, Athulan Vijayaraghavan, and David Dornfeld (2005). "Strategies for Burr Minimization and Cleanability in Aerospace and Automotive Manufacturing." In: *SAE J. Aerospace* 114 (1), pp. 1073–1082.
- Bentley, Jon Louis (1975). "Multidimensional binary search trees used for associative searching." In: *Commun. ACM* 18 (9), pp. 509–517. ISSN: 0001-0782. DOI: <http://doi.acm.org/10.1145/361002.361007>.
- Berger, K. (2006). "Burs, chips and cleanness of parts – activities and aims in the German automotive industry." In: *Presentation at CIRP Working Group on Burr Formation*. Paris.
- Bose, Prosenjit and Godfried Toussaint (1995). "Geometric and computational aspects of gravity casting." In: *Computer-Aided Design* 27 (6), pp. 455–464.
- Bose, Prosenjit, Marc J. van Kreveld, and Godfried T. Toussaint (1993). "Filling Polyhedral Molds." In: *Workshop on Algorithms and Data Structures*, pp. 210–221.
- Brown, Kevin Quentin (1979). "Geometric transforms for fast geometric algorithms." AAI8012772. PhD thesis. Pittsburgh, PA, USA.
- Chazelle, Bernard (1982). "A theorem on polygon cutting with applications." In: *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, pp. 339–349. DOI: <http://dx.doi.org/10.1109/SFCS.1982.58>.
- Chen, Lin-Lin and T. C. Woo (1992). "Computational Geometry on the Sphere With Application to Automated Machining." In: *Journal of Mechanical Design* 114 (2), pp. 288–295. DOI: 10.1115/1.2916945. URL: <http://link.aip.org/link/?JMD/114/288/1>.

- Chen, Lin-Lin, Shuo-Yan Chou, and Tony C. Woo (1993). "Separating and intersecting spherical polygons: computing machinability on three-, four-, and five-axis numerically controlled machines." In: *ACM Trans. Graph.* 12 (4), pp. 305–326. ISSN: 0730-0301. DOI: <http://doi.acm.org/10.1145/159730.159732>. URL: <http://doi.acm.org/10.1145/159730.159732>.
- de Berg, Mark, Otfried Cheong, Marc van Kreveld, and Mark Overmars (2008). *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS. ISBN: 3540779736, 9783540779735.
- Edelsbrunner, H., H. A. Maurer, F. P. Preparata, A. L. Rosenberg, E. Welzl, and D. Wood (1982). "Stabbing line segments." In: *BIT Numerical Mathematics* 22 (3), pp. 274–281.
- Gupta, Prosenjit, Ravi Janardan, Jayanth Majhi, and Tony Woo (1996). "Efficient geometric algorithms for workpiece orientation in 4- and 5-axis NC machining." In: *Computer-Aided Design* 28 (8), pp. 577–587. ISSN: 0010-4485. DOI: 10.1016/0010-4485(95)00071-2. URL: <http://www.sciencedirect.com/science/article/pii/0010448595000712>.
- Harada, Takahiro, Seiichi Koshizuka, and Yoichiro Kawaguchi (2007). "Smoothed Particle Hydrodynamics on GPUs." In: *Proc. of Computer Graphics International*, pp. 63–70.
- Hershberger, John and Jack Snoeyink (1994). "Computing minimum length paths of a given homotopy class." In: *Comput. Geom. Theory Appl.* 4 (2), pp. 63–97. ISSN: 0925-7721. DOI: [http://dx.doi.org/10.1016/0925-7721\(94\)90010-8](http://dx.doi.org/10.1016/0925-7721(94)90010-8).
- Janardan, Ravi and Tony C. Woo (2004). "MANUFACTURING PROCESSES." In: *Handbook of Discrete and Computational Geometry*. Ed. by Jacob E. Goodman and Joseph O'Rourke. Boca Raton, FL: CRC Press LLC. Chap. 55, pp. 1241–1256.
- Kallmann, Marcelo (2005). "Path Planning in Triangulations." In: *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*. Edinburgh, Scotland.
- Kapoor, Sanjiv and H. Ramesh (2000). "An Algorithm for Enumerating All Spanning Trees of a Directed Graph." In: *Algorithmica* 27, pp. 120–130.
- Lee, D.-T. and F. P. Preparata (1984). "Euclidean Shortest Paths in the Presence of Rectilinear Barriers." In: *Networks* 14 (3), pp. 393–410.
- Lorensen, William E. and Harvey E. Cline (1987). "Marching cubes: A high resolution 3D surface construction algorithm." In: *SIGGRAPH Comput. Graph.* 21 (4), pp. 163–169. ISSN: 0097-8930. DOI: <http://doi.acm.org/10.1145/37402.37422>. URL: <http://doi.acm.org/10.1145/37402.37422>.
- McMains, Sara (2000). "Slicing." In: *Geometric Algorithms and Data Representation for Solid Freeform Fabrication*. PhD thesis. Chap. 7, pp. 102–114.
- McMains, Sara and Carlo Séquin (1999). "A Coherent Sweep Plane Slicer for Layered Manufacturing." In: *Proc. 5th ACM Symposium on Solid Modeling and Applications*, pp. 285–295.
- Muller, D. E. and F. P. Preparata (1978). "Finding the intersection of two convex polyhedra." In: *Theoretical Computer Science* 7 (2), pp. 217–236. ISSN: 0304-3975. DOI: DOI:10.1016/0304-3975(78)90051-8. URL: <http://www.sciencedirect.com/science/article/B6V1G-45FKD0W-3M/2/2da173f9db0158f23a6b2ab209dd55fc>.

- Müller, Matthias, Jos Stam, Doug James, and Nils Thürey (2008). “Real time physics: class notes.” In: *ACM SIGGRAPH 2008 classes*. Los Angeles, California: ACM, pp. 1–90. DOI: <http://doi.acm.org/10.1145/1401132.1401245>.
- Müller-Fischer, Matthias, David Charypar, and Markus Gross (2003). “Particle-based fluid simulation for interactive applications.” In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. San Diego, California: Eurographics Association, pp. 154–159.
- Müller-Fischer, Matthias, Pauly Mark, Gross Markus, Keiser Richard, and Wicke Martin (2007). “Physics-Based Animation.” In: *Point-Based Graphics*. Ed. by Markus Gross and Hanspeter Pfister. Burlington: Morgan Kaufmann, pp. 340–387.
- Nguyen, Hubert (2007). *GPU Gems 3*. Addison-Wesley Professional. ISBN: 9780321545428.
- Pascucci, Valerio, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas (2007). “Robust on-line computation of Reeb graphs: simplicity and speed.” In: *ACM Trans. Graph.* 26 (3), p. 58.
- Preparata, Franco P. and Michael I. Shamos (1985). *Computational geometry: an introduction*. New York, NY, USA: Springer-Verlag New York, Inc. ISBN: 0-387-96131-3.
- Reeb, G. (1946). “Sur les points singuliers d’une forme de Pfaff complètement intégrable ou d’une fonction numérique [On the singular points of a complete integral pfaff form or of a numerical function].” In: *Comptes Rendus Acad. Science Paris* 222, pp. 847–849.
- Samet, Hanan (2005). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123694469.
- Tang, K., T. Woo, and J. Gan (1992). “Maximum Intersection of Spherical Polygons and Workpiece Orientation for 4- and 5-Axis Machining.” In: *Journal of Mechanical Design* 114 (3), pp. 477–485. DOI: 10.1115/1.2926576. URL: <http://link.aip.org/link/?JMD/114/477/1>.
- Tang, Kai, Lin-Lin Chen, and Shuo-Yan Chou (1998). “Optimal workpiece setups for 4-axis numerical control machining based on machinability.” In: *Computers in Industry* 37 (1), pp. 27–41.
- Tierny, Julien, Attila Gyulassy, Eddie Simon, and Valerio Pascucci (2009). “Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees.” In: *IEEE Transactions on Visualization and Computer Graphics* 15 (6), pp. 1177–1184. ISSN: 1077-2626. DOI: <http://dx.doi.org/10.1109/TVCG.2009.163>. URL: <http://dx.doi.org/10.1109/TVCG.2009.163>.
- Woo, T. C. (1994). “Visibility maps and spherical algorithms.” In: *Computer-Aided Design* 26 (1), pp. 6–16.