An improved slicing algorithm with efficient contour construction using STL files

Article *in* International Journal of Advanced Manufacturing Technology · September 2015

DOI: 10.1007/s00170-015-7071-9

CITATIONS
19

READS
550

2 authors:

Zhengyan Zhang
Hebei University of Technology
25 PUBLICATIONS **100** CITATIONS
SEE PROFILE

Sanjay Joshi
Pennsylvania State University
105 PUBLICATIONS **3,173** CITATIONS
SEE PROFILE

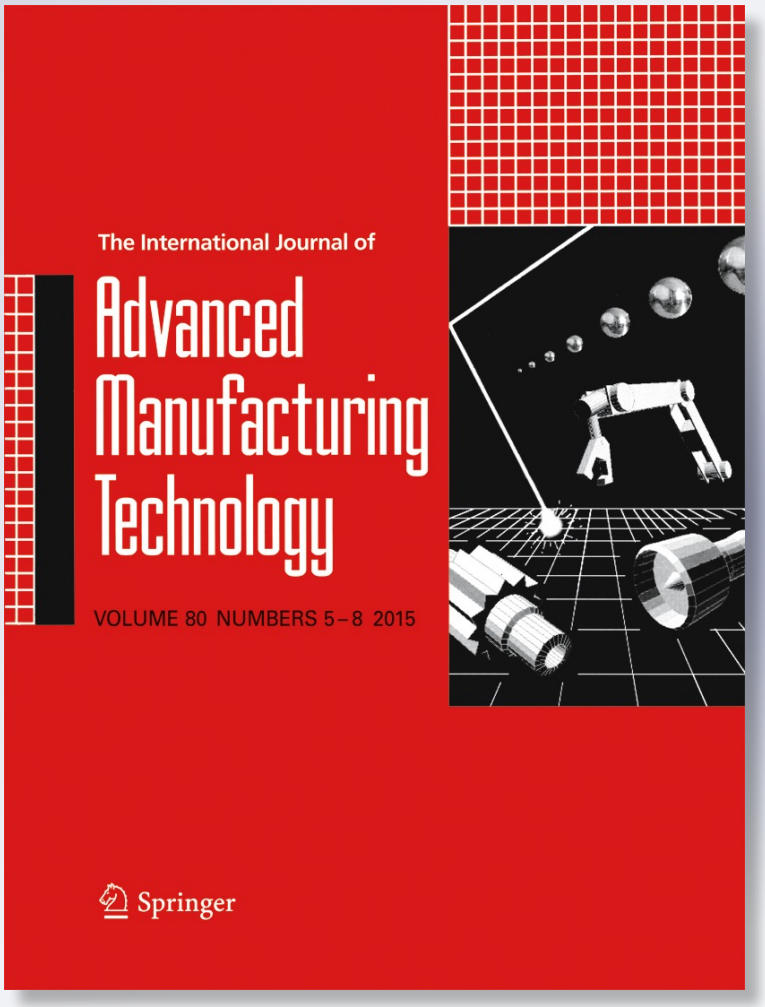
An improved slicing algorithm with efficient contour construction using STL files

Zhengyan Zhang & Sanjay Joshi

The International Journal of Advanced Manufacturing Technology

ISSN 0268-3768
Volume 80
Combined 5-8

Int J Adv Manuf Technol (2015)
80:1347-1362
DOI 10.1007/s00170-015-7071-9



Springer

Int J Adv Manuf Technol (2015) 80:1347–1362
DOI 10.1007/s00170-015-7071-9

ORIGINAL ARTICLE

An improved slicing algorithm with efficient contour construction using STL files

Zhengyan Zhang[1] · **Sanjay Joshi**[2]

Received: 5 October 2014 / Accepted: 19 March 2015 / Published online: 19 April 2015

Abstract Slicing is an important step for all layer-based additive manufacturing (AM) processes. This paper proposes an improved and robust slicing algorithm with efficient contour construction to reduce the slicing time and be able to slice a complex STereoLithography (STL) model with millions of triangles (resulting from high-accuracy STL files). Also, another important feature of the proposed method is it can identify outer and inner contours automatically. Test results to demonstrate the improvements in execution time and comparisons with results from published papers have been given to illustrate the algorithm efficiency. The robustness of this slicing algorithm is demonstrated by several complex models with large numbers of nested contours on each slice.

Keywords Slicing algorithm · Efficient contour construction · Efficiency · Robustness

1 Introduction

Additive manufacturing (AM) is used to directly fabricate parts with higher complex geometry by building them in layers. In most AM process, a solid 3D model is first created, converted to a STereoLithography (STL) file format, and then sliced into a series of layers for output to an AM machine. The slicing procedure is an important element of the AM process steps, and the sliced contour is used in the generation of tool paths, for the layer-by-layer deposition required by the manufacturing process. The STL model is obtained by tessellating the solid model surface into triangles. The accuracy of the tessellation is controlled by the chordal error, which is the maximum shape difference between the tessellated model and the original solid model. The number of triangles needed to tessellate the solid model is a function of the acceptable chordal error and complexity of part geometry. The larger number of triangles generated during the STL file creation process can have an impact on the time required for slicing the model and subsequent creation of the contour for each slice. The number of slices required for fabrication is typically derived from the layer thickness and part build orientation as specified by the user. The triangle facets defined in the STL file have to be sliced by each of the slices, and contours must be constructed for each slice; hence, the efficiency of the algorithm plays a huge role in the time required for slicing and contour creation.

In this paper, an improved slicing algorithm with efficient contour construction is proposed to reduce the slicing time. The proposed method has several important features, such as efficiency, robustness, and the ability to identify outer and inner contours automatically. The remaining sections in this paper are organized as follows: related researches on the slicing technique are introduced in Section 2. Section 3 provides a basic overview of our proposed approach to help in understanding our algorithm. The detail of our main algorithm and the efficient contour construction algorithm can be found in Section 4. To illustrate the efficiency of the algorithm, slicing results for complex STL models with large number of triangles and comparisons to models from other published papers are presented in Section 5. Finally, Section 6 presents the conclusions.

✉ Zhengyan Zhang
zhengyanzhang09@gmail.com

[1] School of Mechanical Engineering, Hebei University of Technology, Tianjin 300132, China

[2] Industrial and Manufacturing Engineering, Pennsylvania State University, University Park, State College, PA 16802, USA

2 Related researches

The STL model is the common format adopted by most commercial software for AM input. The STL model represents the boundary of the object as triangles, and these triangles are stored in the STL file as an unordered list of triangles defined by their vertices and a surface normal. Standard convention using the right-hand rule is used to orient the surface normal so that it is pointing towards the outside.

Various methods and algorithms have been proposed to slice the STL files and generate the slice contours required to define each slice. Uniform slicing, that is the popular slicing, generates constant layer thickness slices. Kirschman et al. [1] introduced the simplest slicing algorithm for STL file. The algorithm intersects all triangles with each z plane and connects the resulting line segments into closed polygons for each slice at one time from the bottom to the top of the STL model. Algorithms for uniform slicing in the worst case require intersecting each slicing plane with each triangle. This can be quite time consuming and wasteful since every triangle may not need to be sliced by each plane. Sorting of triangles is a common strategy used to speed up the algorithms. The details of these methods can be found in references [2–5]. Tata et al. [6] proposed a new approach based on grouping to sort and group facets from the input STL file before the slicing procedure. In their method, a facet processor is employed to arrange the triangular facets in the STL file into groups based on their location with respect to the base of the model. To further reduce the slicing time, topological information construction for the STL file before the slicing procedure is adopted by reference [7, 8]. Although these methods in reference [7, 8] can enhance the slicing algorithm and reduce the slicing time, the arrangement of all triangles or topological information construction for the STL file before the slicing procedure is still time consuming.

Adaptive slicing is a variant of the uniform slicing, where the spacing between the slices is not constant but determined by the geometry and machine capability. The advantage of adaptive slicing is reduced build times and improvements in surface finish. This method has been introduced in references [9–16]. Hybrid slicing, a combination of direct and adaptive slicing, is presented in references [3, 10, 17]. Y.-S. Liao et al. [18] develop a new slice algorithm for improving manufacturing efficiency and working tolerances based on the existing slice algorithm. Emmanuel et al. [12] present an adaptive slicing method to reslice each slab uniformly until the layers thickness is under the desired surface accuracy after STL CAD model has been sliced uniformly into slabs. While these methods may reduce the number of slices, these methods still rely on the intersections of the triangles with the slicing planes and construction of the contour geometry.

Slicing a polyhedral model is performed by intersecting the facets with horizontal slicing planes; this can result in some singularity problems that were introduced by Hyung-Jung Kim et al. in reference [19]. Similar problems can also be found in reference [20, 21]. For handling these problems, Hyung-Jung Kim et al. [19] propose a practical slicing algorithm for removing singularity cases and reducing the computational load by shifting vertices. Kim's slicing algorithm employs a pre-processing step to avoid the intersection when these singularity cases occur. In their pre-processing step, all facets in the STL model are scanned and shifted in a positive or negative z direction by a very small value if a facet, edge, or vertex lies on the slicing plane.

Zeng et al. [22] and Qi, Di et al. [23] propose an efficient algorithm based on adaptive Layer Depth Normal Image (LDNI) to achieve a compromise between slicing accuracy and time for complex constructive solid geometry (CSG) models. Similar work was also done by Chiu et al. [24] and Zhu et al. [25], where each CSG primitive is represented in a ray representation (dexel) and Boolean operation is simplified from 3D to 1D for the dexel model. The input to these is not an STL file of the final object but rather a polygonal tessellated file of each CSG primitive.

3 Basic overview of our proposed approach

3.1 Preliminary definitions

For the convenience of describing the detail of our proposed slicing algorithm, two preliminary definitions are introduced in this section, and these definitions will be used throughout this paper.

Definition 1 Triangle vertices flag

For an arbitrary triangle, its three vertices are sorted counterclockwise in the STL file. Generally, each triangle has two edges intersecting with a given slicing plane, and we denote one of them as the forward edge and only use that edge to compute the intersection. To judge which edge is the forward edge, vertex flag $\text{flag}_i (i=0,1,2)$ is given to each vertex sequentially as the triangle data is processed from the STL file.

See Fig. 1a, $\overrightarrow{N}$ is the normal vector of the triangle, $v_i (i=0, 1,2)$ are the three vertices for the triangle sorted in sequence in the STL file. For the first vertex, v_0, flag_0 is equal to 0; for the second vertex, v_1, flag_1 is equal to 1; and flag_2 is equal to 2 for

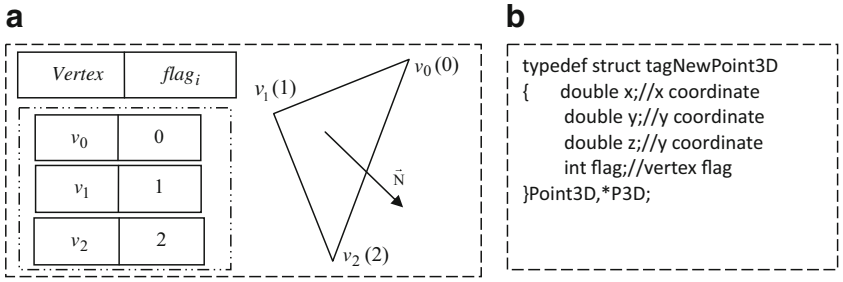


Fig. 1 Triangle and its vertices: (**a**) vertex flag and (**b**)vertex data structure

the last vertex v_2. Figure 1b shows its corresponding data structure.

Definition 2 Triangle edges

See Fig. 2, $\vec{N}$ is the normal vector of the triangle. For an arbitrary triangle which does not lie on the slicing plane, its three edges $s_i(i=1,2,3)$ can be expressed by $s_1=\{v_{zmin}, v_{zmax}\}$, $s_2=\{v_{zmin}, v_{zmed}\}$, $s_3=\{v_{zmed}, v_{zmax}\}$, where, v_{zmax}, v_{zmed}, and v_{zmin} are the three vertices and their z coordinates satisfy $v_{zmax}\geq v_{zmed}\geq v_{zmin}$ (assume z direction is the slicing direction).

3.2 Our approach

The basic approaches for slicing the 3D model can be classified into two different types:

1) Given a slicing plane, search for intersecting triangles.
2) Given a triangle, search for intersecting slicing planes.

The first method, often referred to as the scanning line/plane algorithm, is the common method used for slicing. In the first method, topological information construction [7, 8] and grouping techniques [1] before the slicing procedure have been proposed in order to reduce the slicing time.

The proposed slicing approach is based on the second method. Without any loss of generality, we can assume that the object is oriented such that the slicing places are parallel to the z axis. Since the slicing planes are at discrete intervals, the planes that intersect each triangle can be easily determined by the z coordinates of the triangle and the z coordinates of the planes.

Intersection of the triangle with the slice planes results in the creation of a line segment which will be part of the contour defining the slice geometry at each layer. An important point to note is that for each line segment generated by the intersection between the plane and triangle, only two adjacent edges of the triangle will contribute vertices to the line segment. In fact, one of the edges will always be the edge with the vertices having maximum and minimum z values. The other edge will be one of the two other edges of the triangle. Furthermore, for triangles sharing these edges' duplicate intersection points will be generated by intersecting each triangle as a separate entity. The proposed algorithm seeks to eliminate this duplication by

(i) Exploiting properties of the STL representation

Although the STL file consists of an unordered list of triangles, the three vertices of the triangle are listed in a counter clockwise manner.

(ii) Only computing intersections for one edge (called forward edge) of the triangle and tagging the second edge that would contribute an intersection point.

The process for determining which of the edges to use to compute the intersection point is described in the procedure to detect the forward and backward edges (Section 4.3).

The contour reconstruction algorithm maintains a list of the points that are part of a contour on a given plane along with pointers to the edges, and maintenance of the contour list for each plane requires an insertion procedure to determine the corresponding position of a new intersection point (generated by the triangle intersection) in the contour boundary. An efficient contour construction (ECC) algorithm is developed to search the corresponding position in contour for each intersection between a given triangle and slicing plane. The details can be found in Section 4.4.

The flow chart for the algorithm is shown in Fig. 3.

To illustrate the overall approach, consider the following example showing three triangles and one plane of slicing (see Fig. 4). Let us assume that the first triangle read from the STL file is triangle 1 (v_1,v_2,v_3). Denote s_1 as the edge with v_{zmin}, v_{zmax} as the two vertices. In this case, $s_1=\{v_2,v_3\}$. This edge will

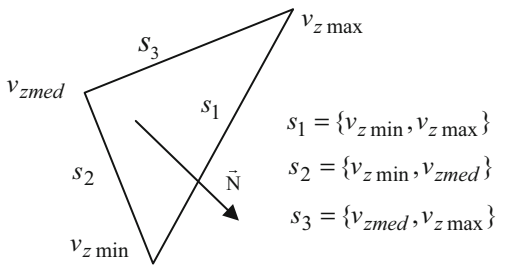


Fig. 2 Triangle vertices and triangle edges

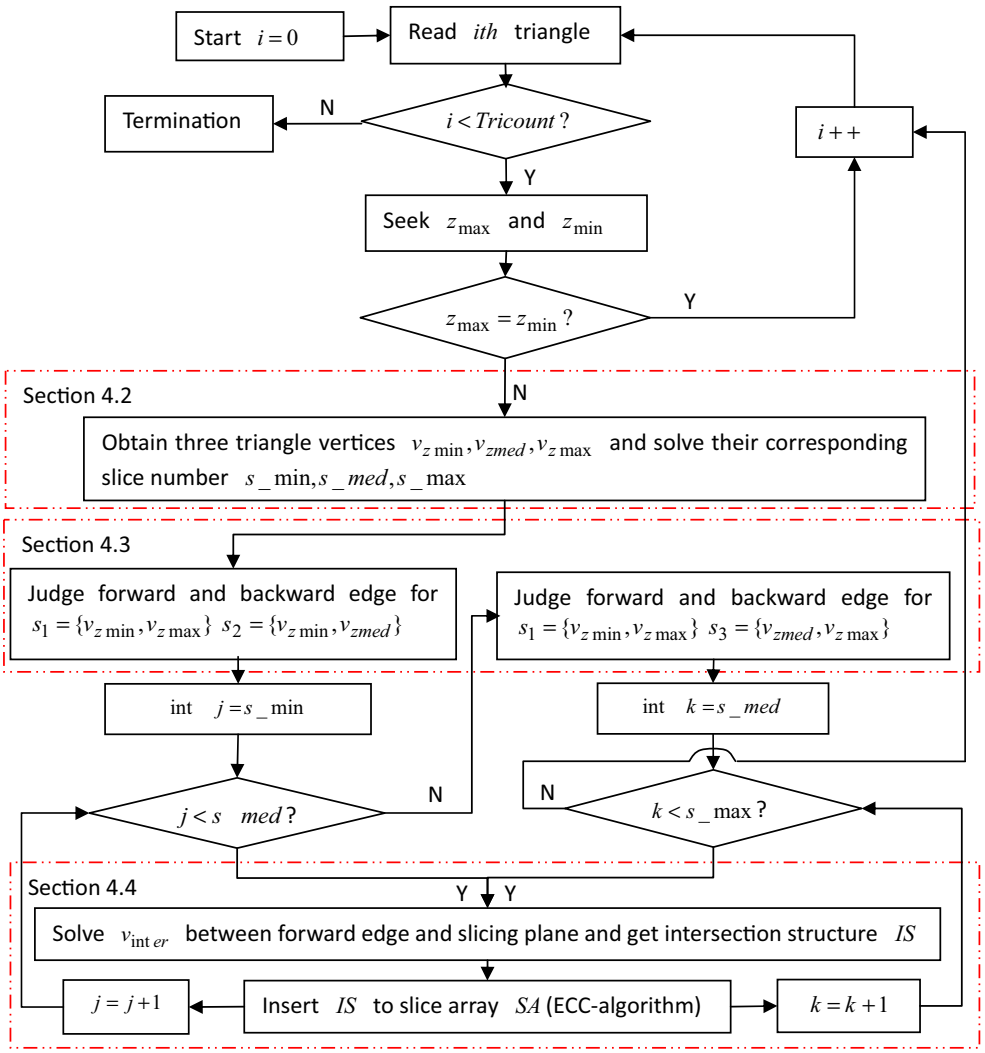


Fig. 3 The flow chart of our proposed slicing algorithm

always intersect with the slice plane. The other edge of intersection will be one of the other two edges of the triangle. Checking the z coordinate will eliminate one of the edges. Using the rules proposed in Section 4.3 to determine the forward edge, $s_2\{v_1, v_2\}$ is identified as the forward edge and only the intersection point (called p_1) is computed. This point will be part of the contour for this slice plane. Next, consider triangle 2. The first intersecting edge is identified as $s_1\{v_2, v_4\}$ and the second intersecting edge is identified as $s_2\{v_2, v_5\}$. Using the rules proposed in Section 4.3 to determine the forward edge, s_1 is identified as the forward edge, and the intersection point between it and the plane is computed (called p_2). Point p_2 is then inserted into the slice array list. Next, consider triangle 3, edge $s_2\{v_2, v_3\}$ is identified as the forward edge and point p_3 is computed as the intersection point. p_3 is then inserted into the slice array list using the ECC algorithm proposed in Section 4.4. At this stage, the sequence in which the points are maintained will automatically determine whether the contour being generated is an inner or outer contour.

It is worthwhile noting that the approach cuts the number of intersections generated by half and further improves robustness by not having to compare the start and end points of various line segments that would be typically generated using other slicing approaches. Further, the contour construction is performed simultaneously and the inner and outer contours

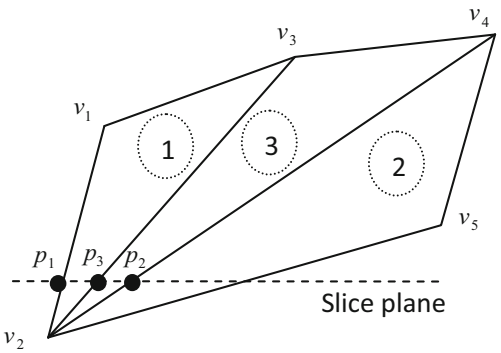


Fig. 4 An example schema to illustrate the overall approach

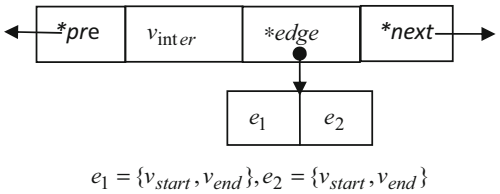


Fig. 5 Intersection structure

are easily identifiable by the opposite direction of traversal of the contours.

4 Slice procedure

4.1 Data structure

Before describing our slicing algorithm, the data structures used are provided.

4.1.1 Intersection structure (IS)

The intersection structure (IS) is used to store the information related to the intersection between the triangle and plane. Besides the intersection v_{inter} between the current triangle (CT) and the current slicing plane (CS), the IS contains other three elements: the edge pointer *edge pointing to intersecting edges e_1 and e_2 (call them forward edge and backward edge, respectively), the previous pointer *pre pointing to the previous IS, and the next pointer *next pointing to the next IS. The structure of IS is shown in Fig. 5.

For slicing the 3D model efficiently, several rules are established as follows.

Rule 1 The edge $s_1(s_1=\{v_{z\min}, v_{z\max}\})$ of the CT must be one of the two edges (e_1, e_2) in the IS.

Rule 2 The z coordinate of the starting vertex v_{start} must be less than or equal to the ending vertex v_{end} for edges e_1 and e_2. Here, we assume the z axis is the slicing direction.

Rule 3 The intersection v_{inter} must be the intersection between the forward edge e_1 and the current slicing plane CS. The backward edge e_2 must be ignored during the solving intersection process.

Rule 4 Assume s_1 and s_2 or s_1 and s_3 are the two edges which should be pointed by edge pointer *edge, which edge should replace e_1 and which edge should replace e_2 are determined by "forward edge and backward edge judgments rules" described in Section 4.3.

Here, $s_1=\{v_{z\min}, v_{z\max}\}$, $s_2=\{v_{z\min}, v_{z\text{med}}\}$, and $s_3=\{v_{z\text{med}}, v_{z\max}\}$.

By analyzing the above rules, there are four cases for edges e_1 and e_2 as follow.

Case 1 $e_1=s_2, e_2=s_1$; in this case, *edge points the structure $\{v_{z\min}, v_{z\text{med}}, v_{z\min}, v_{z\max}\}$.

Case 3 $e_1=s_3, e_2=s_1$; in this case, *edge points the structure $\{v_{z\text{med}}, v_{z\max}, v_{z\min}, v_{z\max}\}$.

Case 2 $e_1=s_1, e_2=s_2$; in this case, *edge points the structure $\{v_{z\min}, v_{z\max}, v_{z\min}, v_{z\text{med}}\}$.

Case 3 $e_1=s_1, e_2=s_3$; in this case, *edge points the structure $\{v_{z\min}, v_{z\max}, v_{z\text{med}}, v_{z\max}\}$.

For understanding the intersection structure, one example is shown in Fig. 6.

Assume $v_i(i=1,\cdots,n)$ are all intersections between edge s_2 and all intersecting slicing plane $CS_i(i=1,\cdots,n)$, and their corresponding intersection structures are $IS_i(i=1,\cdots,n)$, the example shown in Fig. 6 can also illustrate that all the pointer *edge in $IS_i(i=1,\cdots,n)$ point to the same two edges s_2 and s_1. The replacement rule can be described briefly as follows to help in understanding Fig. 6b. Facing the triangle whose normal vector points to a viewer (see Fig. 6a), the left intersecting edge is thought as forward edge e_1 and the right intersecting edge is thought as e_2.

4.1.2 Intersection linked list (ILL)

An intersection linked list (ILL) is used to link a series of intersection structures that would eventually form the contours on a slice. The structure of ILL is shown in Fig. 7.

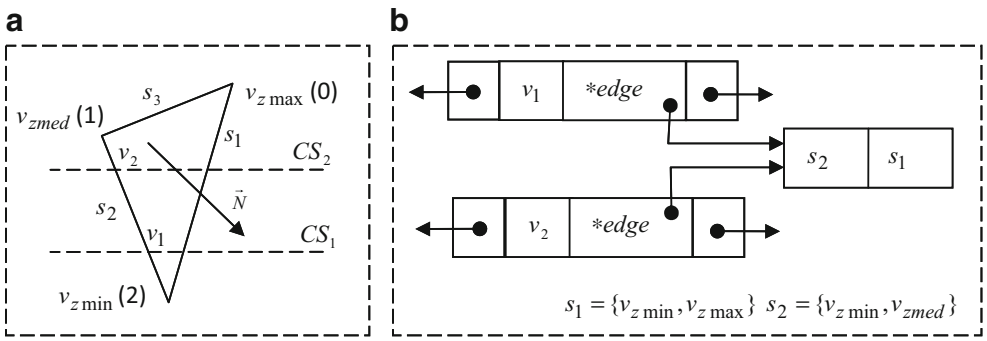


Fig. 6 Example: **a** CT intersects with CS and **b** the corresponding intersection structure

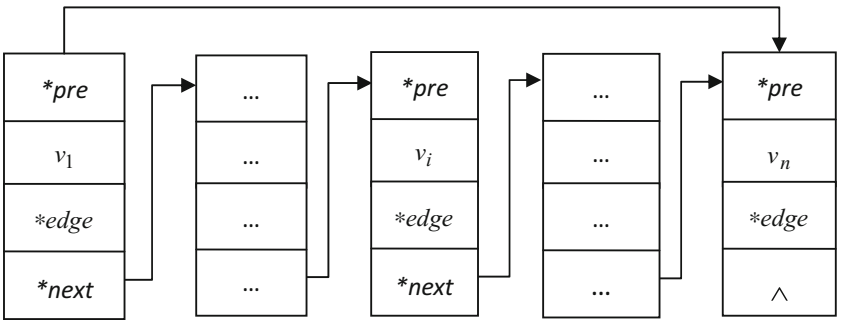


Fig. 7 Intersection linked list

The intersection linked list can be explained as follows.

(1) The previous pointer *pre of the first element in the ILL must keep pointing to the last element in the ILL in order to get the first and last elements immediately from the ILL.
(2) Except the first element in the ILL, the previous pointer *pre of the left elements in the ILL should be NULL.
(3) The next pointer *next of the last element in the ILL must be NULL.
(4) Except the last element in the ILL,the next pointer *next of the left elements in the ILL must be pointing to its next element.

4.1.3 Contour linked list (CLL)

A contour linked list (CLL) is used to link all intersection linked lists to create a contour. Any element in the CLL should include an ILL and the next pointer *next. Its structure is shown in Fig. 8. The contour linked list is a nested linked list.

4.1.4 Slice array (SA)

Slice array (SA) is used to store all slices for a sliced 3D model. A 1D array is used to store all slices, and each element in the SA stands for one slice. For example, all the slices $S_i(i=0,\cdots,n-1)$ will be sorted in slice array SA[i]($i=0,\cdots,n-1$), and the ith slice can be written as S_i=SA[i].

4.2 Global slicing algorithm

Before describing the global slicing algorithm, we first describe how to solve the slice number s_min,s_med,s_max corresponding to $v_{z\min},v_{z\mathrm{med}},v_{z\max}$.

As Fig. 9 shows, $v_{z\max},v_{z\mathrm{med}},v_{z\min}$ are the three vertices of the arbitrary triangle, and the slice number s_min,s_med,s_max can be solved by using the following formula.

$$s_j = \frac{v_j - s_{z\min}}{t}$$

where s_j stands for s_min,s_med,s_max; v_j stands for $v_{z\min}$, $v_{z\mathrm{med}},v_{z\max}$; $s_{z\min}$ denotes the z coordinate of the 0th slice which can be designed to coincide to the x-y plane of the system; and t denotes the slice thickness.

The global slicing algorithm is the main algorithm for slicing the 3D model and is introduced briefly with the "input" and "output" as follows.

Input: STL file of the 3D model
Output: SA

Assume z axis is the slicing direction, the flow chart for the global slicing algorithm is shown in Fig. 3 described in Section 3. And the slicing algorithm will be described step by step as follows.

Step 1 Initially, i=0, read the ith triangle from the STL file and give the triangle vertices flag to each vertex according to the vertex sequence. If i is less than the total number of triangles, continue to the next step. Otherwise, the program terminates.
Step 2 Search the maximum and minimum z coordinates ($z_{\max}$ and $z_{\min}$) of the ith triangle. If $z_{\max}=z_{\min}$, which means the ith triangle lies on the slicing plane, turn to step 8. Otherwise, continue to the next step.
Step 3 Obtain three vertices ($v_{z\min},v_{z\mathrm{med}},v_{z\max}$) of the ith triangle and solve the slice number s_min,s_med,s_max corresponding to $v_{z\min},v_{z\mathrm{med}}$, and $v_{z\max}$. Continue to the next step.
Step 4 Judge the forward edge and backward edge for s_1 and s_2 according to the judgment rules (described in

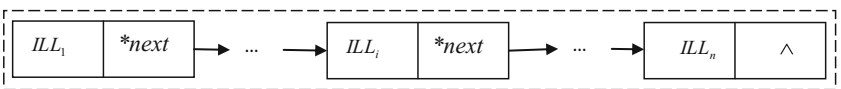


Fig. 8 Contour linked list

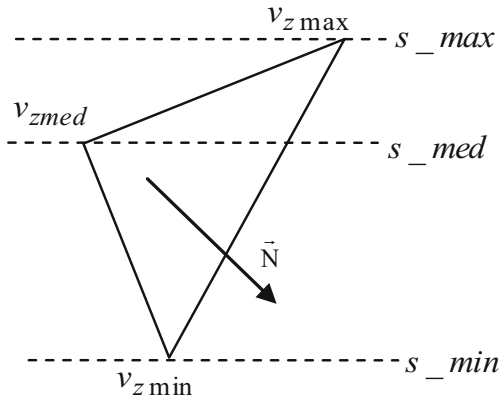


Fig. 9 The slice number corresponding to each vertex

Section 4.3) and then put s_1 and s_2 into the corresponding positions in the intersection structure. Continue to the next step.

Step 5 Initially, int $j = s\_\min$. If $j<s\_\text{med}$, execute the following steps from step 5.1 to step 5.3. Otherwise, turn to step 6.

- Step 5.1 Solve the intersection v_{inter} between the forward edge e_1 and the jth slicing plane and then add v_{inter} into the intersection structure. Set the *pre and *next to NULL. Continue to the next step.
- Step 5.2 Insert the IS into the to slice array SA[j] according to the ECC algorithm described in Section 4.4. Continue to the next step.
- Step 5.3 Let $j=j+1$; turn to step 5.

Step 6 Judge the forward edge and backward edge for s_1 and s_3 according to the judgment rules (described in Section 4.3) and then put s_1 and s_3 into the corresponding positions in the intersection structure. Continue to the next step.

Step 7 Initially, int $k = s\_\text{med}$. If $k<s\_\max$, execute the following steps from step 7.1 to step 7.3. Otherwise, turn to step 8.

- Step 7.1 Solve the intersection v_{inter} between the forward edge e_1 and the kth slicing plane and then add v_{inter} into the intersection structure. Set the *pre and *next to NULL. Continue to the next step.

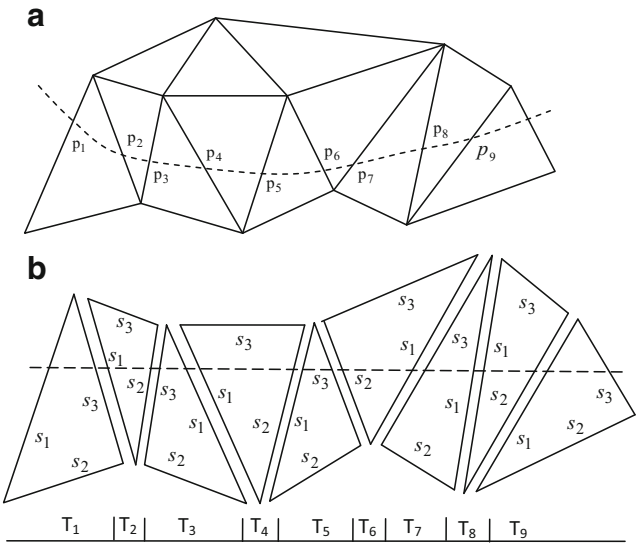


Fig. 11 Our approach schema. **a** The 3D object intersects with the given slicing plane. **b** All intersecting triangles

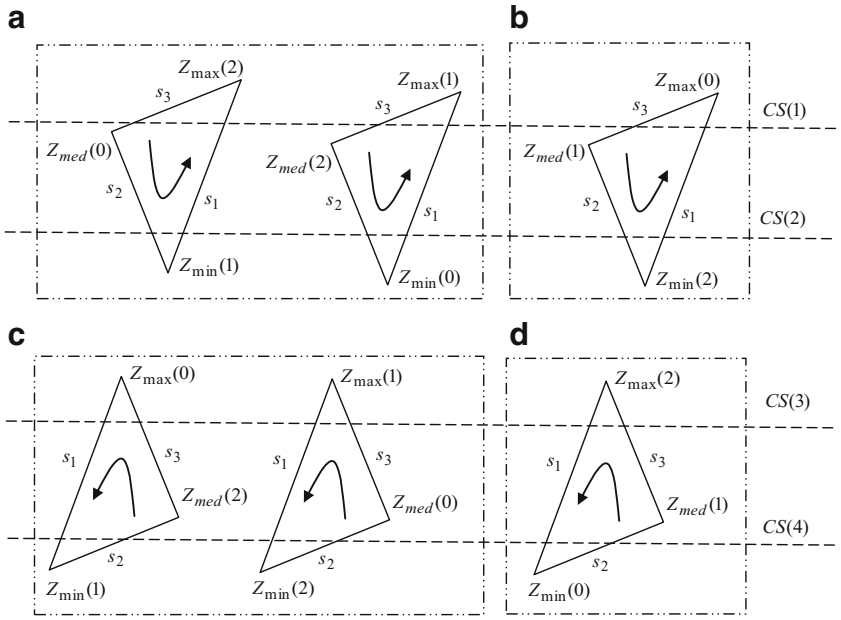


Fig. 10 Forward edge and backward edge judgment cases. **a** group 1, **b** group 2, **c** group 3, and **d** group 4

Fig. 12 Intersection structure information for $T_i(i=1,\cdots,9)$ in Fig. 11b

Triangle	Forward edge	Backward edge	Intersection	*IS*	**edge*
T_1	s_1	s_3	p_1	*pre,p_1,*edge,*next	s_1, s_3
T_2	s_1	s_2	P_2	*pre,p_2,*edge,*next	s_1, s_2
T_3	s_3	s_1	p_3	*pre,p_3,*edge,*next	s_3, s_1
T_4	s_1	s_2	P_4	*pre,p_4,*edge,*nex	s_1, s_2
T_5	s_1	s_3	P_5	*pre,p_5,*edge,*next	s_1, s_3
T_6	s_2	s_1	P_6	*pre,p_6,*edge,*next	s_2, s_1
T_7	s_3	s_1	P_7	*pre,p_7,*edge,*next	s_3, s_1
T_8	s_1	s_2	P_8	*pre,p_8,*edge,*next	s_1, s_2
T_9	s_1	s_3	P_9	*pre,p_9,*edge,*next	s_1, s_3

Step 7.2 Insert the IS into the to slice array SA[k]. Continue to the next step.

Step 7.3 Let $k=k+1$; turn to step 7.

Step 8 Set $i=i+1$; turn to step 1.

4.3 Forward edge and backward edge judgments (FBEJ-algorithm)

When the current triangle CT intersects with the current slicing plane CS, generally, there are two intersections because there exist two intersecting edges (expressed by e_1 and e_2 shown in the intersection structure) with the current slicing plane. If the two intersections are solved and stored in the intersection list for all intersecting triangles with the current slicing plane, there will be redundant intersections in the intersection list. To avoid redundant intersections, we only solve the intersection between the forward edge e_1 in the intersection structure and the current slicing plane and ignore the backward edge e_2 in the intersection structure. For realizing contour construction successfully after solving all the intersections for all intersecting triangles, how to judge which edge among the three triangle edges s_1,s_2, and s_3 should replace e_1 and e_2 in the intersection structure is an important element of the algorithm.

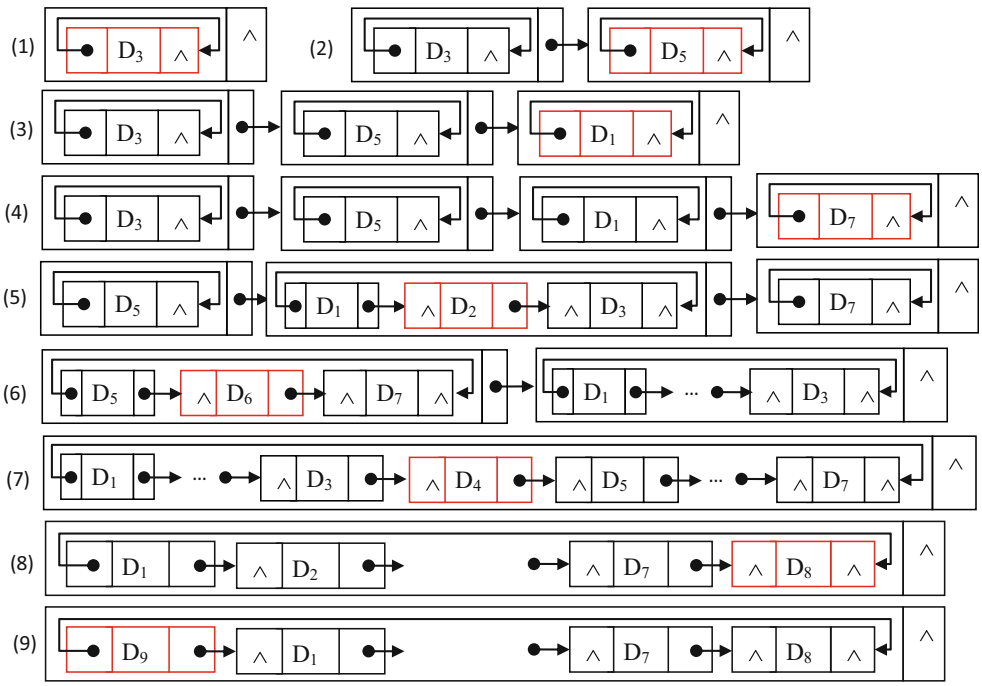


Fig. 13 Efficient contour construction procedure for the example in Fig. 11b

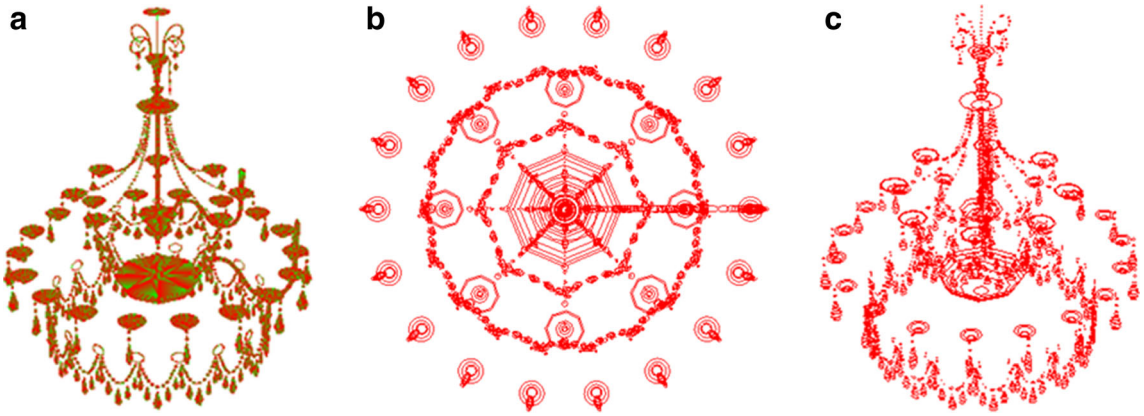


Fig. 14 "Chandelier" model (82.24×81.09×95.82) with 297,916 triangles. **a** Original 3D model, **b** Sliced model—top view, and **c** Sliced model—3D view

In the STL file, the normal vector for each triangle points to the object outside. The vertices of the triangles are not ordered in any manner other than being stored in a counterclockwise manner. When read from the file, they are sequentially assigned the triangle vertices flag (0, 1, 2). The edges are designated s_1,s_2, and s_3 based on definition 2 presented in Section 3. The two intersecting edges with the given slicing plane must be s_1 and s_2 or s_1 and s_3 shown in Fig. 10.

Based on the two possible orientations of s_1 and three possible sequences of reading vertices and flag assignments, six possible cases exist for determining which edge is the forward edge e_1 and which one is the backward edge e_2 in the intersection structure. All the cases in Fig. 10 can be classified into four different groups based on the orientation of s_1 and the vertex flag as follows.

- Group 1: s_1 oriented from z_{min} to z_{max} and $flag_{zmax}$>$flag_{zmin}$; then, e_1=s_2 (or s_3) and e_2=s_1.
- Group 2: s_1 oriented from z_{min} to z_{max} and $flag_{zmax}$<$flag_{zmin}$; then, e_1=s_2 (or s_3) and e_2=s_1.
- Group 3: s_1 oriented from z_{max} to z_{min} and $flag_{zmax}$<$flag_{zmin}$; then, e_1=s_1 and e_2=s_2 (or s_3).
- Group 4: s_1 oriented from z_{max} to z_{min} and $flag_{zmax}$>$flag_{zmin}$; then, e_1=s_1 and e_2=s_2 (or s_3).

The choice of s_2 (or s_3) is determined by the slice plane.

To help in understanding the FBEJ algorithm, taking Fig. 11 as an example, the forward and backward edges are shown in Fig. 12.

$T_i(i=1,\cdots,9)$ in Fig. 11b are the intersecting triangles with the given slicing plane from left to right. Their corresponding forward edges (effective intersecting edge) are shown in Fig. 12. $p_i(i=1,\cdots,9)$ in Figs. 11a and 12 are the intersections between $T_i(i=1,\cdots,9)$ and the given slicing plane. The intersection structure information for $T_i(i=1,\cdots,9)$ are also shown in Fig. 12.

4.4 Efficient contour construction (ECC algorithm)

Contour construction plays an important role in the complete slicing algorithm. Since the triangles are stored randomly and unordered in the STL file, the intersections solved on the current slicing for each triangle are also unordered. Constructing the contour from these unordered intersections can have a significant impact on the slicing process.

Typical contour construction where the intersection of each triangle with a plane results in a line and both end points of the line generated are stored requires that the vertices of the line segments be matched to create the contour. This creates a

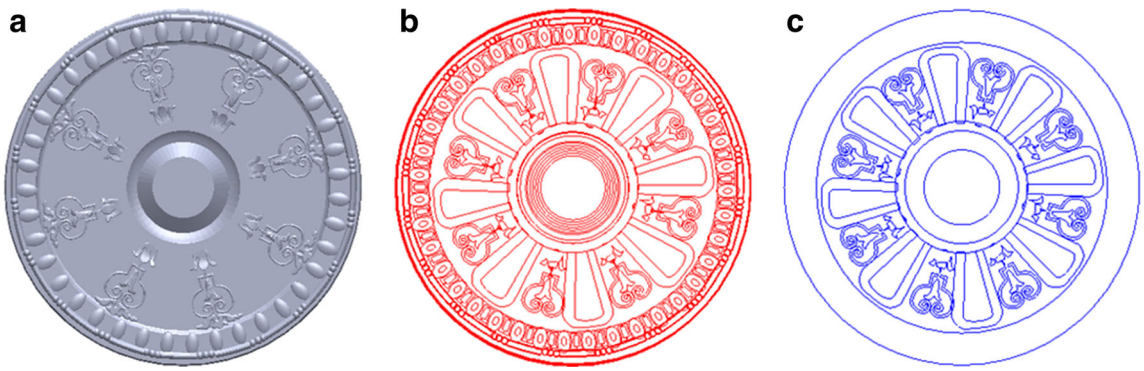


Fig. 15 "Rosette" model (18.16×18.26×1.01) with 180,558 triangles. **a** Original 3D model, **b** Sliced model—top view, and **c** Sliced model—third layer

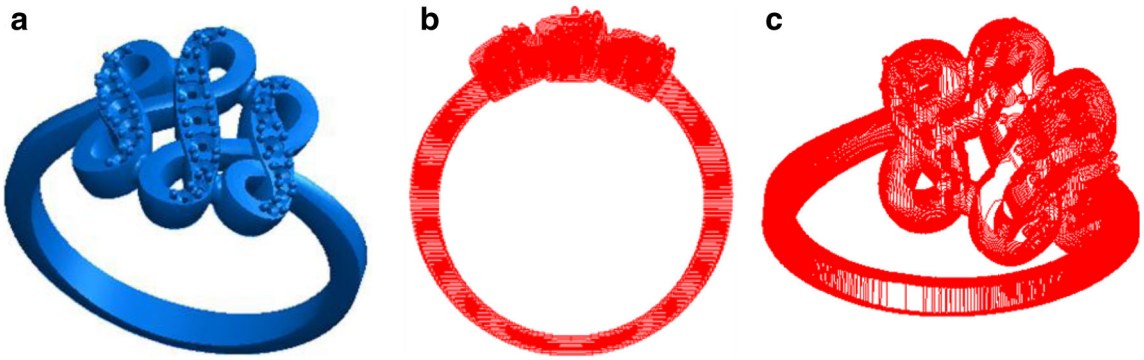


Fig. 16 Ring model (22.58×14.33×23.15) with 33,730 triangles. **a** Original 3D model, **b** Sliced model—front view, and **c** Sliced model—3D view

problem with the robustness of the algorithm given that the end points generated via two different triangles may not be exactly identical (floating point error). In the proposed algorithm, since only one intersection is computed for each triangle, vertex matching is not required. The contour is constructed by inserting each intersection into its position as it is generated by checking for the insertion position. The algorithm requires checking the first and last elements in the intersection linked list for each inserted intersection.

Assume the inserted intersection structure is IS and the intersection linked list is ILL whose first element is $\mathrm{ILL.IS_f}$ and last element is $\mathrm{ILL.IS_l}$, the checking and inserting rules are defined as follows.

Rule 1 If $\mathrm{IS}.e_2=\mathrm{ILL.IS_f}.e_1$, it means IS is the forward adjacent intersection of $\mathrm{ILL.IS_f}$; thus, insert IS into the front of $\mathrm{ILL.IS_f}$. In this case, IS cannot be the backward adjacent intersection of $\mathrm{ILL.IS_f}$; thus, only check whether $\mathrm{IS}.e_2$ and $\mathrm{ILL.IS_f}.e_1$ are the same edges and ignore $\mathrm{IS}.e_1$ and $\mathrm{ILL.IS_f}.e_2$.

Rule 2 If $\mathrm{IS}.e_1=\mathrm{ILL.IS_l}.e_2$, it means IS is the backward adjacent intersection of $\mathrm{ILL.IS_l}$; thus, insert IS into the back of $\mathrm{ILL.IS_l}$. In this case, IS cannot be the forward adjacent intersection of $\mathrm{ILL.IS_l}$; thus, only check whether $\mathrm{IS}.e_1$ and $\mathrm{ILL.IS_l}.e_2$ are the same edges and ignore $\mathrm{IS}.e_2$ and $\mathrm{ILL.IS_l}.e_1$.

Referring to the data structure introduced in Section 4.1, we know the data element in each CLL is the ILL. Assume $\mathrm{ILL}_i(0\le i<n)$ is the intersection linked list in $\mathrm{CLL}_i(0\le i<n)$ and the first element and the last element in ILL_i are $\mathrm{first\_ele}_i$ and $\mathrm{last\_ele}_i$, the inserted intersection structure is IS, the ECC algorithm can be described briefly as follows.

Initialization: bool check_forward = false, bool check_backward = false, int position = 0, and i=0

Variables check_forward and check_backward are used to mark whether the forward adjacent intersection structure or the backward adjacent intersection structure of IS has been found. Variable position is used to mark the position where the backward adjacent intersection structure of IS has been found.

Step 1 If check_backward=false, check whether $\mathrm{first\_ele}_i$ is the backward adjacent intersection structure of the IS using the above rule 1. Otherwise, turn to step 2.

If $\mathrm{first\_ele}_i$ is the backward adjacent intersection structure of the IS, do (1) check_backward=true, position=i; (2) insert IS into the front of $\mathrm{first\_ele}_i$; (3) if check_forward=true, delete CLL_i and the program terminates. Otherwise, turn to step 2.

Table 1 Parameters and slicing time for Figs. 14, 15, and 16

Model	Size ($L\times W\times H$, mm)	Triangle	Thick (mm)	Time (ms) on a 32-bit system	Time (ms) on a 64-bit system
Fig. 14	82.24×81.09×95.82	297,916	1	219	156
			0.1	1260	826
			0.01	15,640	9922
Fig. 15	18.16×18.26×1.01	180,558	0.1	125	78
			0.01	944	525
			0.001	10,402	6294
Fig. 16	22.58×14.33×23.15	33,730	0.1	62	47
			0.01	718	437
			0.001	12,325	7079

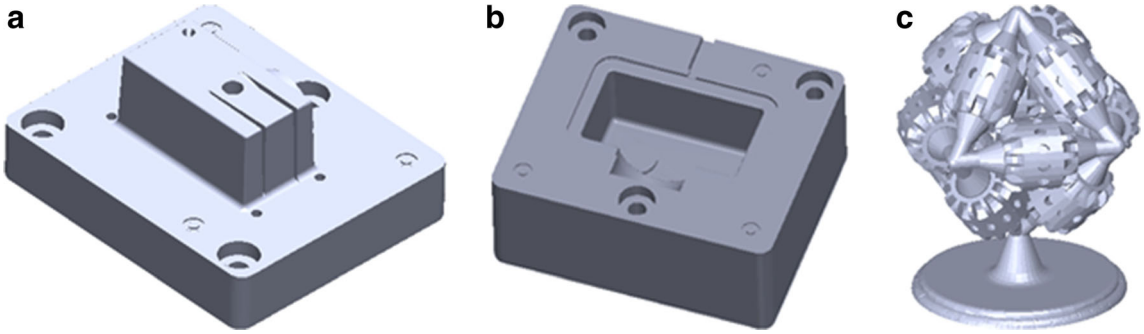


Fig. 17 Comparison models with reference [19]. **a** Mold core model (101.6×44.4×84.1 mm), **b** Mold cavity model (101.6×42.3×84.1 mm), and **c** Gear assembly model (87.6×87.6×105.9 mm)

Step 2 If check_forward=false, check whether last_ele$_i$ is the forward adjacent intersection structure of the IS using the above rule 2. Otherwise, turn to step 3.

If last_ele$_i$ is the forward adjacent intersection structure of the IS, do the following: (1) If position=i which means the backward and forward adjacent intersection structures of the IS are in the same ILL, do not insert IS into the back of last_ele$_i$ and the program terminates. (2) If position≠i, do the following steps.

- check_forward=true.
- Insert IS into the back of last_ele$_i$.
- If check_backward=true, delete $\mathrm{CLL}_{\mathrm{position}}$ and the program terminates. Otherwise, turn to step 3.

Step 3 $i=i+1$, turn to step 1.

To help in understanding the ECC algorithm, taking Fig. 11 as an example, we introduce the ECC algorithm briefly below.

Consider the example of Fig. 11. Assume $T_i(i=1,\cdots,9)$ in Fig. 11b are sorted in sequence $(T_3,T_5,T_1,T_7,T_2,T_6,T_4,T_8,T_9)$ in the STL file and $\mathrm{IS}_j(j=1,\cdots,9)\rightarrow\{\mathrm{pre},p_j,*\mathrm{edge},\mathrm{next}\}$ is the inserted intersection structure corresponding to each triangle $T_i(i=1,\cdots,9)$. For convenience of describing, $\mathrm{IS}_j(j=1,\cdots,9)$ can be written as $\mathrm{IS}_j(j=1,\cdots,9)\rightarrow\{\mathrm{pre},D_j,\mathrm{next}\}$, where, D_j includes two elements p_j and *edge.

The ECC algorithm is described step by step and each step is shown in Fig. 13.

(1) Insert $\mathrm{IS}_3\rightarrow\{\mathrm{pre},D_3,\mathrm{next}\}$.
Since IS_3 is the first inserted element for CLL, IS_3.next points to NULL and IS_3.pre points to itself and insert it into CLL_1 without any judgment.

(2) Insert $\mathrm{IS}_5\rightarrow\{\mathrm{pre},D_5,\mathrm{next}\}$.
After judgment according to the above two rules, we know IS_5 is neither the forward adjacent intersection structure nor the backward adjacent intersection structure of IS_3; thus, let IS_5.pre point to itself and insert it into CLL_2.

(3) Insert $\mathrm{IS}_1\rightarrow\{\mathrm{pre},D_1,\mathrm{next}\}$.
Similar to IS_5, let IS_1.pre point to itself and insert it into CLL_3.

(4) Insert $\mathrm{IS}_7\rightarrow\{\mathrm{pre},D_7,\mathrm{next}\}$.
Similar to IS_5, let IS_7.pre point to itself and insert it into CLL_4.

(5) Insert $\mathrm{IS}_2\rightarrow\{\mathrm{pre},D_2,\mathrm{next}\}$.
Since IS_2 is the backward adjacent intersection structure of IS_1 and the forward adjacent intersection structure of IS_3; thus, insert IS_2 between IS_1 and IS_3. At the same time, change IS_1.pre

Table 2 Comparison results of reference [19] and our algorithm based on the same 3D model

3D model	Size (mm); no. of triangles	Slicing thickness (mm)	Cura software (ms)	Reference [19] method time (ms)	Our method			
					Time (ms) on a 32-bit system	Saving (%)	Time (ms) on a 64-bit system	Saving (%)
Fig. 17a	101.6×44.4×84.1; 4090	0.001	39,234	24,213	10,468	56.8	5032	79.2
Fig. 17b	101.6×42.3×84.1; 3802	0.001	45,085	24,819	9345	62.3	4234	82.9
Fig. 17c	87.6×87.6×105.9; 74,634	0.001	–	540,898	–	–	260,266	51.9

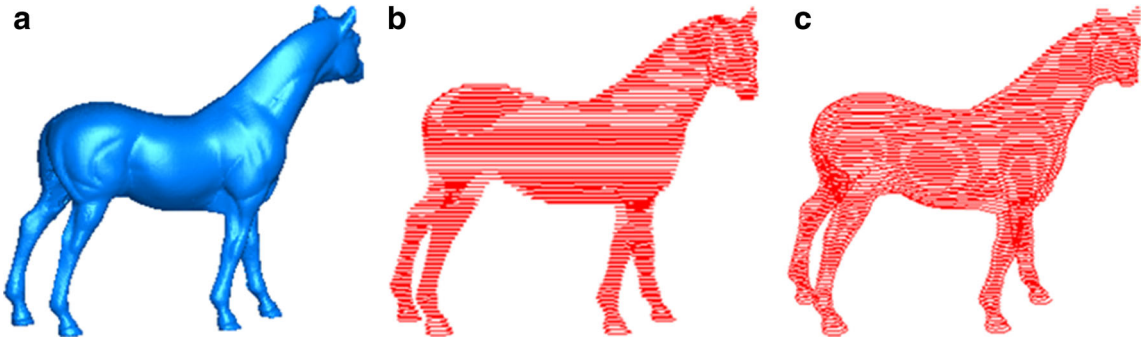


Fig. 18 A horse comparison model with reference [22]. **a** Original 3D model, **b** Sliced model—front view, and **c** Sliced model—3D

from the original pointing to the last element IS_3.

Delete CLL_1 in Fig. 13(4).

(6) Insert $IS_6 \rightarrow \{pre, D_6, next\}$.

Similar to IS_2, insert IS_2 between IS_5 and IS_7.

Delete CLL_3 in Fig. 13(5).

(7) Insert $IS_4 \rightarrow \{pre, D_4, next\}$.

Similar to IS_2, insert IS_4 between IS_3 and IS_5.

Delete CLL_1 in Fig. 13(6).

(8) Insert $IS_8 \rightarrow \{pre, D_8, next\}$.

Using the above two rules, only judge whether the inserted IS_8 is the forward adjacent intersection structure of the first element IS_1 in the CLL_1 in Fig. 13(7) and whether the inserted IS_8 is the backward adjacent intersection structure of the last element IS_7 in the CLL_1 in Fig. 13(7). Ignore $IS_i(i=2,\cdots,6)$ in the CLL_1 in Fig. 13(7).

After judgment using the above rule, we know IS_8 is the backward adjacent intersection structure of the last element IS_7; thus, insert IS_8 into the back of IS_7 and change the IS_1.pre from the original pointing to the last element IS_8.

(9) Insert $IS_9 \rightarrow \{pre, D_9, next\}$

After judging IS_9 and IS_1 using the above rule, we know IS_9 is the forward adjacent intersection structure of IS_1 in the CLL_1 in Fig. 13(8); thus, insert IS_9 into the front of IS_1. Although IS_9 is the backward adjacent intersection structure of IS_8, we do not insert IS_9 into the back of IS_8.

Based on the above procedure, a few key features of the algorithm are listed below.

- Keep the previous pointer *pre of the first element first_ele$_i$ pointing to the last element last_ele$_i$ of ILL_i in CLL_i, since the ECC algorithm only checks the first and the last intersection structures in ILL_i for each inserted intersection. By doing this, we can search the first intersection structure and the last intersection structure immediately and search the corresponding position for inserted intersection structures in minimum time when there are a number of intersection structures in ILL_i.
- When the forward adjacent intersection structure and the backward intersection structure are found in the different ILLs for inserted intersection structures, delete operation must be done. For example, Fig. 13(5), Fig. 13(6), and Fig. 13(7). Otherwise, it means they are found in the same ILLs, insert IS into the front of the first element first_ele$_i$, and do not insert IS into the back of the last element last_ele$_i$ (for example, Fig. 13(9)).

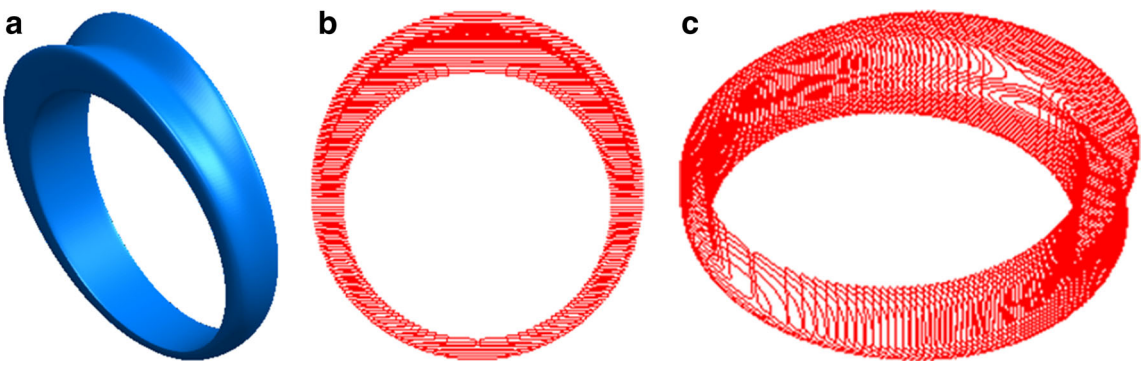


Fig. 19 A ring comparison model in reference [22]. **a** Original 3D model, **b** Sliced model—front view, and **c** Sliced model—3D

Table 3 Comparison results of reference [22] and our algorithm based on the same 3D model

Reference [22] method		Cura software (ms)		Our method		
LDNI resolution (mm)	Time (ms)	Thickness (mm)	Time (ms)	Thickness (mm)	Time (ms) on a 32-bit system	Time (ms) on a 64-bit system
0.1524*0.1524	422	0.1524	187	0.1524	78	47
0.0762*0.0762	546	0.0762	390	0.0762	109	79
0.0381*0.0381	1047	0.0381	593	0.0381	188	125
0.0127*0.0127	6312	0.0127	1638	0.0127	531	359

- All the insert operation and delete operation can be finished by changing the pointer rather than moving element positions in the linked list.

5 Analysis, implementation, and test results

5.1 Algorithm analysis

Since the time complexity and space complexity are two key criteria to evaluate and compare the algorithm, this section will calculate the time complexity and space complexity for our slicing algorithm.

1) Time complexity

The slicing algorithm has three nested procedures. The first one is that read all triangles from the STL file with $O(n)$ time complexity where n is the number of triangles. Suppose that an arbitrary triangle in the STL file intersects with average m slicing planes, the second procedure calculates the intersections for each triangle with time complexity $O(m)$. The last procedure inserts each intersection into the contour linked list. The detail of the ECC algorithm has been described in Section 4.4.

To determine the time complexity to construct one contour, assume there are k ILLs in the CLL before any given intersection IS is inserted into the CLL. For each inserted intersection IS and ILL in one CLL, it only needs to check two times; thus, the worst time complexity for contour construction is $O(2k)$. k is a dynamic small number during the contour construction procedure because there are several adding and delete operations in the ECC algorithm.

From the above analysis, the average total time complexity of our slicing algorithm is $O(nmk)$, where m is far less than n in most cases and k is a dynamic small number compared with the triangle number.

2) Space complexity

Assume each triangle intersects with average m intersecting slicing planes and there are n triangles, the total number of all intersections for the sliced model is mn.

For each intersection structure IS described before, there are four elements—intersection v_{inter} and three pointers *pre, *next,*edge. Each pointer needs a 4-byte RAM on a 32-bit system, and v_{inter} needs 12 bytes if its data type is "float" or 24 bytes if its data type is "double."

For each triangle, there are m intersection structures IS which can be divided into two categories: (1) by intersecting the edge e_2 with slicing planes and (2) by intersecting the edge e_3 with slicing planes. All the *-edge in the first category point to the same structure with four vertices. Also, all the *edge in the second category point to the same structure with four vertices. Thus, there needs 96 bytes if its data type is "float" or 192 bytes if its data type is "double."

Table 4 Comparison results of Rhinoceros, InfinySlice, and our algorithm based on the same 3D model

Ring model			Time (s)					
λ (mm)	Triangle	Thickness (mm)	Reference [22]	InfinySlice	Rhinoceros	Cura software	Our method	
							On a 32-bit system	On a 64-bit system
0.1	1530	0.0762	0.297	1	<1	0.140	0.016	<0.001
0.01	17,308	0.0762	0.328	2	2	0.656	0.031	0.015
0.001	113,692	0.0762	0.453	5	13	3.775	0.094	0.078
0.0001	1,401,978	0.0762	2.500	13	106	16.037	0.842	0.532

Hence, the total RAM for a sliced model is $24n(m+4)$ or $24n(m+8)$ bytes after the slicing is completed. Generally, m is far greater than 4 or 8 because the slicing thickness can reach 0.01 mm or smaller.

5.2 Implementation and results

The slicing algorithm has been implemented using Visual C++. Several test results are provided to demonstrate the main features and the efficiency of this algorithm. The slicing direction is fixed along the z axis. The algorithm was tested in two different test environments as follows.

> Test environment (TN) I: 32-bit Windows 7 system PC with Intel(R) Core(TM) i3-2310M, CPU@2.10 G, 2.10 GHz, NVIDIA NVS 4200M and 2 G RAM which is the most common configuration for a laptop
> Test environment (TN) II: 64-bit Windows 8 system PC with Intel(R) Core(TM) i7-3520M, CPU@2.90 G, 2.90 GHz, NVIDIA NVS 5400M and 8 G RAM

Several test examples, shown in Figs. 14, 15, and 16 were tested. These examples demonstrate the capability of the slicing algorithm to handle complex models with a large number of triangles with multiple and nested contours defining each layer. The slicing times under different slicing thickness for these models are shown in Table 1. Figure 14 is a "chandelier" model with 297,916 triangles. To show the slice contour clearly, we set 1 mm as the slicing thickness for the chandelier model. Figure 15 is a "rosette" model with 180,558 triangles, and as shown in Fig. 15c, the third layer is composed of many nested contours. Figure 16 is a "ring" model with a number of primitives.

In addition to providing test results on these parts, the algorithm is also compared to parts and results presented in two recent papers in the literature [19 and 22], as well a Cura software that is a popular and commercial AM software package developed by Ultimaker Company for STL models. Since Cura is an open-source software, we can obtain the slicing time for a given STL model by adding the test commands in the original program.

Figure 17 shows the models which are provided by the author of [19]. Table 2 shows the comparison results with our algorithm based on the same 3D model with overall size dimensions and number of triangles. Reference [19] compared their slicing algorithm with the existing typical slicing algorithm, and the comparison results illustrate their slicing algorithm takes less time costs than the existing typical slicing algorithm. However, as shown in Table 2, their slicing algorithm is about two to six times slower than the algorithm presented in this paper. Since slicing the model in Fig. 17c with 0.001mm thickness requires more than 2G memory which is beyond the capability of 32-bit system PC, we had to port the slicing program to a 64-bit-system PC which has the ability access larger memory required for slicing with 0.001 mm thickness. This truth can be evidence that the test environment in reference [19] is better than our test environment I since they provide the slicing time for Fig. 17c without providing the test environment in their paper. Thus, the test model in Fig. 17c runs on test environment II mentioned at the beginning of this section.

Figures 18 and 19 show the same models used by reference [22] and provided by the author of reference [22].

The horse model shown in Fig. 18 is used to compare the method in reference [22] and our method. The size of the horse is $19.25\times8.82\times16.04$ mm ($L\times W\times H$), and the total number of triangles is 96,966. The input in reference [22] method is an OBJ model which is composed by triangles, and the input in our method is the STL model which is also composed by triangles. And the outputs are all slicing contours. Hence, the comparison is based on the same parameters; however, reference [22] reports the test data on a Windows XP system PC with Intel(R) Core(TM) i5, CPU750@2.67G, 2.86 GHz and 3 G RAM which is better than our test environment I. The comparison results shown in Table 3 are based on the same slicing thickness. Although the x-y plane resolution

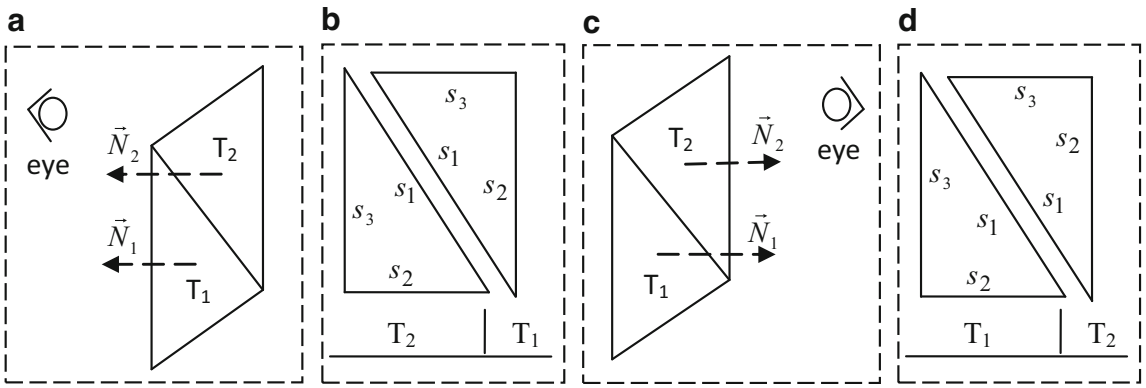


Fig. 20 Schema for describing the feature in section 5.3. **a** Outer surface, **b** Top view of the outer surface—counterclockwise, **c** Inner surface, and **d** Top view of the inner surface—clockwise

is determined by STL precision and we do not know whether the *x*-*y* plane resolution in our method is better than reference [22] method, the data shown in Table 3 can at least illustrate the slicing time in our method is much less than the slicing time in reference [22] method if we assume the STL file precision is the user- or designer-desired *x*-*y* plane resolution.

Figure 19 shows a ring model which is used to compare our slicing algorithm efficiency against the two existing software (Rhinoceros and InfinySlice) mentioned in reference [22]. InfinySlice is a popular and efficient slicer which is designed for STL models. The slicing algorithm in InfinySlice is hardware optimized (parallel computation using multi-core technique), and it is much faster than Rhinoceros [22]. The size of this model is 5.09×19.96×21.0 mm ($L \times W \times H$). Table 4 shows the comparison results under four different chordal errors λ. The slicing times for Rhinoceros and InfinySlice are tested by the author of reference [22]. From Table 4, we can see the number of the STL model with 0.0001 chordal error reaching more than one million (1,401,978), and for the four STL models with different chordal errors λ, our algorithm is faster than the three different slicing methods. To show the slice contour clearly, we set 0.2 mm as the slicing thickness when we slice the "ring" model with 0.001 chordal error and the slicing result is shown in Fig. 19b, c.

5.3 Algorithm feature

The algorithm can distinguish the outer contour and inner contour automatically. A typical slice can have multiple outer contours, and the outer contours can contain inner contours that define holes in the slice. All the outer contours are sorted counterclockwise, and all the inner contours are sorted clockwise. The feature is useful and helpful in contour topological structure reconstruction. Figure 20 is used to explain the feature. Suppose that two given triangles T_1 and T_2 are two adjacent facets, there are two cases as follows.

Case 1 T_1 and T_2 are located on the outer surface of the mesh model (see Fig. 20a).

Case 2 T_1 and T_2 are located on the inner surface of the mesh model (see Fig. 20c).

Their top view of the outer surface and inner surface are shown in Fig. 20b, d, respectively. According to the forward edge and backward edge judgment algorithm discussed above, we can identify the forward and backward edges for each triangle $T_i(i=1,2)$ in Fig. 20b, d, respectively. According to the ECC algorithm introduced in Section 4.4, it is not difficult to understand that the triangles $T_i(i=1,2)$ in Fig. 20a are sorted counterclockwise and the triangles $T_i(i=1,2)$ in Fig. 20c are sorted clockwise.

6 Conclusion

This paper focuses on efficient slice contour construction which can be used to reduce slicing time by proposing a new slicing method. The results show that the algorithm is considerably faster than other existing algorithms and capable of handling large complex models with nested and multiple contours.

The algorithm has two key strengths that make it unique. First, it has the capability to slice a STL model with high detail in fast execution time and can slice a STL model with millions of triangles efficiently independent of the complexity of the STL model. Secondly, it can generate the contours representing the outer and inner contours without any explicit consideration and secondary operations, which can accelerate contour topological structure reconstruction algorithm.

Acknowledgments The comparison models in Fig. 17 are provided by Dr. H.-J. Kim who is the author of reference [19]. Dr. Long Zeng who is a member of Prof. Yuen's research team and the first author of reference [22] contributed the comparison models in Figs. 18 and 19. The jewelry model of Fig. 19 is supported by JewelleryCAD/CAM Ltd. Figures 14 and 15 are free models from the website of "ARCHIBASE.NET." Figure 16 is a free model from the website of "Top3D.net." And the research work is also supported by China Scholarship Council and Prof. Jiquan Hu and Dingfang Chen in Wuhan University of Technology.

References

1. Kirschman CF, Jara-Almonte CC (1992) A parallel slicing algorithm for solid freeform fabrication process. Proceedings of the 1992 Solid Freeform Fabrication Proceedings,August 3–5,Austin, Tx, 26-33
2. Chakraborty D, Choudhury AR (2007) A semi-analytic approach for direct slicing of free form surfaces for layered manufacturing. Rapid Prototyp J 13(4):256–264
3. Sun SH, Chiang HW, Lee MI (2006) Adaptive direct slicing of a commercial cad model for use in rapid prototyping. Int J Adv Manuf Technol 34(7–8):689–701
4. Cao W, Miyamoto Y (2003) Direct slicing from autocad solid models for rapid prototyping. Int J Adv Manuf Technol 21:739–742
5. Starly B, Lau A, Sun W et al (2005) Direct slicing of step based nurbs models for layered manufacturing. Comput Aided Des 37(4):387–397
6. Tata K, Fadel G, Bagchi A et al (1998) Efficient slicing for layered manufacturing. Rapid Prototyp J 4(4):151–167
7. Rock S.J., Wozny M.J. Utilizing topological information to increase scan vector generation efficiency. Appears in Solid Freeform Fabrication Symposium Proceedings, The University of Texas at Austin, Austin, TX, 1991, 28-36
8. McMains S,S'equin C (1999) A coherent sweep plane slicer for layered manufacturing. The proceedings of the 5th ACM SIGG RAPH Symposium on Solid Modeling and Applications, ACM New York, NY, USA,06,1999, 285-295
9. Jun Z (2004) Adaptive slicing for a multi-axis laser aided manufacturing process. J Mech Des 126(2):254
10. Yan JQ, Zhou MY, Xi JT (2004) Adaptive direct slicing with non-uniform cusp heights for rapid prototyping. Int J Adv Manuf Technol 23(1–2):20–27

11. Yang P, Qian X (2008) Adaptive slicing of moving least squares surfaces: toward direct manufacturing of point set surfaces. J Comput Inf Sci Eng 8(3):1–11
12. Sabourin E, Houser SA, Bøhn JH (1996) Adaptive slicing using stepwise uniform refinement. Rapid Prototyp J 2(4):20–26
13. Mani K, Kulkarni P, Dutta D (1999) Region-based adaptive slicing. Computer-Aided Design 31:317–333
14. Ma W, But W-C, He P (2004) Nurbs-based adaptive slicing for efficient rapid prototyping. Comput Aided Des 36(13):1309–1325
15. Hayasi MT, Asiabanpour B (2013) A new adaptive slicing approach for the fully dense freeform fabrication (fdff) process. J Intell Manuf 24(4):683–694
16. Rianmora S, Koomsap P (2010) Recommended slicing positions for adaptive direct slicing by image processing technique. Int J Adv Manuf Technol 46(9–12):1021–1033
17. Zhao Z, Laperriere L (2000) Adaptive direct slicing of the solid model for rapid prototyping. Int J Prod Res 38(1):69–83
18. Liao Y-S, Chiu Y-Y (2001) A new slicing procedure for rapid prototyping systems. Int J Adv Manuf Technol 18:579–585
19. Kim H-J, Wie K-H, Ahn S-H et al (2010) Slicing algorithm for polyhedral models based on vertex shifting. Int J Precision Manuf 11(5):803–807
20. Sanati NA, Rahimi AR, Barazandeh F et al (2009) Improved slicing algorithm employing nearest distance method. Proc Inst Mech Eng B J Eng Manuf 224(5):745–752
21. Vatani M, Rahimi AR, Brazandeh F et al (2009) An enhanced slicing algorithm using nearest distance analysis for layer manufacturing. World Acad Sci, Eng Tech 25:721–726
22. Zeng L, Lai LM-L, Qi D et al (2011) Efficient slicing procedure based on adaptive layer depth normal image. Comput Aided Des 43(12):1577–1586
23. Qi D, Zeng L, Yuen MMF (2013) Robust slicing procedure based on surfel-grid. Computer-Aided Design and Applications 10(6): 965–981
24. Chiu WK, Tan ST (1998) Using dexels to make hollow models for rapid prototyping. Comput Aided Des 30(7):539–547
25. Zhu WM, Yu KM (2001) Dexel-based direct slicing of multi-material assemblies. Int J Adv Manuf Technol 18:285–302