# An Optimal Algorithm for 3D Triangle Mesh Slicing

Rodrigo Minetto[1], Neri Volpato[2], Jorge Stolfi[3],
Rodrigo M. M. H. Gregori[1] and Murilo V. G. da Silva[1]

[1]*Department of Informatics, Federal University of Technology - Paraná (UTFPR), Brazil*

[2]*Department of Mechanical Engineering, Federal University of Technology - Paraná (UTFPR), Brazil*

[3]*Institute of Computing, University of Campinas (UNICAMP), Brazil*

**Abstract**

We describe an algorithm for slicing an unstructured triangular mesh model by a series of parallel planes. We prove that the algorithm is asymptotically optimal: its time complexity is $\mathcal{O}(n \log k + k + m)$ for irregularly spaced slicing planes, where $n$ is the number of triangles, $k$ is the number of slicing planes, and $m$ is the number of triangle-plane intersections segments. The time complexity reduces to $\mathcal{O}(n + k + m)$ if the planes are uniformly spaced or the triangles of the mesh are given in the proper order. We also describe an asymptotically optimal linear time algorithm for constructing a set of polygons from the unsorted lists of line segments produced by the slicing step. The proposed algorithms are compared both theoretically and experimentally against known methods in the literature.

*Keywords:* Additive manufacturing, triangle-plane intersection, contour construction algorithm, algorithm complexity, process planning.

## 1. Introduction

*Additive layered manufacturing*, also known as *3D printing*, is the technique of building a physical object by laying down successive layers of material. The object is typically defined by a three-dimensional geometric model created with a CAD system, generated by computer tomography or magnetic resonance data [1, 2], or by other means.

Before being sent to the 3D printer, the geometric model must undergo *process planning*, which means a sequence of tasks that includes: orienting and positioning the object in the printer's workspace, cutting the geometric model into layers, adding support structures if required, and finally planning the printer's toolpath [3].

[1]Corresponding author: Rodrigo Minetto, Tel. +55 41 3310 4746, Fax. +55 41 3310 4646, e-mail: rminetto@dainf.ct.utfpr.edu.br.

The slicing process can be divided into four sub-tasks as shown in Figure 1. These steps can consume up to 60% of the entire process planning time [4].

. . .

$\Downarrow$

| **(1)** Choosing the $z$-coordinates of the slicing planes | |
|---|---|
| Uniform slicing<br>(constant layer thickness) | Adaptive slicing<br>(variable layer thickness) |
| $\Downarrow$ | |
| **(2)** *Mesh slicing*: computing plane-triangle intersections | |
| $\Downarrow$ | |
| **(3)** *Contour construction*: assembling segments into polygons | |
| $\Downarrow$ | |
| **(4)** Determining the sense of polygons (clockwise or counterclockwise) | |

$\Downarrow$

. . .

Figure 1: Slicing sub-tasks.

This paper describes optimal algorithms to solve the mesh slicing step (Section 3) and the contour construction step (Section 4). For simplicity, in this paper, we will refer to the mesh slicing task only as slicing.

### *1.1. Slicing*

In the slicing step, the geometric model is intersected with parallel planes to obtain the contour of each material layer. We are not concerned in this paper with the selection of these planes — they need to be known a priori. See Figure 2. This step can be done with a constant layer thickness (*uniform slicing*) or with variable layer thickness (*adaptive slicing*). Adaptive slicing provides better surface quality in critical features of the printed model while saving time in regions where rougher finish is acceptable [3].
For greater generality, 3D printing software commonly assumes that the geometric model is reduced to an unordered and unstructured set of triangles that approximates the surface of the object. This representation is a de facto industry standard, embodied in the popular STereoLithography (STL) file format. Therefore, the primary result of the slicing step is also an unordered and unstructured set of line segments on each slicing plane.

### *1.2. Contour construction*

The segments produced by slicing must be organized into one or more closed polygons that delimit the interior (i.e. the area with and without material inside the part) of the object on the corresponding layer. See Figure 3.

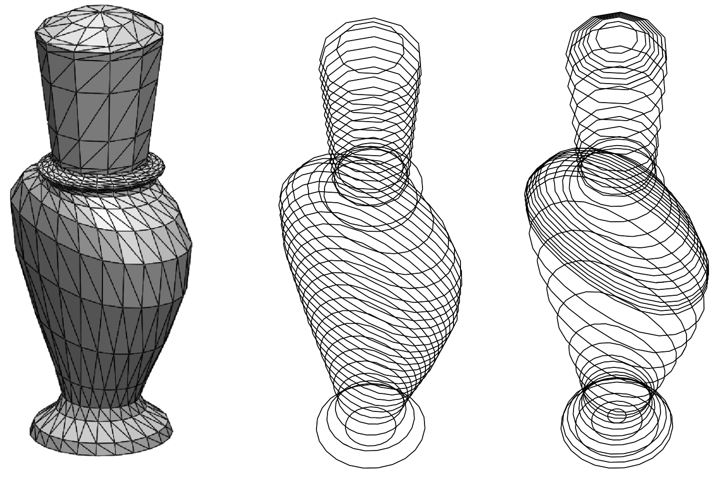


Figure 2: A triangle mesh of the surface of a 3D object model (a) and examples of uniform slicing (b) and adaptive slicing (c).

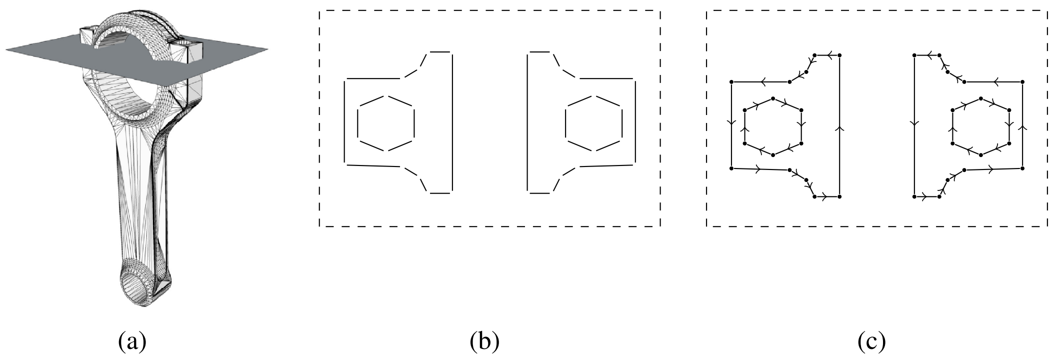


Figure 3: Selected steps in the additive layered manufacturing process planning: (a) slicing the triangle mesh, generating a set (b) of line segments on each plane, that are connected into closed loops (c).

This contour construction step is required because the generation of machine control code for most additive layered manufacturing processes requires a polygonal description of the cross-section, that provides both its perimeter (to accurately build the object's surface) and its enclosed region (to fill the object's interior). Moreover, many processes require the computation of offset contours, e. g. to compensate for nozzle, beam diameter, etc, or to deposit an object shell of prescribed thickness or of different composition; and offset algorithms typically require polygons, rather than unstructured lists of segments.

### *1.3. Mesh consistency*

For the purposes of additive layered manufacturing, the triangle mesh must divide space into two regions, the *interior* and *exterior* of the solid object. For that purpose it must be a closed orientable 2-manifold, without self-intersections or spurious contacts. In particular, the intersection between any two triangles must be either empty, or a single edge of both, or a single vertex of both. Every edge must be shared by exactly two triangles that use this edge in opposite directions, and the triangles incident to any vertex must form a single pyramidal cone [5].

Nevertheless, it is desirable that the software handling the mesh should be *robust*, that is, tolerant of defects of the input mesh, such as those that may be created by acquisition and rounding errors, contacts between different parts of the object, and mistakes in previous processing steps. Even if the software does not correct such defects, it should always terminate normally and produce a well-formed output that preserves as much information as possible about the input mesh. For example, if the input mesh is not closed, the output of the slicing and closing steps should be the correct set of segments organized as open polygonal lines.

### *1.4. Contributions*

The contributions of this paper are three-fold: (1) an optimal algorithm for the triangle mesh slicing problem with arbitrary layer thickness, that runs in $\mathcal{O}(n \log k + k + m)$ where $n$ is the number of triangles, $k$ is the number of planes, and $m$ is the total number of segments (plane-triangle intersections) for the model. The algorithm runs in $\mathcal{O}(n + k + m)$ time if the layer thickness is uniform, or if the triangles are previously sorted by $z$-coordinate; (2) a proof that the lower bound for the unsorted non-uniform slicing problem is $\Omega(n \log k + k + m)$; and (3) an optimal (linear time) algorithm for contour construction.

### *1.5. Structure of the paper*

The remainder of the paper is organized as follows. Section 2 reviews some related work. The slicing and contour construction algorithms are described in Sections 3 and 4, respectively. The experimental comparison of both algorithms are reported in Section 5. Section 6 states the conclusions.

## 2. Related work

In this section we review the literature of additive layered manufacturing process planning that is closely related to the problems considered here, namely, algorithms for triangle mesh slicing and contour construction.

### *2.1. Mesh slicing*

The survey of Pandey *et al.* [6] covers the main slicing algorithms up to 2003. According to them, such algorithms can be broadly divided into *slicing of tessellated models*, that take as inputs a mesh of triangles, and *direct slicing*, that work directly on more general CAD models, e.g. NURBS. The direct slicing strategy may provide more accurate results, by avoiding the errors introduced by the triangulation of the surface [7, 8]. However, in some applications, the object model is obtained already in tessellated form.

The naive slicing algorithm, described by Chalasani and Grogan in [9], consists in testing every cutting plane against every triangle. Thus, its time-complexity is $\mathcal{O}(kn)$. This naive algorithm is still widely used, and there are many papers that improve some of its aspects, via parallelism [4] and optimization of memory usage [10, 11]. However, relatively little work has been published on reducing its computing time.

In 1998, Tata *et al.* [12] described a faster algorithm that uses a two-level tree structure to reduce the number of triangle-plane intersection tests. In their algorithm, triangles are sorted by their minimum $z$-coordinates $z_{\min}$, and then grouped so that triangles with the same $z_{\min}$ value are clustered together. Each group is then divided into sub-groups, according to their maximum $z$-coordinates $z_{\max}$. This tree can be constructed in $\mathcal{O}(n \log n)$ time. Then the $nk$ triangle-plane intersection tests of the naive algorithm are replaced by plane-subgroup intersection tests, since each plane either intersects all triangles in a subgroup, or none of them. In favorable situations (with many triangles in each subgroup) this method can save substantial time. However, in the worst case (when all vertices have distinct $z$-coordinates) the algorithm still requires $\mathcal{O}(nk)$ time.

In 1999, McMains and Séquin [5], described a sweep-plane algorithm for slicing a structured mesh. Their algorithm simulates the process of continuously sweeping a plane across the triangle mesh, maintaining the set of polygons that comprise its intersection with the plane. These polygons are updated whenever the sweep plane hits a vertex of the mesh, or coincides with one of the specified slicing planes. The intersection of the model with the sweeping plane is organized into a set of closed loops, thus dispensing with a separate contour construction step. McMains and Séquin claim that their algorithm runs in $\mathcal{O}(n \log n + m)$ time for objects with simple topology (low constant genus) but may take $\Omega(n^2)$ time for objects with genus $\Omega(n)$. On the other hand, this algorithm requires an input mesh with full topological (adjacency and incidence) information, not just a set of triangles. Béchet et al.[13], described an algorithm to recover the topology of the mesh from an STL file, by using a binary tree in which the vertices are stored and sorted; this approach runs in time $\mathcal{O}(n \log n)$ in the worst case.

An ideal algorithm would retrieve only the set of triangles sliced by each plane, and the amount of work should be linear on the size of this set, that is $\mathcal{O}(n + k + m)$ rather than

$\mathcal{O}(nk)$. For uniform slicing, the algorithm of Huang *et al* [14] (2012) achieves this optimal time complexity. However, this algorithm depends on the layer thickness being constant to determine the planes that intersect a given triangle, and thus cannot be applied to adaptive slicing.

### 2.2. *Contour construction*

The naive contour construction algorithm starts with an arbitrary segment $s$, from the input list $S$. That segment becomes the first side of a new polygon. Then one looks for another segment $r$ in $S$ that has an endpoint coincident with one endpoint of $s$. Such a segment must exist if the input is a slice of a well-formed 2-manifold mesh. Once such a segment is found, it is reoriented as needed, and appended to the polygon; and the search is repeated with $r$ in place of $s$ until obtaining a closed loop. Segments are removed from $S$ as they are processed. The entire process is repeated to get additional contours until $S$ is empty. When applied to $m$ line segments, this approach requires $\mathcal{O}(m^2)$ time.

In 2003, Park [15] proposed to use the Bentley-Ottmann [16] sweep line algorithm to solve the contour construction problem. The general Bentley-Ottmann algorithm runs in $\mathcal{O}((m + p)\log m)$ time, where $m$ is the number of line segments and $p$ the number of intersections between those segments. If the only intersections are the segment endpoints (as is the case for well-formed inputs), then $p = 2m$. Thus the Bentley-Ottmann algorithm can be simplified, and runs in $\mathcal{O}(m \log m)$ time.

Zeng et al. [8] and Qi et al. [17] proposed to convert the STL model into layers of grids by using *layer depth normal image* (LDNI) sampling decomposition, that is, each object layer is sampled by a set of parallel rays in order to obtain a list of points. Then, by using the neighborhood information, the points are connected for the contour construction. As described by the authors, depending of the STL model, it may be required to adopt a high sampling resolution to preserve sharp features.

## 3. The Slicing Algorithm

### 3.1. *Statement of the problem*

Formally, the input for the slicing problem is assumed to consists of $n$ triangles $T = (T[1], T[2], \ldots, T[n])$, in arbitrary order; and $k$ slicing planes, perpendicular to the $Z$ axis, defined by a list of increasing $z$-coordinates $P = (P[1], P[2], \ldots, P[k])$, with constant spacing between them (for uniform slicing) or arbitrary spacing (for adaptive slicing).

Each triangle $T[j]$ is defined by three vertices $T[j].v_1$, $T[j].v_2$ and $T[j].v_3$. For a given triangle, the lowest and highest $z$-coordinates of the vertices are refered to as $T[j].z_{\min}$ and $T[j].z_{\max}$ respectively. See Figure 4.

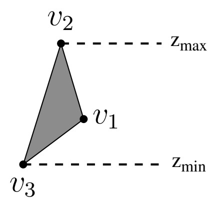


Figure 4: Attributes of a triangle.

The output of the slicing problem is a list $S[i]$, containing the line segments generated by all triangles that intersect plane $P[i]$.

Let $n_i$ be the number of triangles that intersect the plane at $z$-coordinate $P[i]$, and $k_j$ be the number of planes that intersect triangle $T[j]$. Also, let $\overline{n}$ be the average of the counts $n_i$ and $\overline{k}$ be the average of the counts $k_j$, that is

$$\overline{n} = \frac{1}{k}\sum_{i=1}^{k} n_i \qquad\qquad \overline{k} = \frac{1}{n}\sum_{j=1}^{n} k_j. \tag{1}$$

Then the output of the slicing problem consists of $m = n\overline{k} = k\overline{n}$ segments, resulting from the triangle-plane intersections.

In the slicing algorithm, we assume that the $Z$-coordinates of all slicing planes are distinct from the $Z$-coordinates of all vertices. We ensure this condition while reading the input model, by rounding all vertex coordinates to even multiples of some basic unit $\epsilon$ (say, 0.005 mm) and all plane $z$-coordinates to odd multiples of $\epsilon$. Then a triangle $T[j]$ intersects $P[i]$ if and only if $P[i]$ is *strictly* between $T[j].z_{\min}$ and $T[j].z_{\max}$. Note that coordinates can be rounded already in the input mesh file.

We also discard any triangle that has two or more coincident vertices. Note that this clean-up does not change the set of points of 3-space that lie on the mesh.

## *3.2. The main algorithm*

Our slicing algorithm (Algorithm 1) uses a sweeping plane strategy similar to that of McMains and Séquin [5], but highly simplified and optimized for unstructured triangle sets. The input data consists of the list of triangles $T$, the list of plane $z$-coordinates $P$, the layer thickness $\delta$, and a Boolean parameter *srt*. The parameter $\delta$ should be positive (and an even multiple of $\epsilon$) if the planes have uniform thickness, that is $P[i] = P[i-1] + \delta$ for all $i \in \{2, \ldots, k\}$; otherwise, the parameter $\delta$ should be set to zero. The Boolean parameter *srt* should be **true** if and only if the triangle list $T$ is already sorted by the $z_{\min}$ coordinates.

**Algorithm 1**

```
1: INCREMENTAL-SLICING ($n$, $T[1 \ldots n]$, $k$, $P[1 \ldots k]$, $\delta$, *srt*)
2:    // Split the triangle list.
3:    $L[1 \ldots k+1] \leftarrow$ BUILD-TRIANGLE-LISTS ($n, T, k, P, \delta$, *srt*);
4:    // Perform a plane sweep.
5:    $A \leftarrow \{\ \}$;
6:    for $i \in \{1, \ldots, k\}$ do
7:        $A \leftarrow A \cup L[i]$;
8:        $S[i] \leftarrow \emptyset$;
9:        for each $t \in A$ do
10:           if $t.z_{\max} < P[i]$ then
11:               $A \leftarrow A \setminus \{t\}$;
12:           else
13:               $(q_1, q_2) \leftarrow$ COMPUTE-INTERSECTION ($t, P[i]$);
14:               $S[i] \leftarrow S[i] \cup \{(q_1, q_2)\}$;
15:           end if
16:       end for
17:   end for
18:   return $S[1 \ldots k]$;
19: end
```

In step 3 of Algorithm 1, the input triangle list $T[1 \ldots n]$ is partitioned into $k+1$ lists of triangles $L[1], L[2], \ldots L[k+1]$, where $L[i]$ consists of all triangles whose $z_{\min}$ lies between the $z$-coordinates $P[i-1]$ and $P[i]$; $P[0]$ and $P[k+1]$ being assumed to be $-\infty$ and $+\infty$, respectively. This step is shown in Figure 5 and detailed in Section 3.3.

In steps 6-17 we compute the triangle-plane intersections for each plane. Specifically, we simulate the march of a plane sweeping the mesh, jumping from each slicing plane to the next. During the simulation, we keep a set $A$ of the *active* triangles, those that may intersect with the next slicing plane at coordinate $P[i]$. In step 7, we add to the active set $A$ all triangles that have $z_{\min}$ between $P[i-1]$ and $P[i]$. In step 11, we remove from the set $A$ those triangles whose $z_{\max}$ lies below $P[i]$, since they will not generate any intersection with that plane or any subsequent plane. The remaining triangles of $A$ intersect the slicing plane at $P[i]$ at a non-trivial line segment, because $t.z_{\min} < P[i] < t.z_{\max}$. In steps 13 and 14, the intersection segment $\{q_1, q_2\}$ is computed and stored in the list of segments $S[i]$ of that plane.

## *3.3. Grouping the triangles*

The procedure BUILD-TRIANGLE-LISTS, Algorithm 2, is used by Algorithm 1 (step 3)

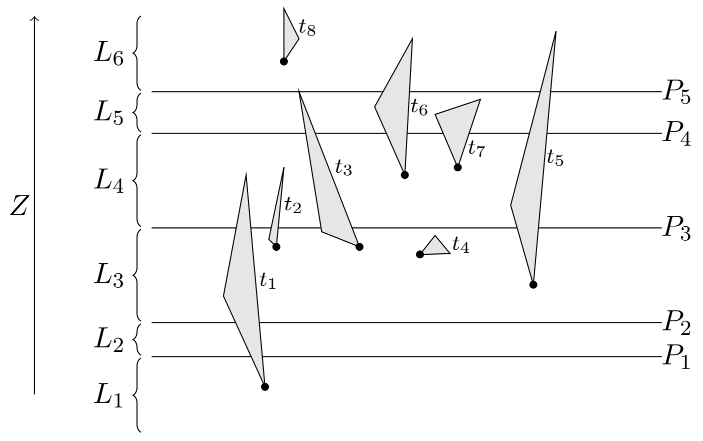


Figure 5: An example of triangle grouping by $z_{\min}$ (depicted by black dots) in step 3 of Algorithm 1. The resulting lists are: $L_1 = \{t_1\}, L_2 = \{\ \}, L_3 = \{t_2, t_3, t_4, t_5\}, L_4 = \{t_6, t_7\}, L_5 = \{\ \}$ and $L_6 = \{t_8\}$. After step 7 with $i = 4$, the sweeping plane is assumed to lie between $P_3$ and $P_4$, and the active set will be $A = \{t_5, t_3, t_6, t_7\}$.

to split the list $T$ into the lists $L[1 \ldots k + 1]$, according to the triangle's $z_{\min}$ field. There are three cases to consider: uniform slicing, non-uniform slicing of previously sorted triangles, and non-uniform slicing of unsorted triangles.

In the uniform slicing case (steps 3-9), the index $i$ of the correct list for each triangle $t$ is computed directly from the $t.z_{\min}$ coordinate, the $Z$-coordinate $P[1]$ of the first plane, and the layer thickness $\delta$, as described by Huang et al. [14]. Otherwise, if the input list $T$ is already sorted by the $z_{\min}$ field, the split can be done by a single merge-style simultaneous traversal of the lists $T$ and $P$ (steps 10-17). Otherwise, the BINARY-SEARCH procedure (Algorithm 3) is used to locate the correct list index $i$ for each triangle $t$ (steps 18-23).

## *3.4. Complexity*

It is easy to see that every step of the INCREMENTAL-SLICING algorithm, except step 3, takes a total time that is at most proportional to $n$, $k$, $m$ or sums of these variables. Namely, step 7 is executed $k$ times, and adds to the active set $A$ every triangle exactly once (except those in $L[k + 1]$), so its total cost is $\mathcal{O}(n + k)$. Step 11 is executed at most once for each triangle, and steps 13 and 14 are executed once for each output segment, therefore their cost are $\mathcal{O}(n)$ and $\mathcal{O}(m)$ respectively; and the cost of step 10 is then $\mathcal{O}(n+m)$. We conclude that INCREMENTAL-SLICING, except for step 3, runs in $\mathcal{O}(n + k + m)$ time.

As for BUILD-TRIANGLE-LISTS, if the input list $T$ is already sorted by $z_{\min}$, or if the plane coordinates are uniformly spaced, the output lists are built in $\mathcal{O}(n+k)$ time. Otherwise they can be built in $\mathcal{O}(n+k)$ time plus the cost of $n$ calls to BINARY-SEARCH (Algorithm 3),

**Algorithm 2**

```
1:  BUILD-TRIANGLE-LISTS ($n$, $T[1 \ldots n]$, $k$, $P[1 \ldots k]$, $\delta$, srt)
2:    $L[1 \ldots k+1] \leftarrow \{\ \}$;
3:    if $\delta > 0$ then  // Uniform slicing.
4:      for each $t \in T[1 \ldots n]$ do
5:        if $(t.z_{\min} < P[1])$ then { $i \leftarrow 1;$ }
6:        else if $(t.z_{\min} > P[k])$ then { $i \leftarrow k+1;$ }
7:        else { $i \leftarrow \lfloor (t.z_{\min} - P[1])/\delta \rfloor + 1;$ }
8:        $L[i] \leftarrow L[i] \cup \{t\}$;
9:      end for
10:   else if srt then  // Pre-sorted triangles.
11:     $j \leftarrow 1$;
12:     for $i \in \{1, \ldots, k\}$ do
13:       while $(T[j].z_{\min} < P[i])$ and $(j \leq n)$ do
14:         $L[i] \leftarrow L[i] \cup \{T[j]\}$;
15:         $j \leftarrow j+1$;
16:       end while
17:     end for
18:   else  // General case.
19:     for each $t \in T[1 \ldots n]$ do
20:       $i \leftarrow$ BINARY-SEARCH $(k, P, t)$;
21:       $L[i] \leftarrow L[i] \cup \{t\}$;
22:     end for
23:   end if
24:   return $L[1 \ldots k+1]$;
25: end
```

which runs in $\mathcal{O}(\log k)$ time. Therefore, the total cost of BUILD-TRIANGLE-LISTS in this case is $\mathcal{O}(n \log k + k + n)$.

We conclude that the cost of INCREMENTAL-SLICING, including step 3, is $\mathcal{O}(n+k+m)$ for uniform slicing or non-uniform slicing with pre-sorted triangles, or $\mathcal{O}(n \log k + k + m)$ otherwise.

The worst case cost in terms of $n$ and $k$ is when every triangle intersects every plane. In that case, $m = nk$, which dominates the other terms; so the time complexity is $\Theta(nk)$. Since the algorithm must output all those $nk$ segments, it is asymptotically optimal even in that case.

An alternative way to build the lists $L[1 \ldots k+1]$ in the general case would be to sort the $n$ values $t.z_{\min}$ in $\mathcal{O}(n \log n)$ time, and then proceed as in the sorted case. With this approach,

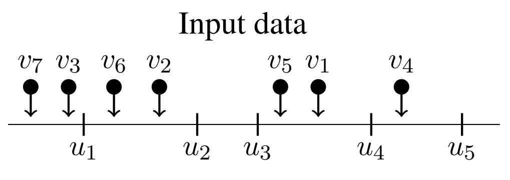
**Algorithm 3**

```
BINARY-SEARCH ($k, P[1 \dots k], t$)
    **if** $t.z_{\min} > P[k]$ **then return** $k+1$; **end if**
    $l \leftarrow 0$;   // *Lowest plane.*
    $r \leftarrow k$;   // *Highest plane.*
    **while** $r - l > 1$ **do**
        // *Here* $P[l] < t.z_{\min} < P[r]$ *and* $l + r \geq 2$.
        $m \leftarrow \lfloor (l+r)/2 \rfloor$;
        **if** $t.z_{\min} > P[m]$ **then**
            $l \leftarrow m$;
        **else**
            $r \leftarrow m$;
        **end if**
    **end while**
    **return** $r$;
**end**
```

the time complexity would be $\mathcal{O}(n \log n + n + k)$, which may seem to be an improvement when $n < k$. However, it is easy to verify that $n \log n + n + k > n \log k$, if $n < k$. Therefore, the asymptotic bound would still be the same.

## *3.5. Optimality*

We now show that our INCREMENTAL-SLICING is optimal in the asymptotic, output-sensitive, worst case sense. Consider the following *multipoint search problem* $\mathbb{P}$. Each instance of $\mathbb{P}$ consists of a given list of $k$ distinct real values $u_1, u_2, \dots, u_k$ sorted in increasing order, and a given list of $n$ real values $v_1, v_2, \dots, v_n$ in arbitrary order, where each $v_j$ is distinct from all $u_i$ and is less than $u_k$. The solution of that instance is a table giving, for each $j$ in $\{1 \dots n\}$, the smallest $i$ in $\{1 \dots k\}$ such that $v_j < u_i$. See Figure 6.


Input data
$v_7$ $v_3$ $v_6$ $v_2$ $v_5$ $v_1$ $v_4$
$u_1$ $u_2$ $u_3$ $u_4$ $u_5$

(a)

Output solution

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $i$ | 4 | 2 | 1 | 5 | 4 | 2 | 1 |

(b)

Figure 6: An instance of the multipoint search problem $\mathbb{P}$ (a) and its solution (b).

Suppose that the only operations performed on the values $v_j, u_i$ are comparisons. Note that there are $k^n$ possible outputs, requiring $\log_2(k^n) = n \log_2 k$ bits of information to identify the correct output for a given instance. Since each comparison between $u_i$ and $v_j$ has only 2 possible outcomes, it provides at most 1 bit of information about the input data. Therefore the number of comparisons must be at least $n \log_2 k$. We conclude that any algorithm that solves problem $\mathbb{P}$ using only comparisons, requires at least $\Omega(n \log k)$ operations. The same conclusion holds if the algorithm is allowed to perform arbitrary tests on the values $v_j$ and $u_i$, if the output of each test has $\mathcal{O}(1)$ possible outcomes.

Now, given an instance $(k, u, n, v)$ of problem $\mathbb{P}$, construct the following instance of the triangle slicing problem: for each $u_i$, take a slicing plane $P[i]$ with $z$ coordinate $u_i$; for each $v_j$, take a triangle $T[j]$ with $z_{\min} = v_j$ and extending upwards until just above the next slicing plane. Solving the slicing problem for planes $P[1 \ldots k]$ and triangles $T[1 \ldots n]$ gives a solution to the instance $(k, u, n, v)$ of $\mathbb{P}$. Since this problem requires $\Omega(n \log k)$ operations to solve, the slicing problem too must require that much work.

Any algorithm that solves the slicing problem must also output $k$ lists containing the $m$ segments, therefore a lower bound for the non-uniform slicing problem is $\Omega(n \log k + k + m)$. We conclude that Algorithm 1 is asymptotically optimal.

## 4. Contour construction

As explained in Section 3, the output of Algorithm 1 is a list $S[i]$ for each plane in $i = \{1 \ldots k\}$, containing the line segments where the plane at $z$-coordinate $P[i]$ intercepts the triangles of the mesh. The next step of the additive layered manufacturing process planning is to assemble those segments into a set of closed polygons (this step can also be executed in parallel with Algorithm 1, between steps 16 and 17).

### *4.1. Statement of the problem*

The input of the contour construction problem is a list $S[i]$ of $q$ line segments, in arbitrary order and orientation, resulting from the intersection of the input triangles with a plane $P[i]$. The output is a set of $r$ contours $C = (C[1], \ldots, C[r])$ comprising those segments. Each contour is a polygon whose sides are some of the input segments, joined and oriented head-to-tail; so that each input segment is in exactly one contour.

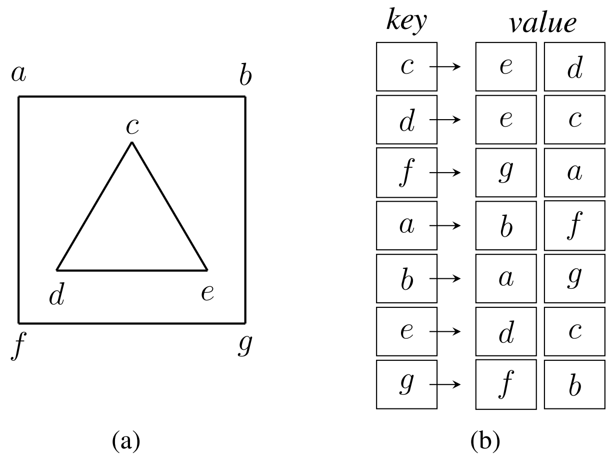
If the input triangles (after rounding) are a well-formed mesh (a closed oriented 2D manifold without self-intersections), those chains will be a set of pairwise disjoint closed-loops, some of them interior to the object, some of them exterior to it. In this case, the endpoints of each segment are distinct, no two segments intersect except at a single endpoint, and each endpoint belongs to exactly two segments. However, if the input triangles are not a well-formed mesh, there may be endpoints shared by any number of segments. The number will be odd if and only if that endpoint is on the free border of the input mesh. In this case, some of the chains must be open polygons.

### *4.2. Description of the algorithm*

Our proposal for this step is described by the CONTOUR-CONSTRUCTION procedure (Algorithm 4). We use a hash table [18] to solve this problem in linear time. Each entry in the hash table has a *key*, which is a single point $u$ of the slicing plane that is the endpoint of one or more segments; and a *value*, that is a pair $(v, w)$ of points such that $\{u, v\}$ and $\{u, w\}$ are endpoints of two segments. See Figure 7.

**Algorithm 4**

```
1: CONTOUR-CONSTRUCTION $(q, S[1 \dots q])$
2:   // Insert the segments into the hash table.
3:   $H \leftarrow$ NEW-HASH $(q)$;
4:   for each $(u, v) \in S$ do
5:     $H \leftarrow$ INSERT-HASH $(H, u, v)$;
6:     $H \leftarrow$ INSERT-HASH $(H, v, u)$;
7:   end for
8:   // Build the closed polygons.
9:   $r \leftarrow 0$;
10:  while NUMBER-OF-ENTRIES$(H) > 0$; do
11:    $p_1 \leftarrow$ CHOOSE-KEY $(H)$;
12:    $(p_2, last) \leftarrow$ REMOVE-ENTRY $(H, p_1)$;
13:    $j \leftarrow 2$;
14:    repeat
15:      $(u, v) \leftarrow$ REMOVE-ENTRY $(H, p_j)$
16:      if $u = p_{j-1}$ then
17:        $p_{j+1} \leftarrow v$;
18:      else
19:        $p_{j+1} \leftarrow u$;
20:      end if
21:      $j \leftarrow j + 1$;
22:    until $p_j = last$;
23:    $r \leftarrow r + 1$;
24:    $C[r] \leftarrow (p_1, p_2, \dots, p_j)$;
25:  end while
26:  return $C[1 \dots r]$;
27: end
```



| *key* | *value* | |
|---|---|---|
| $c$ | $e$ | $d$ |
| $d$ | $e$ | $c$ |
| $f$ | $g$ | $a$ |
| $a$ | $b$ | $f$ |
| $b$ | $a$ | $g$ |
| $e$ | $d$ | $c$ |
| $g$ | $f$ | $b$ |

Figure 7: (a) A set of segments produced by the mesh slicing algorithm for one plane; (b) the hash table built by our contour construction algorithm from those segments.

In step 3 of Algorithm 4, the function NEW-HASH $(q)$ creates a hash table large enough to store $q$ entries. In steps 5 and 6, the function INSERT-HASH is called (twice) to insert each segment with endpoints $u$ and $v$ in the hash table $H$, with keys $u$ and $v$ respectively. Specifically, the call INSERT-HASH $(H, u, v)$ first locates the entry of $H$ with key $u$. If there is no such entry, it creates one and sets its value to $(v, \bullet)$, where $\bullet$ denotes an invalid (null) point. If the entry already exists, its value must be $(w, \bullet)$, where $w$ is some other point; in that case, the value is replaced by $(w, v)$.

In steps 14–25 of CONTOUR-CONSTRUCTION, the contours are built one by one, removing their vertices from $H$, until $H$ becomes empty. Specifically, in step 11 the function CHOOSE-KEY$(H)$ is supposed to return the key $u$ of an arbitrary entry in $H$, which becomes the starting vertex $p_1$ of the polygon. In step 12, the function REMOVE-KEY finds and removes the corresponding entry. Note that, at this moment, each unprocessed vertex should have two unprocessed neighbors. One of the two points in that entry is arbitrarily chosen to be the second vertex of the loop. See Figure 8(a).

The algorithm then repeatedly finds and removes the two neighbors $u, v$ of the current last vertex $p_j$ (step 15), finds which one is the next vertex, and extends the polygon with the new vertex $p_{j+1}$ (steps 16–21). See Figure 8(b). This iteration stops when the polygon is closed, that is, when the current last vertex $p_j$ is the neighbor *last* of $p_1$ that was not chosen in step 12 (step 22).

The next step in the process would be to identify the sense (clockwise or counterclockwise) of each loop. The point-in-polygon method proposed by Volpato et al. [19] could be used.

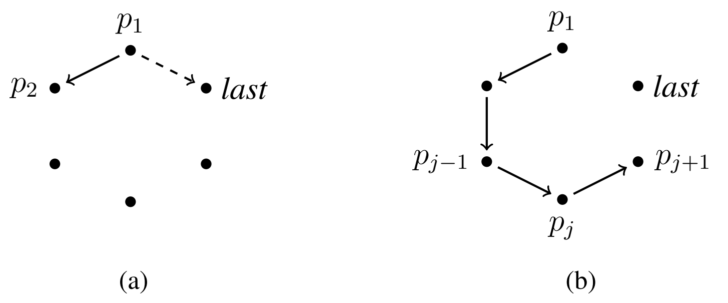


Figure 8: (a) State of the CONTOUR-CONSTRUCTION algorithm after step 12; (b) state just before step 21.

*4.3. Complexity*

Steps 5 and 6 of the contour construction algorithm call INSERT-HASH two times to create $q$ entries in $H$. The entries are removed from the hash table as they are used in steps 12 and 15, which are therefore executed a total of $q$ times. Statements 23 and 24 are executed once for each contour, and therefore at most $\lfloor q/3 \rfloor$ times. For a well dimensioned hash table, the expected time to search for, add, or remove an entry is $\mathcal{O}(1)$ [18]. Therefore, the total time complexity of algorithm CONTOUR-CONSTRUCTION is $\mathcal{O}(q)$.

Note that the sum of $q$ over all slicing planes is the parameter $m$ of Section 3, the total number of triangle-plane intersections. Therefore, the total cost of the chaining step, for all slices, is $\mathcal{O}(m)$.

## 5. Experiments

In order to evaluate the performance of our algorithms, we determined its running time on the 13 STL models listed on Table 1. For tests with uniform slicing, we used a fixed layer thickness of 0.032 mm (as used by the Stratasys Eden 250 printer). For tests with adaptive slicing, we varied the layer thickness randomly between 0.016 mm and 0.032 mm.

The execution times were measured on an Intel Core i7 machine (3.4 GHz) with 32 GB of RAM, 256K of cache L2 and 8 MB of cache L3, running the Linux operating system (64 bits). The source code of our algorithms (in C++) is publicly available on our website [20].

*5.1. Slicing*

In this section we compared the running times of our INCREMENTAL-SLICING algorithm on the models of Table 1 with our `C++` implementation of the trivial slicing algorithm as described by Park [15] and with the first part of the `slice` function included in the open source 3D slicing engine *Slic3r*. The latter is limited to uniform slicing. See Table 2 and Figure 9.

Table 1: STL models used in the experiments, showing the number of triangles ($n$), the number of planes ($k$), the average number of plane intersections per triangle ($\bar{k}$), and the total number of triangle-plane intersections ($m$).

| | | | **Uniform** slicing | | | **Adaptive** slicing | | |
|---|---|---|---|---|---|---|---|---|
| STL name | $n$ | Size (MB) | $k$ | $\bar{k}$ | $m$ | $k$ | $\bar{k}$ | $m$ |
| 01. Liver [21] | 38,142 | 6.4 | 6,242 | 61.4 | 2,341,955 | 8,327 | 81.9 | 3,125,518 |
| 02. Femur | 42,150 | 11.4 | 3,155 | 28.0 | 1,180,994 | 4,206 | 37.4 | 1,575,271 |
| 03. Bunny | 270,021 | 12.9 | 1,547 | 5.8 | 1,561,262 | 2,060 | 7.7 | 2,077,024 |
| 04. Demon [22] | 935,236 | 44.6 | 3,126 | 7.3 | 6,846,219 | 4,168 | 9.8 | 9,134,486 |
| 05. Sphenoid | 983,134 | 47.0 | 1,971 | 3.9 | 3,839,399 | 2,623 | 5.2 | 5,129,972 |
| 06. Rider [22] | 1,281,950 | 61.1 | 849 | 1.2 | 1,569,364 | 1,127 | 1.6 | 2,085,785 |
| 07. Tesla | 2,296,928 | 110.0 | 4,547 | 4.9 | 11,309,068 | 6,062 | 6.6 | 15,172,446 |
| 08. Sphere [21] | 3,060,992 | 146.0 | 6,038 | 6.2 | 19,083,782 | 8,055 | 8.3 | 25,484,004 |
| 09. Soldier [22] | 3,394,041 | 161.8 | 2,390 | 3.4 | 11,663,621 | 3,183 | 4.6 | 15,555,765 |
| 10. Bear | 6,248,232 | 298.0 | 3,961 | 6.9 | 43,346,582 | 5,280 | 9.3 | 57,836,973 |
| 11. Warrior [23] | 8,852,207 | 422.1 | 307 | 0.1 | 912,739 | 405 | 0.1 | 1,220,672 |
| 12. Robot [23] | 12,274,288 | 585.3 | 1,011 | 0.6 | 6,879,612 | 1,344 | 0.7 | 9,144,887 |
| 13. Skull | 16,668,257 | 795.0 | 4,680 | 10.9 | 182,441,957 | 6,240 | 14.6 | 243,094,438 |

On all models, and for both uniform and adaptive slicing, our INCREMENTAL-SLICING algorithm was many times faster than Park's slicing, that was found to process about 200–250 million triangle-plane pairs per second, but only 0.5 million actual segments per second, on the largest models. For uniform slicing, our procedure was slightly better than Slic3r, which is probably using an asymptotically optimal algorithm. They were found to process about 8–11 and 6–8 million items (triangles, planes, or segments) per second, respectively.

Figure 10 shows two examples of slicing contours computed with our algorithm.

## *5.2. Contour-Construction*

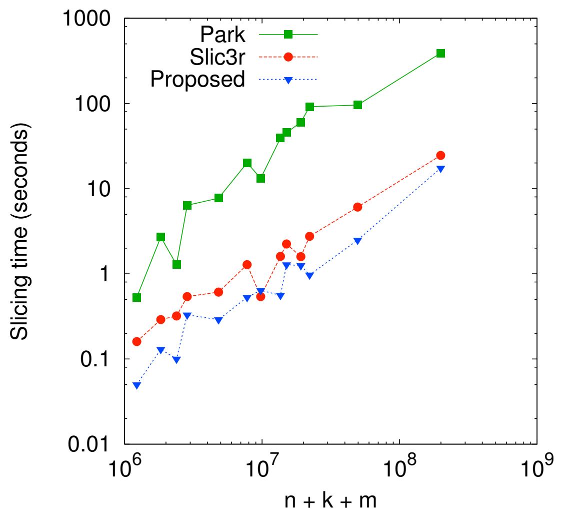
In this section we compare the performance of our CONTOUR-CONSTRUCTION algorithm against: (1) the trivial algorithm (see Section 2.2) that we implemented in C++; (2) the Bentley-Ottmann line sweep algorithm [16], as sugested by Park [15], and implemented by the CGAL [24] library in C++; and (3) the algorithm for contour construction by the open source 3D slicing engine *Slic3r* [25]. We executed the four algorithms on the models of Table 1, sliced with uniformly spaced planes. The number $k$ of planes and the number $m$ of segments found are given in Table 1. The running times are shown in Table 3 and Figure 11.

As shown in Table 3, the proposed hash-based algorithm is significantly faster than the other three (except on the smallest models). The running time was almost exactly linear in $m$, at the rate of about 1.8 million segments per second.

Surprisingly, the Park algorithm, which uses the Bentley-Ottmann algorithm and should

Table 2: Running times (in seconds) of our implementation of Park's algorithm, the `slice` function of Slic3r, and our optimal slicing algorithm, for uniform and adaptive slicing. The boldface values are the minimum running time in each row, for each type of problem. The Slic3r software is limited to uniform slicing.

| STL | Uniform slicing | | | Adaptive slicing | | |
|---|---|---|---|---|---|---|
| | Park | Slic3r | Proposed | Park | Slic3r | Proposed |
| 01. Liver | 1.28 | 0.32 | **0.10** | 1.76 | — | **0.17** |
| 02. Femur | 0.53 | 0.16 | **0.05** | 0.79 | — | **0.09** |
| 03. Bunny | 2.70 | 0.29 | **0.13** | 3.52 | — | **0.22** |
| 04. Demon | 20.12 | 1.28 | **0.53** | 23.65 | — | **1.23** |
| 05. Sphenoid | 7.73 | 0.61 | **0.29** | 10.15 | — | **0.59** |
| 06. Rider | 6.37 | 0.54 | **0.33** | 8.51 | — | **0.64** |
| 07. Tesla | 39.12 | 1.60 | **0.56** | 53.09 | — | **2.73** |
| 08. Sphere | 91.70 | 2.75 | **0.97** | 121.50 | — | **4.82** |
| 09. Soldier | 45.45 | 2.24 | **1.28** | 60.13 | — | **2.61** |
| 10. Bear | 95.71 | 6.07 | **2.49** | 126.87 | — | **7.96** |
| 11. Warrior | 13.21 | **0.54** | 0.64 | 17.52 | — | **1.47** |
| 12. Robot | 60.11 | 1.59 | **1.25** | 79.81 | — | **3.47** |
| 13. Skull | 389.80 | 24.54 | **17.42** | 519.03 | — | **40.29** |



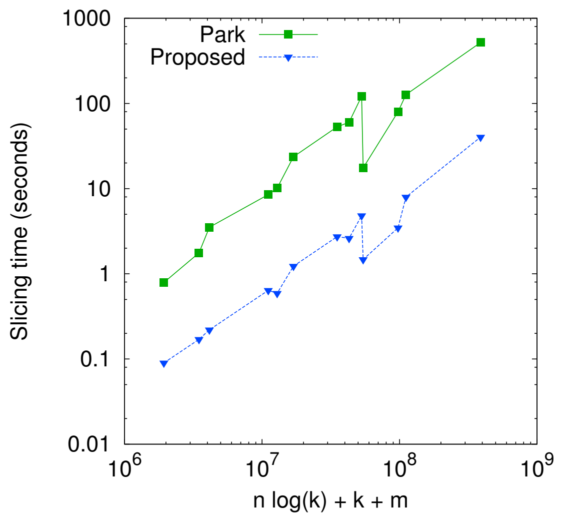


Figure 9: Running times of the three slicing algorithms (Park, Slic3r, Proposed) for uniform slicing (left) and adaptive slicing (right), on the STL models of Table 1, as a function of the respective asymptotic complexity parameters ($n + k + m$ and $n \log k + k + m$, respectively).

have theoretical worst-case bound $\mathcal{O}(m \log m)$, was much slower than the Trivial algorithm, which could take asymptotic time $\Omega(m^2)$: the speeds were 20–100 thousand and 103–460

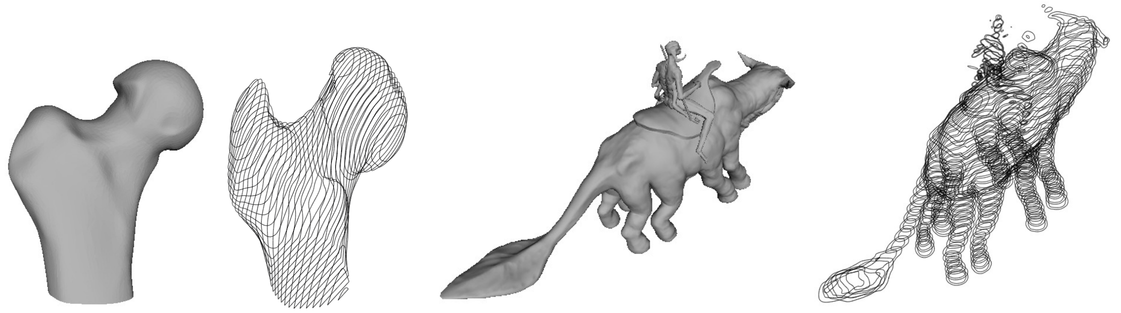


Figure 10: Samples of STL models (Femur and Soldier) and mesh slicing (layer thickness of 1.0 mm and 2.0 mm for illustration purposes).

Table 3: Running times (in seconds) for the four contour-construction algorithms (Trivial, Park, Slic3r, and Proposed) applied to the result models of Table 1.

| Model | Trivial | Park | Slic3r | Proposed |
|---|---:|---:|---:|---:|
| 01. Liver | **0.80** | 35.97 | 3.57 | 1.03 |
| 02. Femur | **0.38** | 16.59 | 2.00 | 0.50 |
| 03. Bunny | 1.23 | 22.00 | 8.51 | **0.65** |
| 04. Demon | 11.77 | 140.77 | 69.49 | **2.93** |
| 05. Sphenoid | 7.81 | 73.26 | 38.24 | **1.64** |
| 06. Rider | 2.03 | 27.82 | 25.02 | **0.61** |
| 07. Tesla | 23.16 | 238.59 | 213.04 | **4.76** |
| 08. Sphere | 63.79 | 407.57 | 405.08 | **8.63** |
| 09. Soldier | 46.69 | 275.14 | 176.06 | **5.09** |
| 10. Bear | 417.50 | 1,507.52 | 530.49 | **19.66** |
| 11. Warrior | 0.38 | 6.69 | 48.30 | **0.14** |
| 12. Robot | 14.85 | 75.05 | 251.59 | **1.50** |
| 13. Skull | 8,094.36 | — | 2,165.11 | **101.04** |

thousand segments per second, respectively. The Park procedure even ran out of memory before completing the large Skull model. We conjecture that the CGAL library has introduced excessive overhead.

The trivial procedure was also much faster than Slic3r, that typically processed about 50 thousand segments per second; except for the two largest models, Bear (when they almost tied) and Skull (when Slic3r was about 4 times faster).

## *5.3. Total running time*

Table 4 shows the combined running times of the slicing and contour construction steps of (1) our implementation of Park's algorithm, (2) the Slic3r software, (3) and the imple-

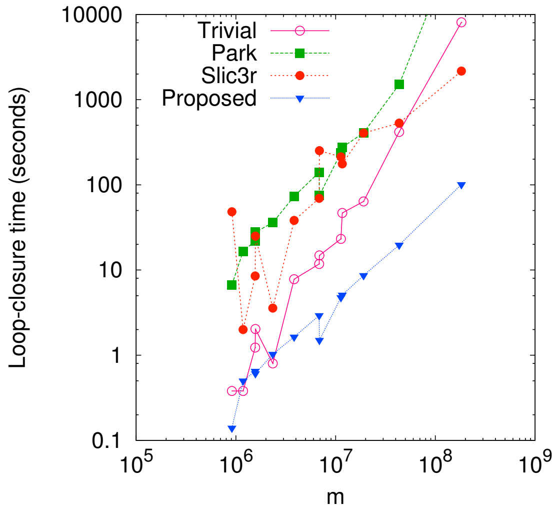


Figure 11: Running times of the four chaining implementations (Trivial, Park, Slic3r, and Proposed) for the result of uniform slicing of the STL models of Table 1, as a function of the number of segments $m$.

mentation of the proposed optimal algorithms INCREMENTAL-SLICING and CONTOUR-CONSTRUCTION. We executed the algorithms on the models of Table 1, sliced with uniformly spaced planes.

Table 4: Total running times (slicing and contour-construction steps), in seconds, for Park's algorithm, the Slic3r software, and the proposed approach, for the models of Table 1, with uniform slicing. The boldface values are the minimum running time in each row.

| Model | Park | Slic3r | Proposed |
|---|---:|---:|---:|
| 01. Liver | 37.25 | 3.89 | **1.13** |
| 02. Femur | 17.12 | 2.16 | **0.55** |
| 03. Bunny | 24.70 | 8.80 | **0.78** |
| 04. Demon | 160.89 | 70.77 | **3.46** |
| 05. Sphenoid | 80.99 | 38.85 | **1.93** |
| 06. Rider | 34.19 | 25.56 | **0.94** |
| 07. Tesla | 277.71 | 214.64 | **5.32** |
| 08. Sphere | 499.27 | 407.83 | **9.60** |
| 09. Soldier | 320.59 | 178.30 | **6.37** |
| 10. Bear | 1603.23 | 536.56 | **22.15** |
| 11. Warrior | 19.90 | 48.84 | **0.78** |
| 12. Robot | 135.16 | 253.18 | **2.75** |
| 13. Skull | — | 2189.65 | **118.46** |

As the table shows, our algorithm was faster than the other two for all models, sometimes by a large factor. For example, Slic3r took almost 36 minutes to process the Skull model, while our program did it in less than 2 minutes. For most models, and all three programs, we observed that the contour construction phase took 80–90% of the total running time. We note that the discrepancy in the running time for the Warrior and Robot models are due to the large fraction of small triangles that are entirely contained in the space between successive slicing planes. Figure 12 shows the total running time of our algorithm relative to those of Park's algorithm and of Slic3r, for each of those models.

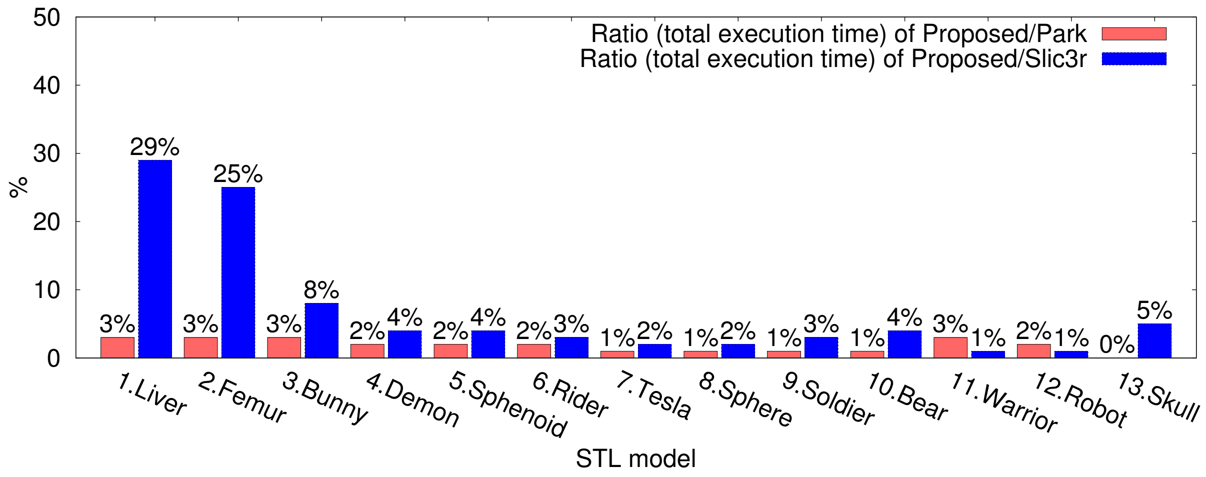


Figure 12: Total running time of our algorithm, expressed as percentages, relative to the running times of Park's algorithm and of Slic3r, for each of the models of Table 1 with uniform slicing.

## 6. Conclusions

This paper describes novel algorithms for the triangle mesh slicing problem and the contour construction problem. The proposed slicing algorithm uses a sweeping plane strategy highly simplified and optimized for unstructured triangle sets. Its time complexity is proportional to $\mathcal{O}(n \log k + k + m)$, where $n$ is the number of triangles, $k$ is the number of slicing planes, and $m$ is the number of triangle-plane intersections segments for arbitrary layer thickness. The time complexity reduces to $\mathcal{O}(n + k + m)$ if the planes are uniformly spaced or the triangles of the mesh are given in the proper order. We proved that this time complexity is optimal in the asymptotic output-sensitive sense. For the contour construction problem we develop an optimal (linear time) algorithm by using a hash-based strategy. The algorithms were compared both theoretically and experimentally against known algorithms and were evaluated with several representative STL files. Specially, when considering a large number of triangles, planes and intersections (models with large $z$-height), a remarkable improvement in execution time was achieved in relation to other algorithms from the literature. This is especially relevant in process planning for areas such as medicine where meshes have a large number of triangles.

## 7. Acknowledgements

This work was partially supported by the National Council for Scientific and Technological Development (CNPq), Grant 444789/2014-6 and Fundação Araucária. The basis for the trivial slicing algorithm was given by Raveh Gonen.

## References

[1] C. K. Chua, K. F. Leong, C. S. Lim, Rapid Prototyping: Principles and Applications, 3rd Edition, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2010.

[2] I. Gibson, D. W. Rosen, B. Stucker, Additive Manufacturing Technologies: Rapid Prototyping to Direct Digital Manufacturing, 1st Edition, Springer US, Boston, MA, 2010.

[3] P. Kulkarni, A. Marsan, D. Dutta, A review of process planning techniques in layered manufacturing, Rapid Prototyping Journal 6 (1) (2000) 18–35.

[4] C. Kirschman, C. Jara-Almonte, A parallel slicing algorithm for solid freeform fabrication processes, in: Solid Freeform Fabrication Symposium, 1992, pp. 26–33.

[5] S. McMains, C. Séquin, A coherent sweep plane slicer for layered manufacturing, in: ACM symposium on Solid modeling and applications - (SMA), ACM Press, New York, New York, USA, 1999, pp. 285–295.

[6] P. M. Pandey, N. V. Reddy, S. G. Dhande, Slicing procedures in layered manufacturing: a review, Rapid Prototyping Journal 9 (5) (2003) 274–288.

[7] W. Ma, W.-C. But, P. He, NURBS-based adaptive slicing for efficient rapid prototyping, Computer-Aided Design (CAD) - Elsevier 36 (13) (2004) 1309 – 1325.

[8] L. Zeng, L. M.-L. Lai, D. Qi, Y.-H. Lai, M. M.-F. Yuen, Efficient slicing procedure based on adaptive layer depth normal image, Computer-Aided Design (CAD) - Elsevier 43 (12) (2011) 1577–1586.

[9] K. Chalasani, B. Grogan, An algorithm to slice 3D shapes for reconstruction in prototyping systems, in: ASME Computers in Engineering Conference, 1991, pp. 209–216.

[10] S. Choi, K. Kwok, A tolerant slicing algorithm for layered manufacturing, Rapid Prototyping Journal 8 (3) (2002) 161–179.

[11] M. Vatani, A. Rahimi, F. Brazandeh, An enhanced slicing algorithm using nearest distance analysis for layer manufacturing, Proceeding of World Academy of Science, Engineering and Technology (25) (2009) 721–726.

[12] K. Tata, G. Fadel, A. Bagchi, N. Aziz, Efficient slicing for layered manufacturing, Rapid Prototyping Journal 4 (4) (1998) 151–167.

[13] E. Béchet, J.-C. Cuilliere, F. Trochu, Generation of a finite element MESH from stereolithography (STL) files, Computer-Aided Design (CAD) 34 (1) (2002) 1–17.

[14] X. Huang, Y. Yao, Q. Hu, Research on the Rapid Slicing Algorithm for NC Milling Based on STL Model, in: Asia Simulation Conference (AsiaSim), 2012, pp. 263–271.

[15] S. C. Park, Tool-path generation for z-constant contour machining, Computer-Aided Design (CAD) - Elsevier 35 (1) (2003) 27 – 36.

[16] J. Bentley, T. Ottmann, Algorithms for reporting and counting geometric intersections, IEEE Transactions on Computers C-28 (9) (1979) 643–647.

[17] D. Qi, L. Zeng, M. M. F. Yuen, Robust Slicing Procedure based on Surfel-Grid, Computer-Aided Design and Applications 10 (6) (2013) 965–981. doi:10.3722/cadaps.2013.965-981.

[18] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, 2nd Edition, McGraw-Hill Higher Education, 2001.

[19] N. Volpato, A. Franzoni, D. Luvizon, J. Schramm, Identifying the directions of a set of 2d contours for additive manufacturing process planning, The International Journal of Advanced Manufacturing Technology 68 (1-4) (2013) 33–43.

[20] An optimal algorithm for 3D triangle mesh slicing and loop-closure. Author: Rodrigo Minetto, `http://www.dainf.ct.utfpr.edu.br/%7erminetto/projects/slicing/`, 2015.

[21] ASCII Stereolithography Files, `http://people.sc.fsu.edu/~jburkardt/data/stla/stla.html`, accessed: April, 2015.

[22] Turbosquid 3D Models, `http://www.turbosquid.com`, accessed: February, 2015.

[23] CG Trader, `http://www.cgtrader.com`, accessed: April, 2015.

[24] CGAL: The Computational Geometry Algorithms Library, `http://www.cgal.org/`, accessed: August, 2015.

[25] Slic3r: G-code generators for 3D printers, `http://slic3r.org/`, accessed: August, 2015.

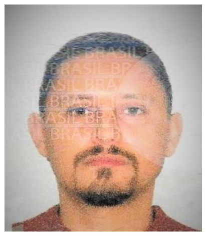
## 8. Vitae

**Rodrigo Minetto** works mainly in image processing and computer vision. He received the Ph.D. degree in computer science in 2012 from University of Campinas, Brazil and Université Pierre et Marie Curie, France. Since 2012 he is an assistant professor at Federal University of Technology - Paraná (UTFPR) - Brazil.

**Neri Volpato** is a senior lecturer at the Federal University of Technology - Paraná (UTFPR) - Brazil, and works in the area of Additive Manufacturing, CAD/CAM and CNC machining. He got his Ph.D. in 2001 at the University of Leeds, UK, from the department of Mechanical Engineering.

**Jorge Stolfi** is a full professor at University of Campinas (UNICAMP), Brazil. He received the Ph.D. degree in computer science from University of Stanford, USA, in 1989. He also worked as a research engineer at the Digital Systems Research Center from 1989 to 1992. His research interests include natural language processing, computational geometry, computer graphics, numerical analysis, image processing, and applications.

**Rodrigo Gregori** received the M.Sc. degree in computer science in 2013 from Federal University of Technology - Paraná (UTFPR) - Brazil. His research interests are computational geometry, algorithms and additive manufacturing.



**Murilo da Silva** works mainly in algorithms and complexity. He obtained his Ph.D. in computer science in 2008 at the University of Leeds, UK. Since 2010 is an assistant professor at Federal University of Technology - Paraná (UTFPR) - Brazil. Currently he is a visiting scholar at Simon Fraser University, Canada.