Pool Segmentation for Predicting Water Traps

Yusuke Yasui
Mechanical Engineering
UC Berkeley

Sara McMains[*]
Mechanical Engineering
UC Berkeley

Thomas Glau
Research & Development
Daimler AG
Stuttgart, Germany

ABSTRACT

We propose a new method to detect potential water trap regions in voids of oriented polygonal models that approximate the geometry of mechanical parts. Since water traps decrease water jet cleaning efficiency, 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.

Categories and Subject Descriptors

J.6 [**Computer Applications**]: COMPUTER-AIDED ENGINEERING—*Computer-aided design (CAD)*; J.6 [**Computer Applications**]: COMPUTER-AIDED ENGINEERING—*Computer-aided manufacturing (CAM)*

Keywords

pool segmentation, plane sweep algorithm, water trap, 2D slice polygon, directed graph

1. INTRODUCTION

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 (such as detached burrs and chips from machining) has increased considerably. Reliably removing solid particle contaminants from the surfaces of mechanical parts has become increasingly important in the automotive industry. However, miniaturization and increased geometric complexity has made it more difficult to access all the surfaces of parts to remove contaminants.

[*]mcmains@me.berkeley.edu

In this paper, we consider cleaning with high-pressure water jets. Water jets are effective for removing contaminants from the surface of mechanical parts, but the water may become trapped inside the part if the geometry of voids is complex. Since contaminants may accumulate in such regions and trapped water must be drained after the cleaning, finding an orientation that minimizes the potential water trap regions is important to increase the cleaning efficiency and reduce the draining time and effort after cleaning.

We propose a new method to pre-identify the regions of cleaning-incompatible water traps in voids of mechanical parts using a geometric volume segmentation method, given the part orientation. We assume that the part geometry is given as a 2-manifold triangulated polygonal mesh and that the force applied to the water is only the gravitational force. In order to provide interactive "Design for Cleanability" (DFC) feedback to designers, our algorithm does not rely on computationally expensive methods such as computational fluid dynamics (CFD).

1.1 Previous Work

To increase the efficiency of cleaning processes, analytical tools that predict cleaning effectiveness at the design and process planning stages are needed. Initial research has focused on understanding the effect of key cleaning process parameters [1, 2, 3, 4].

Generally speaking, manufacturing processes are highly complex phenomena, especially when fluid is involved. Since simulating such phenomena using CFD simulation is time-consuming, some purely geometric approaches have been proposed. 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 [5, 6]. Their algorithm finds the orientation that minimizes the number of venting holes that need to be added to allow air to escape to insure a complete fill. Yasui and McMains proposed an algorithm to test whether a given rotation axis can fully drain a workpiece when the workpiece is rotated around the axis [13, 14]. The method we propose in this paper is also purely geometric and does not involve computationally costly simulations.

2. 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 due to water traps. We assume throughout this paper that the part geometry has been rotated to the desired test orientation, so that gravity

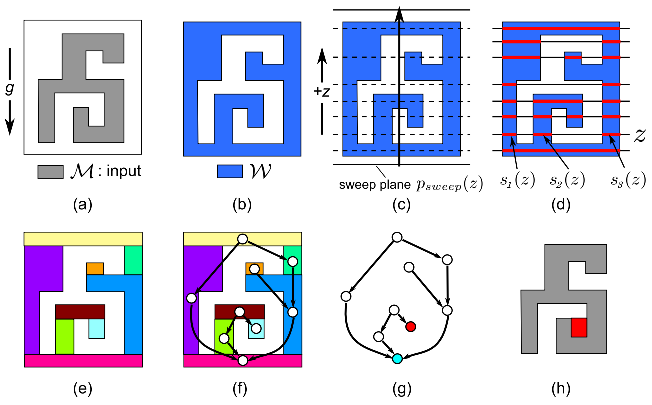


Figure 1: Overview: (a) From input geometry $\mathcal{M}$, (b) we define the space $\mathcal{W} = \mathcal{B} \setminus \mathcal{M}$ where water could flow. (c) Using a sweep plane $p_{sweep}(z)$, (d) we track the evolution of connected slice components $s_i(z) \in \mathcal{W} \cap p_{sweep}(z)$. (e) 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 constructed graph, we locate potential water trap regions. (h) We map the region(s) to input $\mathcal{M}$.

always acts vertically (i.e. down the z-axis).

Figure 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.

The Reeb graph [11] 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. [10] or Tierny et al. [12] 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.

2.1 Preliminaries

We introduce some notation that we will use to explain how we split $\mathcal{W}$ into pools and add the directed edges between nodes corresponding to pools. We consider a sweep plane $p_{sweep}(\mathrm{z} = z)$ perpendicular to the z-axis (i.e. the gravity direction) intersecting it at z. Given a sweep plane $p_{sweep}(z)$, we define the *slice* at z, $S(z)$, as the intersection of $\mathcal{W}$ and $p_{sweep}(z)$: $S(z) = \mathcal{W} \cap p_{sweep}(z)$. As shown in Figure 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 $s_i(z)$, $S_{below}(s_i(z))$, 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 $s_i(z)$, $S_{above}(s_i(z))$, 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 $S_{below}(s_i(z))$ and $S_{above}(s_i(z))$, 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 $|S_{below}(s_i(z))| = 0$, we call $s_i(z)$ a *beginning slice polygon* since a new slice polygon appears as the sweep plane moves from $p_{sweep}(z^-)$ to $p_{sweep}(z^+)$. On the other hand, if $|S_{above}(s_i(z))| = 0$, we call $s_i(z)$ an *ending slice polygon*, since an existing slice polygon disappears as the sweep plane moves from $p_{sweep}(z^-)$ to $p_{sweep}(z^+)$. If $|S_{below}(s_i(z))| \geq 2$ and $|S_{above}(s_i(z))| \geq 1$ or $|S_{below}(s_i(z))| \geq 1$ and $|S_{above}(s_i(z))| \geq 2$, 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 $p_{sweep}(z^-)$ to $p_{sweep}(z^+)$. Finally, if $|S_{below}(s_i(z))| = |S_{above}(s_i(z))| = 1$, 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 $p_{sweep}(z^-)$ to $p_{sweep}(z^+)$.

2.2 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 $p_{sweep}(z)$ from $\mathrm{z} = -\infty$ to $\mathrm{z} = +\infty$. If $\mathcal{W} \cap p_{sweep}(z)$ yields a beginning slice polygon, we generate a new pool bounded from below by the beginning slice polygon. If $\mathcal{W} \cap p_{sweep}(z)$ yields a no-change slice polygon, the no-change slice polygon $s_i(z)$ defines the pool $pool(s_j(z^-))$ where $s_j(z^-) \in S_{below}(s_i(z))$. If $\mathcal{W} \cap p_{sweep}(z)$ yields an ending slice polygon, we complete the corresponding existing pool, bounding it from above with the ending slice polygon. Finally, if $\mathcal{W} \cap p_{sweep}(z)$ 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 1 (f)).

2.3 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 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. 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 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 [7, 8]. 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 $\mathcal{W} \cap p_{sweep}(z)$ yields any of beginning, ending, and/or merge/split slice polygons, $p_{sweep}(z)$ always intersects with one of the vertices in $\mathcal{W}$.

3.1 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 2). Each boundary cycle consists of a set of triangles. Any 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. Since we process each vertex in ascending order of 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 $p_{sweep}(z)$. Each line segment constituting the closed polygonal chain is defined by the intersection between $p_{sweep}(z)$ 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 $p_{sweep}(z)$ does not intersect any vertices in $\mathcal{W}$) as shown in Figure 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 $p_{sweep}(z)$ intersects with 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 $p_{sweep}(z)$ 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 in section 2.1. 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.

3.1.1 Boundary Cycle Management

Boundary cycles are generated, completed, or updated when we process each vertex in $\mathcal{W}$. As McMains et al. 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 and B 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 $\mathcal{W} \cap p_{sweep}(z)$ yields a beginning, an ending, or a merge/split slice polygon, $p_{sweep}(z)$ always intersects with a beginning, an 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 2 (b)); when an existing boundary cycle is completed at an ending vertex an existing slice polygon boundary disappears (Figure 2 (l)); and, when multiple boundary cycles merge into one boundary cycle or one boundary cycle splits into mul-

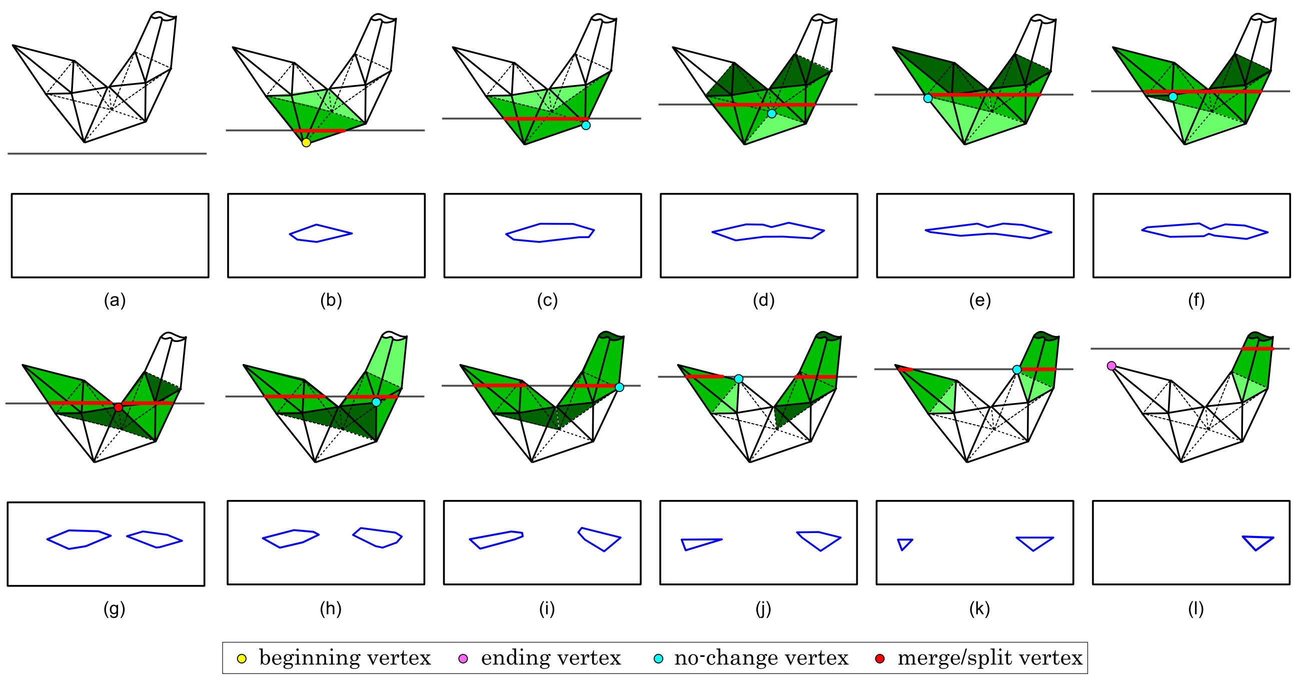


Figure 2: Figures (a)-(l) illustrate how the triangles in boundary cycles are updated as the sweep plane moves from bottom to top over a portion of $\mathcal{W}$. The top drawing in each subfigure shows the set of triangles (colored) currently in boundary cycles just after the sweep plane processes the indicated vertex; the bottom drawing in each subfigure shows the corresponding slice polygon boundaries, with the sweep plane shown as a rectangle.

tiple 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 2 (g)).

We update triangles in an existing boundary cycle at a no-change vertex (Figure 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.

3.1.2 *Boundary Cycle Classification*

As shown in Figure 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 $s_i(z)$, we 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, $\partial s_i(z) = \{(\partial s_i(z))_1, \cdots, (\partial s_i(z))_{|\partial s_i(z)|}\}$.

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 $p_{sweep}(z)$ 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).

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 Pool Segmentation

Now, we describe how to implement pool segmentation. In section 2.2, 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.

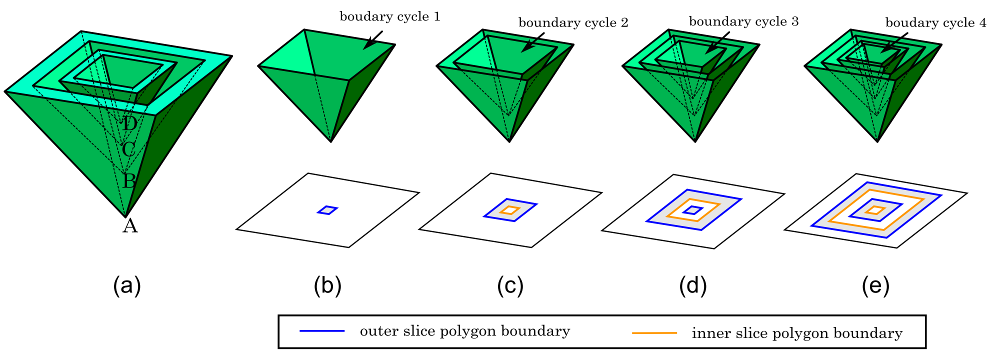


Figure 3: Each of subfigures (b)-(e) shows the boundary cycles and the corresponding slice polygon boundaries on the sweep plane just after processing vertices A, B, C, and, D shown in (a), respectively. Boundary cycles 1 and 3 are outer boundary cycles; Boundary cycles 2 and 4 are inner boundary cycles. 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.

Algorithm 1 InitializePool(b_{outer}, z)

```
Input: $b_{outer}$: an outer boundary cycle, $z$: z-coordinate
Output: $p$: pool generated at $z$
$b_{outer}$-> pool $\leftarrow p$
$p$-> $T_1 \leftarrow \emptyset$
$(\partial s)_1 \leftarrow \emptyset$
for each triangle $t \in b_{outer}$ do
    Compute $t_{cut}$, the portion of $t$ higher than $z$
    $p$-> $T_1 \leftarrow$ ($p$-> $T_1) \cup t_{cut}$
    $(\partial s)_1 \leftarrow (\partial s)_1 \cup (t \cap p_{sweep}(z^+))$
end for
$j \leftarrow 2$
for each $b_{inner} \in inner(b_{outer})$ do
    $b_{inner}$-> pool $\leftarrow p$
    $p$-> $T_j \leftarrow \emptyset$
    $(\partial s)_j \leftarrow \emptyset$
    for each triangle $t \in b_{inner}$ do
        Compute $t_{cut}$, the portion of $t$ higher than $z$
        $p$-> $T_j \leftarrow$ ($p$-> $T_j) \cup t_{cut}$
        $(\partial s)_j \leftarrow (\partial s)_j \cup (t \cap p_{sweep}(z^+))$
    end for
    $j \leftarrow (j+1)$
end for
$p$->BottomFace $\leftarrow \bigcup_{k=1}^{j-1} (\partial s)_k$
return $p$
```

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. 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.

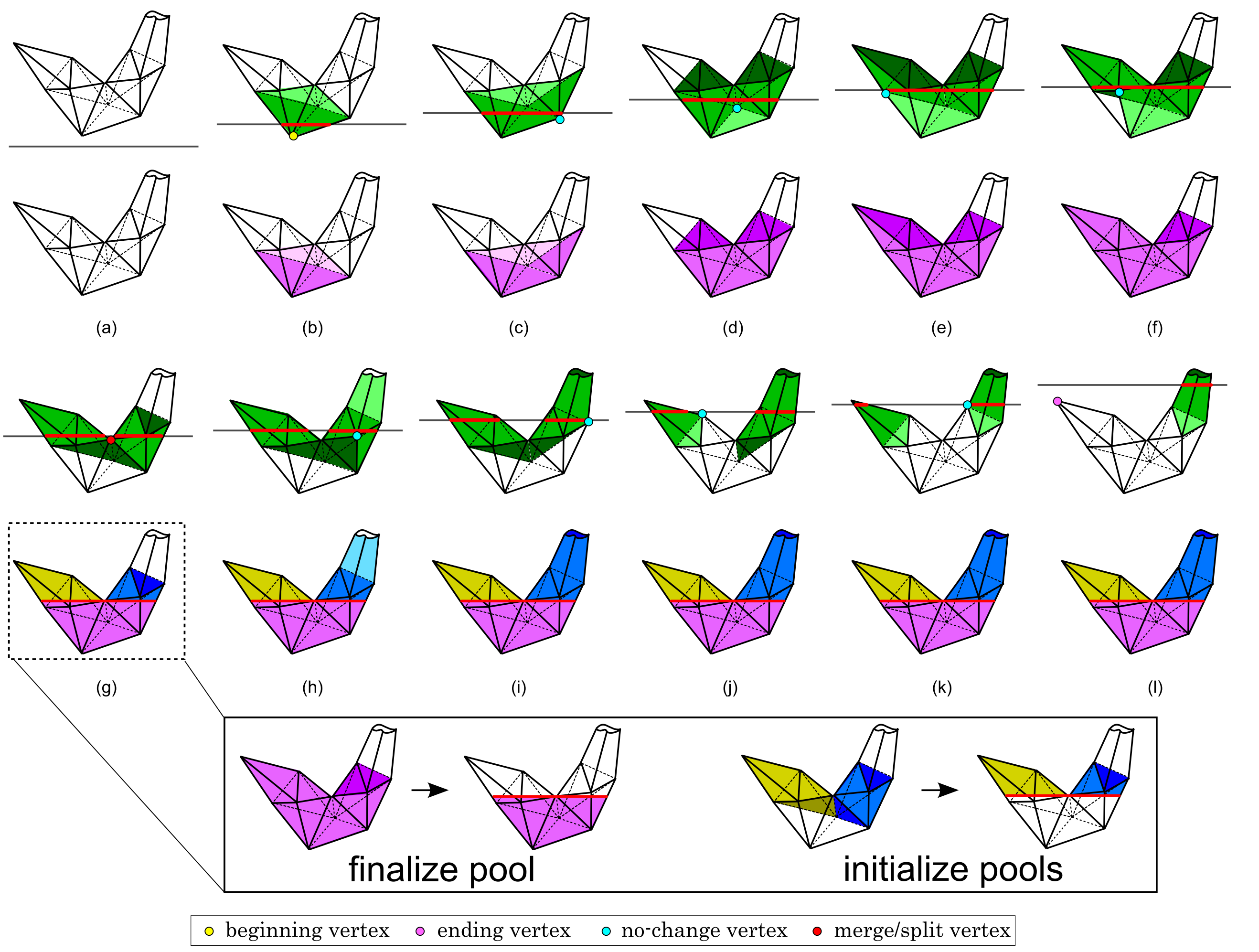
We generate a new pool p at a z-coordinate z where the slice topology changes by performing the following operations that *initialize* the pool defined by the outer boundary cycle b_{outer} at z. Letting $inner(b_{outer})$ be the set of enclosed inner boundary cycles associated with b_{outer}, first, we assign the triangles in b_{outer} and each $b_{inner} \in inner(b_{outer})$ to p.

Algorithm 2 FinalizePool(b_{outer}, z)

```
Input: $b_{outer}$: outer boundary cycle, $z$: z-coordinate
Output: $p$: pool completed at $z$
$p \leftarrow b_{outer}$-> pool
$b_{outer}$-> pool $\leftarrow$ 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 $t \in$ ($p$-> $T_j$) 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
    $p$-> $T_j \leftarrow T_{cut}$
end for
$p$->TopFace $\leftarrow \bigcup_{j=1}^{n} (\partial s)_j$
return $p$
```

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 $p_{sweep}(z^+)$. 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



(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
finalize pool
initialize pools
● beginning vertex ● ending vertex ● no-change vertex ● merge/split vertex

Figure 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 2; the bottom drawing shows the corresponding construction of pools.

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 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 C. 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 $V(z)$ as input and returns a set of boundary cycles generated, completed, and updated at z, respectively (refer to Algorithm 5). **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.

Algorithm 3 SimplifiedCasePoolSegmentation(V)

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

A series of steps to construct pools based on these rules is illustrated in Figure 4. When a new boundary cycle is generated, we initialize a new pool defined by the boundary cycle (Figure 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 4 (c)-(f) and (h)-(k)). When one boundary cycle splits into multiple boundary cycles (Figure 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 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.

3.2.1 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, the appearance of the 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 4.1), 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, 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 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 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 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 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 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 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 cy-

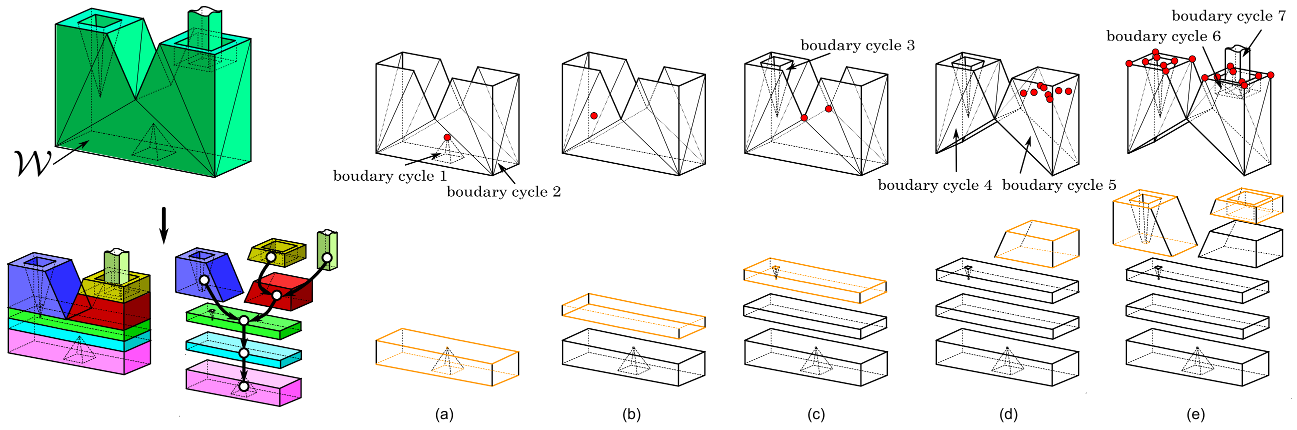


Figure 5: Figures (a)-(e) show how pools are constructed according to generation, completion, and updating of boundary cycles in general pool segmentation. We segment $\mathcal{W}$ into pools where the topology of slice polygon changes. The top drawing in each subfigure shows the boundary cycles just 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.

cles, 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 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 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 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 3.1.2.

4. 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 2.3, 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.

4.1 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 (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.

Algorithm 4 GeneralCasePoolSegmentation(V)

```
  Input: $V$: set of vertices in $\mathcal{W}$
  where $V = \{V(z_1), V(z_2), \cdots, V(z_n)\}$ ($z_i < z_j$ if $i < j$)
  // $V(z_i)$ is a set of vertices whose z-coordinate is $z_i$
  for $i = 1$ to $n$ do
    $(G, C, U) \leftarrow$ ProcessVertices($V(z_i)$)
    $(G_{inner}, G_{outer}) \leftarrow$ 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 $b$-> pool
      $b$-> pool-> $T_j \leftarrow$ ($b$-> pool-> $T_j$) $\cup$ ($b$-> $T_{new}(z_i)$)
    end for
    for each $b_{outer} \in C_{outer}$ do
      $P_{final} \leftarrow P_{final} \cup$ 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 $b_{outer}$-> pool $\neq$ nil then
        $P_{final} \leftarrow P_{final} \cup$ 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 $b_{outer}$-> pool $\neq$ nil then
        $P_{final} \leftarrow P_{final} \cup$ 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
      $P_{init} \leftarrow P_{init} \cup$ 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
```

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

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

Then the volume of this pool can be computed using equation (2):

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

Table 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	1552
time (sec.)	0.07	0.827	2.661	2.005	3.855

5. RESULTS AND DISCUSSION

Figure 6 shows the result of our segmentation and the identified water trap regions using the presented method on an industrial cylinder head model.

Table 1 shows timing data for pool segmentation and directed graph construction on the part shown in Figure 6 and other models. The timing was performed on a computer with a 2.66 GHz Intel Core i7 CPU with 4GB 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 highly matters. 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.

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 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.

6. CONCLUSION

In this paper, 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. In the future, we plan to utilize this data structure to accelerate physics-based simulation of fluid flow inside

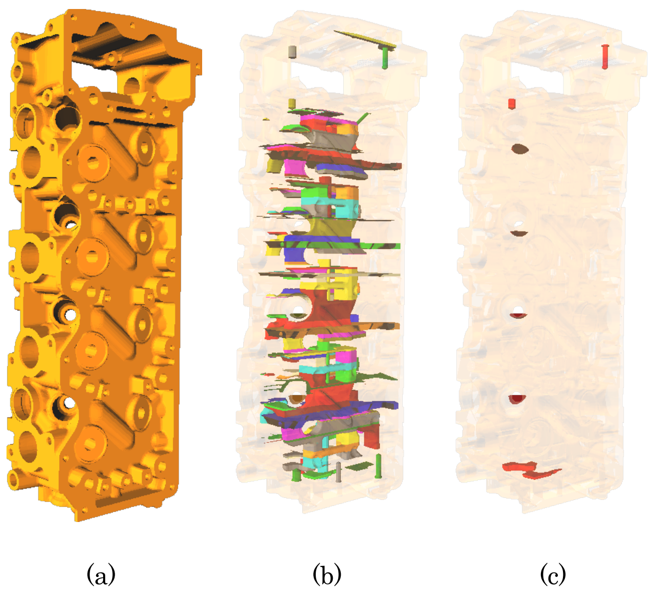


Figure 6: We applied our algorithm to a mechanical workpiece shown in (a). (b) Pool segmentation of the workpiece. Pools are assigned random colors. (c) Water trap regions of the workpiece. Note that we do not show the pools bounded by triangles which come from the corresponding bounding box for visualization purpose in this figure.

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.

7. ACKNOWLEDGMENTS

We wish to acknowledge Thomas Glau for the initial idea of tracking water draining by analyzing slice contour overlaps. We also would like thank Klaus A. Berger, Adarsh Krishnamurthy, Wei Li, and Sushrut Pavanaskar for valuable feedback. This material is based on work supported in part by Daimler AG, UC Discovery under Grant No. DIG07-10224, and the National Science Foundation under Grant No. 0621198.

8. REFERENCES

[1] D. Arbelaez, M. Avila, A. Krishnamurthy, W. Li, Y. Yasui, D. Dornfeld, and S. McMains. Cleanability of mechanical components. In *Proceedings of 2008 NSF Engineering Research and Innovation Conference*, 2008.

[2] M. Avila, C. Reich-Weiser, D. Dornfeld, and S. McMains. Design and manufacturing for cleanability in high performance cutting. In *Proceeding of 2nd International High Performance Cutting Conference*, 2006.

[3] M. C. Avila, J. D. Gardner, C. Reich-Weiser, A. Vijayaraghavan, and D. Dornfeld. Strategies for burr minimization and cleanability in aerospace and automotive manufacturing. *SAE J. Aerospace*, 114(1):1073–1082, 2005.

[4] K. Berger. Burrs, chips and cleanness of parts - activities and aims in the German automotive industry. In *Presentation at CIRP Working Group on Burr Formation*, 2006.

[5] P. Bose and G. Toussaint. Geometric and computational aspects of gravity casting. *Computer-aided Design*, 27(6):455–464, 1995.

[6] P. Bose, M. J. van Kreveld, and G. T. Toussaint. Filling polyhedral molds. In *Workshop on Algorithms and Data Structures*, pages 210–221, 1993.

[7] S. McMains. Chapter 7 Slicing. In *Geometric Algorithms and Data Representation for Solid Freeform Fabrication*, pages 102–114. PhD thesis, 2000.

[8] S. McMains and C. Séquin. A coherent sweep plane slicer for layered manufacturing. In *Proc. 5th ACM Symposium on Solid Modeling and Applications*, pages 285–295, 1999.

[9] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217 – 236, 1978.

[10] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26(3):58, 2007.

[11] G. Reeb. Sur les points singuliers d'une forme de pfaff completement intergrable ou d'une fonction numerique [On the singular points of a complete integral pfaff form or of a numerical function]. *Comptes Rendus Acad. Science Paris 222*, pages 847–849, 1946.

[12] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 15:1177–1184, November 2009.

[13] Y. Yasui and S. McMains. Testing an axis of rotation for 3D workpiece draining. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 223–233, New York, NY, USA, 2009. ACM.

[14] Y. Yasui and S. McMains. Testing a rotation axis to drain a 3D workpiece. *Computer-Aided Design*, In Press, Corrected Proof, 2011.

APPENDIX

We manage boundary cycles by modifying the data structure introduced by McMains' sweep plane slicing algorithm [8].

A. VERTEX CLASSIFICATION IMPLEMENTATION

We implemented the algorithm using a half-edge data structure [9]. 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 7.

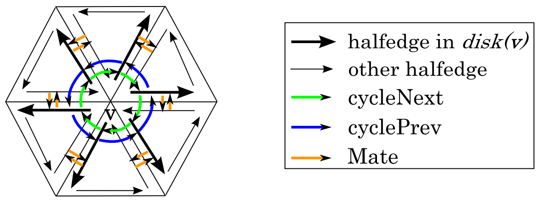


Figure 7: Given a vertex v, the disk cycle $disk(v)$ is a set of halfedges extending from v. The halfedges in $disk(v)$ are ordered clockwise when viewed from the exterior of $\mathcal{W}$ and connected in the form of circular double-linked list.

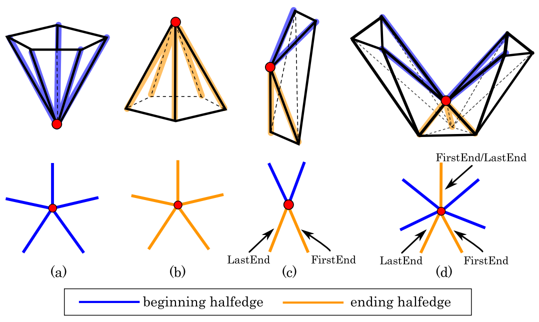


Figure 8: (a) Beginning Vertex (b) ending vertex (c) no-change vertex (d) merge/split vertex. The bottom row shows the configuration of halfedges in the corresponding disk cycle. We call the first and last ending halfedge of consecutive ending halfedges in the disk cycle *FirstEnd* and *LastEnd*.

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 8). 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*.

B. BOUNDARY CYCLES IMPLEMENTATION

As explained in 3.1, 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.

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.

B.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.

B.2 Ending Vertex

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

B.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 8 (c)). Then, we update pointers as follow.

```
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.

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.

B.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 8 (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 9).

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

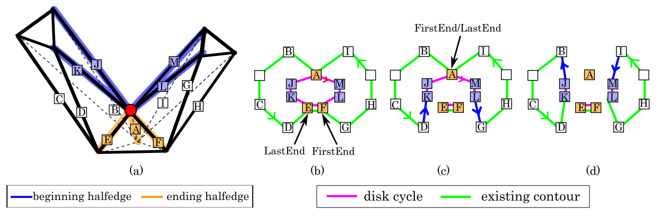


Figure 9: (a) Model with merge/split vertex v (indicated vertex). Blue edges indicate beginning halfedges and orange edges indicate ending halfedges in $disk(v)$. (b) The halfedges in $disk(v)$ (connected with magenta lines) and the halfedges in the existing boundary cycle (connected with green lines). (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.

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.

C. 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 5. 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 B.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

Algorithm 5 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
    $b \leftarrow$ ProcessBeginningVertex$(v)$
    $G \leftarrow G \cup b$.
  else if $v$ is an ending vertex then
    $b \leftarrow$ ProcessEndingVertex$(v)$
    $C \leftarrow C \cup b$.
  else if $v$ is a merge/split vertex then
    $(B_C, B_G) \leftarrow$ ProcessMergeSplitVertex$(v)$
    $C \leftarrow C \cup B_C$.
    $G \leftarrow G \cup B_G$.
  else if $v$ is a no-change vertex then
    $(b, T_{inserted}) \leftarrow$ ProcessNoChangeVertex$(v)$
    // assuming initially $(b\text{->} T_{new}(z)) = \emptyset$
    $b\text{->} T_{new}(z) \leftarrow (b\text{->} T_{new}(z)) \cup T_{inserted}$
    $U \leftarrow U \cup b$.
  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 $(G, C, U)$
```

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.