A Heuristic Algorithm for Slicing in the Rapid Freeze Prototyping of Sculptured Bodies

Eric Barnett, Jorge Angeles, Damiano Pasini, and Pieter Sijpkes

Abstract The subject of this paper is a heuristic slicing algorithm for converting STL or PLY CAD data into boundary and fill paths for rapid freeze prototyping (RFP). The algorithm, developed for one commercial robotic system, can also be used to produce toolpaths for other rapid prototyping systems. The algorithm entails five steps: (a) geometry data and other control parameters are imported; (b) the geometry is sliced at several equidistant heights to form bounding paths; (c) contours for the scaffolding material are computed; (d) part and scaffolding paths are buffered in or out to account for deposition path width; and (e) fill paths are computed. A STL file of a 300 mm-high statue of James McGill is used as an example part for demonstrating the capabilities of the algorithm.

1 Introduction

Ice construction has fascinated people for thousands of years. On the practical side, ice roads and shelters have enabled access to and settlement of remote areas. Artistic ice construction has a long tradition of its own; in recent decades, it has become

Eric Barnett
Centre for Intelligent Machines and Department of Mechanical Engineering, McGill University, Montreal, Quebec H3A 2K6, Canada, e-mail: ebarnett@cim.mcgill.ca

Jorge Angeles
Centre for Intelligent Machines and Department of Mechanical Engineering, McGill University, Montreal, Quebec H3A 2K6, Canada, e-mail: angeles@cim.mcgill.ca

Damiano Pasini
Department of Mechanical Engineering, McGill University, Montreal, Quebec H3A 2K6, Canada, e-mail: damiano.pasini@mcgill.ca

Pieter Sijpkes
School of Architecture, McGill University, Montreal, Quebec H3A 2K6, Canada,
e-mail: pieter.sijpkes.mcgill.ca

more popular than ever. As new technologies have become available, ice construction is becoming increasingly automated. Computer numerical control (CNC) ice construction is quite well established, with several companies offering the capability to build certain parts on demand[1].

In traditional CNC machining, a part is formed by removing material using techniques such as milling and drilling. An obvious alternative to *material removal* is *material addition*, also known as rapid prototyping (RP) [1]. RP is a relatively new technology, first achieving widespread use in the 1980s. An important distinction exists between RP *deposition* systems such as fused deposition modeling (FDM) and systems such as selective laser sintering (SLS) and stereolithography, which selectively fuse material already in place. In the field of ice construction, Bryant and Leu have developed a deposition rapid freeze prototyping (RFP) system consisting of a valve/nozzle water delivery system positioned by stepper-motor driven axes [2, 3, 4].

There are significant advantages to using RP as opposed to traditional CNC techniques. Extremely complex parts can be built using RP that would either be impossible or require specific, expensive tooling using traditional CNC techniques. Also, the process from design to fabrication is much simpler using RP; it can be as simple as “printing” a 3D part much as one would print a 2D document with a regular printer. Of course, RP also has many drawbacks, the most obvious being fabrication time: a part that measures roughly 100 mm in each dimension could easily take 50 hours to build. Also, RP machines typically cost several hundred thousand dollars, and most build materials cost about $50/kg. Additional advantages of RFP construction include the use of an inexpensive, more environmentally-friendly build material and lower equipment costs, since a valve/nozzle deposition system is used and fume control infrastructure is not needed.

In [5], we reported on the long history of ice construction research at McGill University. In the past few years, we have focused on the development of computer-assisted ice construction techniques. At the small scale, two systems have been retrofitted for rapid prototyping with ice: an Adept Cobra 600 robot and a Fab@home desktop 3D printer [6]. For the Cobra 600 system, only the end-effector and part of the distal link are inside the freezer during construction. Our Cobra 600 RFP system is novel because the Cobra 600 has never before been used for RP, not to mention RFP. The four-axis SCARA architecture is well-suited to RP, although many additional subsystems and considerable software development are needed to retrofit the system for RFP. Fluid is supplied under pressure from pressurized dispensing tanks outside the freezer; inside the freezer, the fluid lines are heated using an on/off temperature controller; the set point for the temperature near the nozzle tips is 20°C.

Elsewhere, we described in [7] the trajectory control algorithm developed for the Cobra 600 RFP system. This scheme is specifically adapted to the strengths and weaknesses of the Cobra architecture and the Adept C40 Compact Controller. The algorithm is an improved version of a previously developed Matlab code [8]

[1] Ice Sculptures Ltd., Grand Rapids, MI ((www.iceculture.com)
Ice Culture Inc., Hensall, ON (www.iceguru.com)

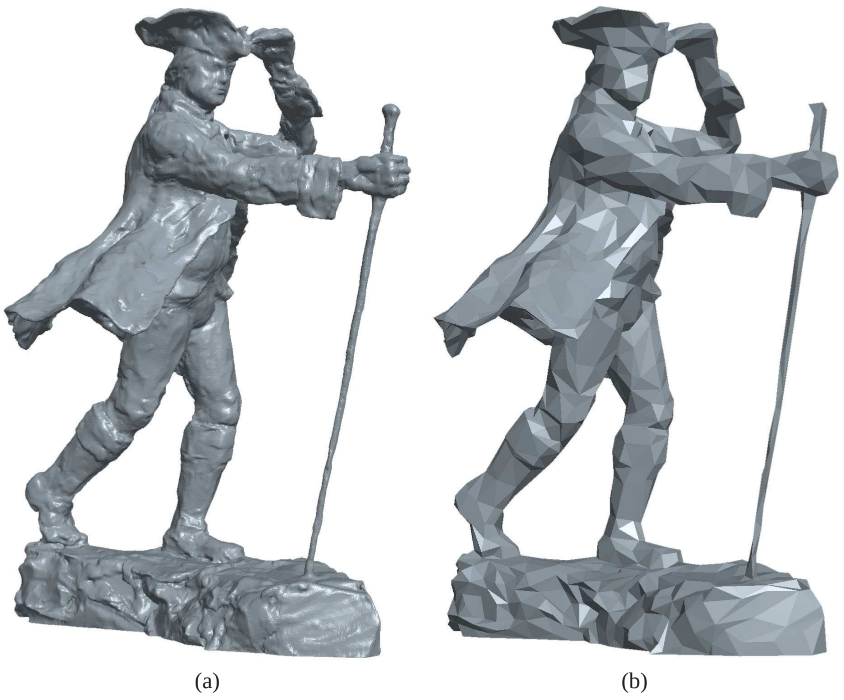

Fig. 1 James McGill STL part: (a) One million facet model; (b) Decimated model with 3906 facets

and since is not adapted specifically for the Cobra 600, it can be used to generate control trajectories for other rapid prototyping systems. The previous code produced self-intersecting paths for some parts with complex geometries; this problem has been eliminated with `rpslice` through the implementation of a new path buffering technique. Additionally, the data storage and processing efficiency has improved considerably, allowing parts with a high level of detail to be sliced.

Matlab is preferable as a coding environment to programming languages such as C or C++ because it is a superlanguage, which means that it offers many useful functions, less development time, and ease of debugging. Specifically, many functions available in the mapping toolbox are useful for rapid prototyping. The performance of `rpslice` is measured in terms of the quality of path data generated and the computational time required; specifications for the computer used are: Intel Core 2 Duo processor @ 2.2 GHz, 2 GB RAM, 500 GB drive @ 7200 RPM. Throughout the paper, we use a James McGill[2] CAD model, shown in Fig. 1, to demonstrate the capabilities of `rpslice`.

[2] James McGill (1744–1813) bequested an estate and funds in his will for the construction of McGill College (later McGill University), officially founded in 1821. A natural scale original statue of James McGill features prominently in the McGill downtown campus.

Many steps are involved in the slicing algorithm; literature on the subject typically focuses on one of the steps [9, 10, 11, 12].

2 Data Import and Transformation

The first step in `rpslice` is to import a TXT file containing control parameters; the parameters for the part shown in Fig. 1(a) are included in Table 1.

Table 1 Control parameter input file

Parameter [units][a]	[Data format] Description (Data restrictions)
1	[int] Number of STL and/or PLY files to import (>0)
jamesmcgill.stl	[char char] STL and/or PLY file names (no spaces within each name)
0	[1/0 (on/off)] *inexact* option: used to indicate if common vertices for adjacent triangular facets are not identical
2	[1/2/3] Slicing option: 1, boundary paths are found; 2, boundary and fill paths are formed; 3, 2D option (indicates the model need only be sliced once and contour data can be copied for all of the part slices)
1	[1/0 (on/off)] scaffolding option
1×10^{-4}	[float] Tolerance value for comparing floating-point data (>0)
1.5 1.5 [mm]	[float float] [part support] Deposition path width (>0)
0.4 [mm]	[float] Slice thickness (>0)
45 [°]	[float] Max. angle between path segments that is considered to be *smooth* (>0)
0 0 0 0	[1/0 float float float] [(on/off) Sx Sy Sz] Scaling vector (>0)
1 -90 0 0 [°]	[1/0 float float float] [(on/off) Rx Ry Rz] Rotation vector (1-2-3 Euler angles)
0 0 0 0 [mm]	[(1/0) Tx Ty Tz][b] [(on/off) Tx Ty Tz] Translation vector
0 1	[float float] [min max] Part fraction to slice: [0 1] indicates the entire part ($0 <=$ float $<= 1$)
0 1	[float float] [min max] Scaffolding subset($0 <=$ float $<= 1$)
1 1 1	[int int int] [p1 p2 p3] Part copy vector: p3 part copies will be built, laid out in a grid measuring [p1 $\times$ p2] (>0)
5 [mm]	[float] Scaffolding buffer

[a] The parameters used for the James McGill ice statue STL file are shown here.
[b] There are three options for each axis: Numerical entries [float] indicate the minimum coordinate for the part on an axis; [cent] entries will center the part on an axis; [orig] entries will keep the part centered at its original location on an axis.

2.1 *Facet data importation with* `facetread`

PLY or STL facet data are imported with our `facetread` function, which can read binary or ASCII format. Table 2 shows the storage and processing efficiencies for different formats and settings, using the the James McGill statue shown in Fig. 1(a) as an example.

It can be seen that ASCII files are considerably less efficient in both data-storage and processing time. More storage space is required for ASCII files, mainly because approximately twice as many characters are needed to store floating-point numbers in the ASCII format. Additionally, for STL files in the ASCII format, several text labels surround the data for each facet. Both of these factors contribute in making STL ASCII file sizes approximately five times larger than their binary counterparts. Reading in ASCII data also takes much longer, because file sizes are larger, data must be encoded into machine format, and data are stored in variable-width fields. The advantage of using the ASCII format is that it can be read directly by people, which can be useful for debugging. However, since the binary format is significantly more efficient in terms of processing time and data-storage, it should be used in almost all cases.

The PLY format also consists of triangular facets, which are stored much more efficiently. In the STL format, each facet is stored as nine floating point numbers, which represent the three vertices. Since adjacent facets share a common edge, every vertex is repeated several times in the file. In the PLY format, each unique vertex is stored as three floating point numbers. Each facet is stored as three integers, which are the indices of its three vertices. From Table 2, we see that the PLY format is significantly more storage-efficient than the STL format.

In order to exploit Matlab's vectorization capabilities and minimize import time, looping structures should be avoided when importing large amounts of data. For some of the formats described above, this can be difficult, since each facet contains several different data types, and in some cases, extraneous information. However, with Matlab's `fscanf` and `fread` commands for ASCII and binary data respectively, it is possible to skip unwanted information and read in all facet data with a single command.

Throughout `rpslice`, looping structures are avoided whenever possible to reduce processing time; this is especially important when large, detailed models are sliced. Simple computations are vectorized, and more complex operations are accomplished with cell functions. Both of these techniques lead to significant reductions in processing time. However, they can also be highly memory-intensive, especially when cell functions are used. For some especially complex or memory-intensive operations, `for` loops are used.

2.2 Transformation of facet data

The `rpslice` algorithm translates, rotates, and scales the triangular facets based on the data imported from the input file. This feature is especially useful when a STL or PLY file is the only CAD model available. If rotation or scaling is desired, the model is initially moved to a location where its minimum value in each dimension is zero. Facets are then rotated, scaled, and translated as described in Table 1.

Table 2 A comparison of different techniques for reading in the facet data for the part shown in Fig. 1(a)

File format	Technique	File size (MB)	Import time (s)
binary PLY	two `fread` statements	18.1	0.7
ASCII PLY	two `fscanf` statements	34.9	7.1
binary STL	one `fread` statement	47.6	0.7
binary STL	`for` loop with one `fread` statement	47.6	32.5
ASCII STL	one `fscanf` statement	257.8	50.0
ASCII STL [a]	`while` loop with multiple `fscanf` statements	16.1	603.8

[a] The previous code [8] was used in this case; A decimated model with only 62500 facets was used because the one million-facet model failed to read in after over ten hours.

Rotation is performed using the 1-2-3 Euler-angle order, though this order can be easily changed, for example, if the 1-2-3 order leads to an orientation singularity typical of the Euler-angle representation of rotations [13].

3 Part Boundary Paths

The boundary contours for each slice are formed using a function called `facet-slice`, composed of the three subfunctions, `intersectfacet`, `segmerge` and `pathfilter`. First, `intersectfacet` is used to identify the facets that intersect each slice and compute plane-plane intersections, forming an array of disconnected line segments. Next, an iterative procedure is performed to link the line segments in each slice to form part boundary paths. A function called `polymerge` exists in Matlab to perform this task, though it returns an "OUT OF MEMORY" error if the array of line segments to merge is too large. As such, we developed a new iterative linking function, which we call `segmerge`. This function forms a polygon segment-by-segment: at each iteration, the last point of the partially-completed polygon is matched with an endpoint of a segment in the disconnected array. A comparison between `segmerge` and `polymerge` is included in Table 3. For RFP, the model is sliced at increments of 0.4 mm, which corresponds to the thickness of scaffolding slices. However, since ice slices are only 0.2 mm thick, every second water slice has repeated boundary paths.

Boundary paths that have unnecessarily high resolution are decimated using `pathfilter`; a minimum point-spacing of 0.2 mm is imposed to accommodate limitations of the Cobra controller [7]. Additionally, subsequent steps in the algorithm complete much more quickly when there are fewer path points.

Table 3 A comparison of `segmerge` and Matlab's `polymerge` function for forming contours from segments with matching endpoints[a]

Facets	Segments	Computational Time (s)	
		`polymerge`	`segmerge`
3906	126	0.05	0.03
62500	545	0.17	0.06
500000	1604	1.10	0.22
1000000	2154	–[b]	0.35

[a] Different resolutions are tested for a slice 8 mm from the base of the part shown in Fig. 1
[b] OUT OF MEMORY error

4 Scaffolding Boundary Paths

For the Cobra 600 RFP system, scaffolding is needed to support overhanging part features. During deposition, the RFP system alternates between part and scaffolding slices. It is possible to model the scaffolding with CAD software by "subtracting" the part from a block of material. However, this technique is not always straightforward and will usually result substantial waste of scaffolding material, since support is only needed *below* overhanging features of the part. Additionally, if the part is only available in a surface rather than a solid format, this technique frequently fails. These factors motivate the generation of scaffolding contours from the part itself, after the slicing stage.

Chalasani et al. [14] developed a rapid prototyping technique in which they consider only the 2D contours in each slicing plane, while others [9, 15] consider the 3D model, analyzing triangular facets. Our `rpscaf` function is similar to the first technique, with additional options included to accommodate the specific characteristics of the Cobra RFP system.

The `rpscaf` function works in the following manner. A "merged" region is formed for every slice, which consists of the Boolean union between all of the part slices above. The scaffolding region for one slice is defined as the Boolean subtraction of the part region for that slice from the merged region for the slice above. This technique is fast and efficient, though it also produces many thin scaffolding features. Since thin features are built less accurately by the RFP system, the merged regions are buffered outward, by an amount specified in the input TXT file, before performing the Boolean subtraction. This is accomplished using the `bufferf` function, described in Sect. 5. All ice features in every slice are thus completely surrounded by scaffolding; obviously, this increases the amount of scaffolding material and construction time needed. However, it also significantly increases the part accuracy. Figure 2 displays graphical representations of the regions described here for one slice of the part shown in Fig. 1(a). In Fig. 2(b), part and scaffolding regions are shown *after* path buffering, described in Sect. 5.

5 Path Buffering

Path buffering is needed because many of the part and scaffolding paths generated in the previous two steps are identical. Additionally, since the path width *pw* used with our RFP systems is 1.5 mm, if the path and scaffolding boundaries were used directly as deposition paths, an error of $pw/2 = 0.75$ mm would be introduced. To avoid this error, all boundary paths are buffered inward by $pw/2$ to form the deposition paths. A custom contour-buffering technique was implemented in [8], but it was not computationally efficient, and produced self-intersecting paths for certain complex geometries. For `rpslice`, path buffering is performed with `bufferf`, a modified version of the `bufferm` function available in Matlab's mapping toolbox.

5.1 *The Matlab* `bufferm` *function*

The syntax of `bufferm` is: `[latb,lonb] = bufferm(lat,lon,...`
`dist,direction,npts,outputformat)`

`[latb,lonb]`	output latitude-longitude region
`[lat,lon]`	input latitude-longitude region
`dist`	buffering distance
`direction`	buffering direction (`in` or `out`)
`npts`	number of points used for vertex buffer circles
`outputformat`	data output format (`cell`, `vector`, or `cutvector`)

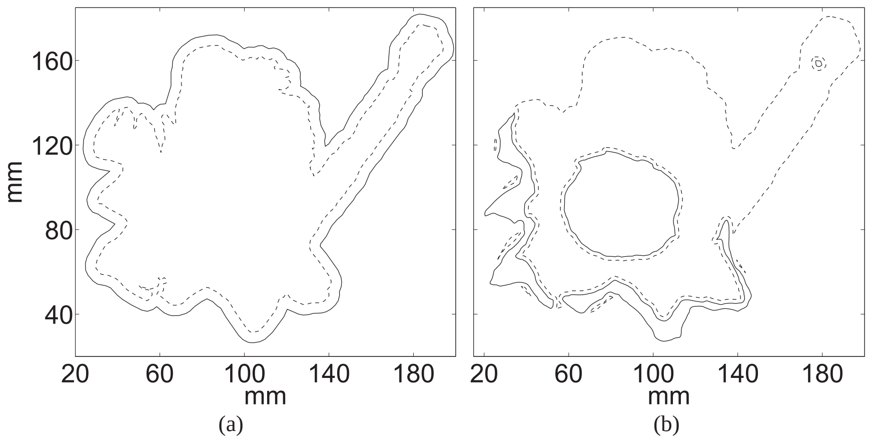

Fig. 2 (a) Merged region for one slice before buffering (dashed) and after buffering (solid); (b) Part (solid) and scaffolding (dashed) regions for the same slice

A polygon region is composed of several contours, stored in the NaN-delimited vector format or in separate cells of an array. Vertices for external contours are ordered clockwise, while internal contours, or holes, are ordered counterclockwise. Internal contours are buffered in the opposite direction to the commanded buffering direction.

The input contours are buffered one at a time. Rectangles are formed, centered around each contour segment, of width `2dist` and length the segment length. Circles with `npts` points are formed around each point on the contour. The Matlab function `polybool` is then used to form the Boolean union between each circle and the adjacent rectangle on one side. The `polybool` function then merges all of these new polygons, one at a time, to form the buffered contours. A Boolean union or intersection is then performed between the buffered contour group and the original contour, depending on whether the inward or outward buffering is desired and whether or not the contour represents a hole.

In this fashion, a buffered contour group composed of one or more contours is formed for each of the input contours. The groups are then merged, one at a time, to form the output region, using Boolean unions for external contours and subtractions for internal contours.

5.2 `bufferf`*, a contour buffering function for planar regions*

For `bufferm`, input data are assumed to be latitude and longitude coordinates, and geometric calculations are thus done for a spherical rather than a planar surface. Therefore, if contour data pertain to a planar surface, this function will introduce some error when computing buffered contours. Data can be scaled down to fit within a region where surface curvature has a minimal effect to minimize this error. However, computations are simpler for planar geometry and it is preferable to work directly with input data. Thus, we have replaced all spherical surface computations with their planar counterparts.

The `bufferm` function is quite robust; as long as input data are in the correct format, it will never produce self-intersecting paths. However, it also has a high computational cost; thus, any modifications that can reduce this cost will significantly impact the overall processing time for `rpslice`. Nearly all of the computational time expended during `bufferm` is during calls to `polybool`, a function in Matlab that performs Boolean operations on polygonal regions. As a result, our modifications consist mostly of techniques for minimizing the number of calls to `polybool`.

Firstly, we replaced the circle-and-rectangle technique used by `bufferm` with a function which computes a single “boundseg” polygon for each line segment, as shown in Fig. 3(a). This reduces the number of polygons to merge by half.

Secondly, we eliminated the iterative procedure whereby the boundseg polygons are merged one-by-one. As with `bufferm`, `polybool` can be used to perform Boolean operations on polygonal regions composed of multiple contours. However,

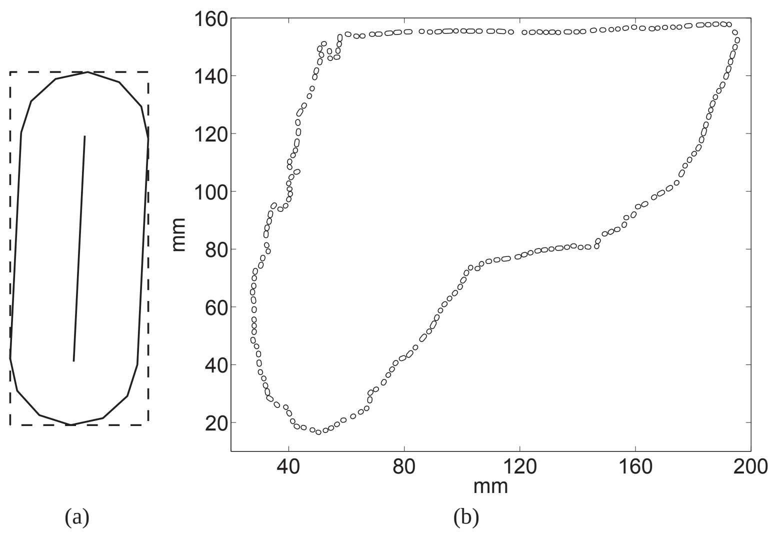

Fig. 3 (a) One contour segment along with its boundseg polygon and bounding rectangle; (b) One group of boundseg polygons for a slice 8 mm from the base for the part shown in Fig. 1(a)

contours within each region must not intersect with either each other or themselves. Therefore, we created a `prebool` function, which forms a handful of contour groups from the boundseg polygons for one contour. The maxima and minima in the Cartesian dimensions are used to form a bounding rectangle around each boundseg polygon, as shown in Fig. 3(a); contour groups are formed composed of contours whose bounding rectangles are non-intersecting. One boundseg contour group, for a slice 8 mm from the base of the STL model shown in Fig. 1(a), is shown in Fig. 3(b). Since the contour is composed of 1018 segments, 2036 Boolean operations would be needed to buffer it using `bufferm`. Using the boundseg polygons and `prebool`, only 10 Boolean operations are needed because 10 groups of segment-buffered polygons are formed. An alternative method would be to detect whether polygons intersect, which would result in even fewer contour groups to merge. However, detecting polygon intersections requires nearly as much computational time as performing Boolean operations, thereby lowering the overall performance.

We also reduced the number of calls to `polybool` needed to merge the buffered versions of the original contours that form the region. Since `polybool` operations are only needed when polygonal regions are intersecting, non-intersecting polygons can simply be grouped with each other in cell arrays or the NaN-separated vector format. In our slicing algorithm, region contours will not intersect as long as the STL input file is properly defined. The nesting among contours, or the number of other contours each contour lies within, is also known. Even levels of nesting indicate external contours and odd levels of nesting indicate internal contours. In `bufferf`, buffered contours are first merged to form groups of contours at each nesting level. All contours that have been buffered inward will also not intersect with each other, so one group of contours at each inwardly-buffered nesting level can be formed with no calls to `polybool`. Contours that have been buffered outward might intersect, however; thus, at each nesting level, groups of non-intersecting contours are found with `prebool` and then merged with `polybool`, if necessary. Once one group of contours has been formed at each nesting level, a series of Boolean operations is performed, alternating between subtraction and union, to form the final buffered region. Usually, there are only 1–3 levels of nesting, which means 0–2 Boolean operations are needed at this step.

Table 4 shows the computational times for `bufferm` and `bufferf` for different slices of the James McGill statue in Fig. 1(a). The `bufferf` function is always significantly faster, and the performance gain becomes most apparent when the polygon resolution is high.

Table 4 Computational times for Matlab's `bufferm` function vs. `bufferf` for the James McGill statue in Fig. 1(a)

Slice height (mm)	Vertices	Computational Time (s)	
		`bufferm`	`bufferf`
8	235	1.55	0.21
8	2155	113.93	4.67
178	235	1.97	0.29
178	2973	49.90	3.76

6 Fill Paths

Many authors [16, 17, 12] reportedly use zig-zag paths to fill object areas in their rapid prototyping algorithms. With the zig-zag technique, typically a series of parallel lines are intersected with the boundary contours, and adjacent lines are linked to produce paths that are very long and have many abrupt changes in direction. Xu and Shaw [18] consider 2D material gradients within each slice to produce smooth filling paths. We developed an iterative inward path buffering technique to form fill paths, mainly to avoid abrupt changes in direction [8]. However, this technique is computationally very intensive, smooth paths are not produced in all cases, and it fails for certain complex geometries. To address these problems, we have developed a new filling-path technique, called `zzfill`, similar to the zig-zag filling technique, except that the parallel lines are simply not linked. This eliminates *all* abrupt changes in direction during deposition, since we require that deposition only occurs when the tool is moving at a constant speed [7]. A graphical representation of the part and support contours, with the fill paths, is shown in Fig. 4.

7 Results

Table 5 shows the breakdown of computational times for slicing the coarse and fine James McGill STL models shown in Fig. 1. Additional post-processing steps are needed to produce the trajectory data needed for the Cobra deposition control program [7]. The part resolution should be close to the minimum point spacing imposed during `facetslice` to achieve the optimum balance between computational speed and part accuracy.

Table 5 Breakdown of computational time for the slicing algorithm for the James McGill statue in Fig. 1(a)

Facets	3906	1 million
traj. points[a]	933 858	1 824 094
	Computational Time (s)	
`facetread`	0.03	0.74
`facetslice`	6.59	194.72
`rpscaf`	108.85	135.37
`bufferf`	297.32	1469.80
`zzfill`	12.09	14.85
total	441.25	1844.80

[a] Total trajectory points in Cartesian space

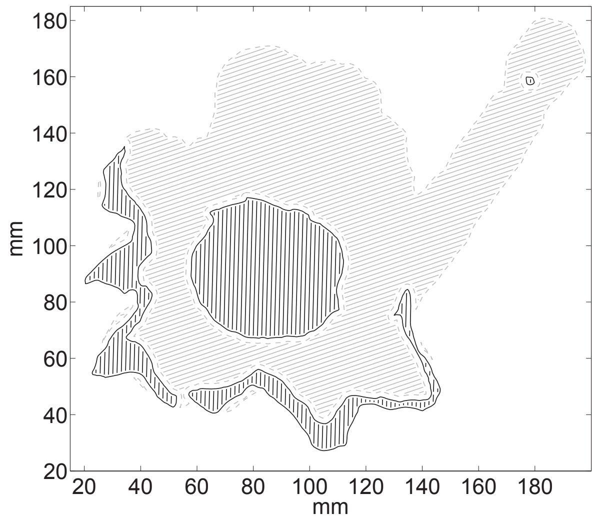

Fig. 4 Fill paths for one slice of the part shown in Fig. 1(a); solid lines indicate the part boundaries and dashed lines indicate the scaffolding boundaries

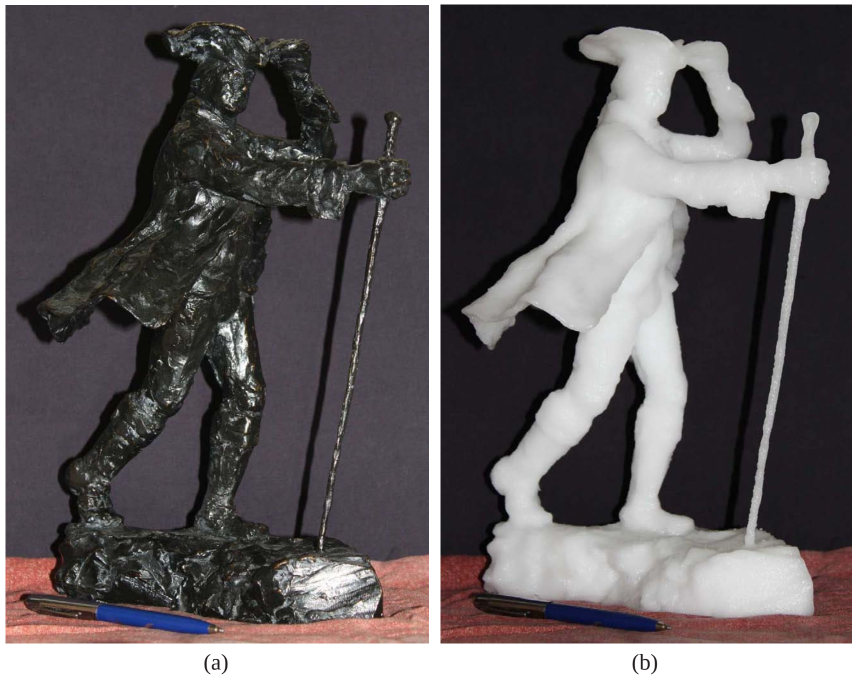

Fig. 5 James McGill statue, 306 mm high: (a) Original bronze statue; (b) Ice statue, built on its side, with the Cobra 600 RFP system: 852 slices built in 132 hours

8 Conclusions

The major steps involved in a slicing algorithm for Rapid Freeze Prototyping are reported in this paper. Some functions used in the algorithm were developed from scratch, while others were produced by either modifying Matlab functions or using them directly. The paths produced with the algorithm were used to generate the control trajectories for building the James McGill ice statue of Fig. 5(b). Figure 5(a) shows the bronze version from which the STL file of Fig. 1 was produced. Future work with the slicing algorithm will involve increasing the efficiency and robustness of some parts of the code. Also, the algorithm will be configured to be accessible to a wide range of potential users. Ultimately, a user will be able to provide a STL file and an input TXT file, execute an EXE file, and produce the trajectory control information for the Cobra 600 RFP system, without any expert assistance.

Acknowledgements The authors gratefully acknowledge the support of The Social Sciences and Humanities Research Council of Canada (SSHRC), the Natural Sciences and Engineering Research Council of Canada (NSERC), Quebec's Fonds québecois de la recherche sur la nature et les technologies (FQRNT), and McGill University's Faculty of Engineering. The generous rebate received from Adept Technology is dutifully acknowledged.

References

1. R.H. Crawford, J.J. Beaman, IEEE Spectr. **36**(2), 34 (1999)
2. F.D. Bryant, M.C. Leu, Rapid Prototyp. J. **55**(1), 317 (2009)
3. F.D. Bryant, G. Sui, M.C. Leu, Rapid Prototyp. J. **9**(1), 19 (2003)
4. W. Zhang, M.C. Leu, Z. Yi, Y. Yan, IEEE Spectr. **20**, 139 (1999)
5. P. Sijpkes, E. Barnett, J. Angeles, D. Pasini, in *Archit. Res. Cent. Consort. Spring Conf. (ARCC 2009)* (San Antonio, TX, Apr. 15–18, 2009), 6 pages
6. E. Barnett, J. Angeles, D. Pasini, P. Sijpkes, in *IEEE Int. Conf. Robot. Autom.* (Kobe, JP, May 12–17, 2009), pp. 146–151
7. E. Barnett, J. Angeles, D. Pasini, P. Sijpkes, in *to appear in Proc. ASME 2010 Int. Des. Eng. Tech. Conf.* (Montreal, QC, Canada, Aug. 15–18, 2010)
8. A. Ossino, E. Barnett, J. Angeles, D. Pasini, P. Sijpkes, Trans. Can. Soc. Mech. Eng. **33**(4), 689 (2009)
9. S. Allen, D. Dutta, J. Des. Manuf. **5**(3), 153 (1995)
10. S.H. Choi, K.T. Kwok, Rapid Prototyp. J. **8**(3), 161 (2002)
11. P. Haipeng, Z. Tianrui, Rapid Prototyp. J. **187–188**, 623 (2007)
12. R.C. Luo, Y.L. Pan, C.J. Wang, Z.H. Huang, in *IEEE Int. Conf. Robot. Autom.* (Orlando, FL, May 15–19, 2006), pp. 883–888
13. J. Angeles, *Rotational Kinematics* (Springer-Verlag, New York, 1988)
14. K. Chalasani, L. Jones, L. Roscoe, in *Solid Freeform Fabr. Symp.* (Austin, TX, Aug. 7–9, 1995), pp. 229–241
15. X. Huang, C. Ye, S. Wu, K. Guo, J. Mo, Int. J. Adv. Manuf. Tech. **42**, 1074 (2008)
16. H. Chen, N. Xi, W. Sheng, Y.Chen, A. Roche, J. Dahl, in *IEEE Int. Conf. Robot. Autom.* (Taipei, Taiwan, Sept. 14–19, 2003), pp. 3504–3509
17. R.C. Luo, C.L. Chang, J.H. Tzou, Z.H. Huang, in *IEEE Int. Conf. Robot. Autom.* (Barcelona, Spain, Apr. 18–22, 2005), pp. 584–589
18. A. Xu, L.L. Shaw, Comput. Aided Des. **37**, 1308 (2005)