COMPUTER-AIDED DESIGN & APPLICATIONS, 2017
VOL. 14, NO. 2, 242–249
http://dx.doi.org/10.1080/16864360.2016.1223443

Generating point clouds for slicing free-form objects for 3-D printing

William Oropallo[a], Les A. Piegl[a], Paul Rosen[a] and Khairan Rajab[b]

[a]University of South Florida, USA; [b]Najran University, Saudi Arabia

ABSTRACT
3-D printing, also known as additive manufacturing, has gained a lot of attention both within and outside CAD research. Even popular media have touted the technology as one of the game changer technologies of the 21st century. Simply stated, most printing devices add material to an unfinished part, layer by layer, until the entire object is completed. In order to make this happen, the object is sliced into thin layers which are produced and glued together. Since NURBS are the standard form of modeling tools, the process entails converting the NURBS into an STL model (piecewise triangular model) which is then sliced into a set of closed polygonal loops. In order to avoid the many problems with the STL-based slicing, in this paper we investigate a point cloud-based approach to direct slicing of NURBS based models. It uses the original NURBS model and converts the model into a point cloud, based on layer thickness and accuracy requirements, for direct slicing. The only major computational requirement is point evaluation which can be done error free and in an inexpensive manner. The generation of the point cloud is the main topic of this paper.

KEYWORDS
3-D printing; NURBS; point cloud; object slicing

1. Introduction

3-D printing has become a viable technology in the past several decades. Although it is not a new technology, its roots go back as early as the 1960s, increases in memory and computing power made it useful for a variety of purposes ranging from rapid prototyping to high precision manufacturing. There has not been any shortage in popular media coverage either; wild speculations range from made-at-home parts to cloning an entire human being. While some of these speculations may actually come true over time, the current state-of-the-art is anything but close to the imagination of the authors of these claims. It is not uncommon to see that once a technology is on the popular bandwagon, researchers tend to forget about the gaps left in the fundamentals. 3-D printing is no exception [3]. In this paper we address one of those fundamentals: object slicing [2, 4, 9–12, 14].

The current state-of-the-art is to model the object using NURBS. Once the modeling has been completed, the object is converted into a tessellated model and saved as an STL file. The file is then passed onto a slicer that cuts the triangulated model into closed polygonal sections. The interior of the sections is filled with the required material and glued to the layer underneath. In principle this all sounds well, however, there are several fundamental problems with the tessellation as well as the slicing:

- The tessellation produces an approximation only and it takes a lot of triangles to get accurate results. This in itself is not the issue, the problem is that once a certain level of tolerance is reached, the tessellation software tends to have all sorts of numerical problems (not to mention the time it takes to complete the large number of numerical calculations).
- Currently used commercial tessellation software may not triangulate the object with a uniform triangulation, i.e. neighboring triangles can be quite different in size giving rise to numerical problems downstream. If the tessellation is done via an error tolerance, then flat areas produce large triangles whereas curved areas give rise to small ones. Triangles of varying sizes do not mix well in numerical processes.
- STL files tend to have gaps, dangling edges and faces, overlapping triangles, etc. This is due, in part, to problems with the tessellation software and to the fact that an object is passed from one system to the other during the (collaborative) design process. Unfortunately, these systems have different internal representations and hence conversion is almost always necessary [6].

CONTACT Les A. Piegl lespiegl@mail.usf.edu

Sadly, conversions come with losses which in turn produce poor results during triangulation.

- Poor quality STL files need to be healed in order to be useful for slicing [13]. Unfortunately, healing produces even more losses as the underlying surface is missing (the idea of the STL model is to avoid the NURBS model, i.e. to eliminate the precise model altogether). What is needed is not model repair, but rather model regeneration to given requirements [7].
- Slicing triangulated models gives rise to all sorts of numerical anomalies: nearly parallel faces, overlapping faces, touching along an edge or vertex, etc. These challenges are difficult to handle as they require a tolerance (tessellation) upon a tolerance (anomaly) upon another tolerance (numerical process) to handle. In the worst case, these stacked-up tolerances may produce a model that is not within the required accuracy [8].
- Most mechanical engineering parts contain tons of conventional objects such as planes and cylinders. When the object has a planar part that is parallel to the slicer, it can pose a challenge if the plane falls between two slicing positions. Again, this anomaly must be handled with special code that makes the slicer a complicated piece of software that may not be easy to maintain.

This paper investigates the possibility to use a point cloud to address the above problems. It uses the original NURBS model and converts the model into a point cloud, based on layer thickness and accuracy requirements, for direct slicing. The method requires no expensive tessellation, has no anomalies, needs no model repair and no conversion. The only major computational requirement is point evaluation which can be done error free and in an inexpensive manner. Such an approach may have been prohibitive a decade or so ago. However, with the proliferation of powerful hardware and the abundance of memory, point-based approaches are more than viable today. Add the possibility of parallelization via a cheap multi-core GPU, the point-based approach becomes better suited in today's applications than the triangle-based ones.

The organization of the paper is as follows. Section 2 introduces some NURBS notations for a better comprehension of the B-spline details. In Section 3 we introduce the overall approach followed by Section 4 where object decomposition is discussed. Section 5 provides details on how the point cloud is generated, providing ample examples and illustration. A conclusion section closes the paper.

2. B-spline notation

To better comprehend the method presented herein, some B-spline notations are in order. A B-spline surface or degree p in u-direction and q in v-direction is a tensor-product surface in the following form [5]:

$$S(u,\ v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)P_{i,j}^{w}$$

where $P_{i,j}^{w}$ are the weighted control points, $N_{i,p}(u)$ and $N_{j,q}(v)$are the normalized B-splines defined over the knot vectors

$$U = \{\underbrace{u_0 = \cdots = u_p}_{p+1}, u_{p+1}, \ldots, u_r, \underbrace{u_{r-p} = \cdots = u_r}_{p+1}\}$$

$$V = \{\underbrace{v_0 = \cdots = v_q}_{q+1}, v_{q+1}, \ldots, v_s, \underbrace{v_{s-q} = \cdots = v_s}_{q+1}\}$$

We assume that the knot vectors are always clamped, i.e. the end knots are repeated with the multiplicities above. If the B-spline surface has no internal knots, it degenerates to a Bezier surface, which will be used for point cloud generation in this paper.

3. Overview of the approach

Objects to be printed (manufactured) are assumed to be bounded by B-spline surfaces, i.e. any object is considered as a collection of B-spline patches. Holes and cavities are covered by B-spline patches as well and the inner part of the object is determined by the orientation of the covering surfaces.

We take two parameters from the 3-D printer:

- λ - the layer thickness. We assume this to be constant throughout the printing process.
- ε- accuracy, i.e. the addressability of the printer, the printer head can move from position to position with at least that much distance.

The overview of our approach is as follows:

- Take each B-spline surface that bounds the object and perform the operation below for each surface.
- Decompose each surface into small sub-patches that are Bezier surfaces.
- Create a global data structure that holds all the sub-patches.
- For each slicer, create an active list of sub-surfaces that may intersect the slicing plane (a reference of the sub-patch in the global data structure).

- Sample the patches in the active list based on the required tolerance and keep the resulting point cloud in a temporary data structure.
- As the slicer moves up, refresh the active list and the point cloud: drop surfaces (and the points on them) that no longer intersect the plane and add the ones that became intersecting. Sample the new surfaces and add the points to the active point cloud.

Since the method deals with a point cloud, a completely global approach would require the storage, manipulation as well as the processing of a large amount of data. What we do in this work is based on the principle of *coherence*; we are looking at a small band of surfaces and the corresponding points that are sampled off of those surfaces. In other words, we maintain a dynamic list for both the sub-patches as well as the points belonging to those surfaces.

4. Surface decomposition

In this section details are given on how a set of B-splines are decomposed based on the setup parameters of the 3-D printer.

4.1. Slicer band

Decomposing a set of surfaces so that on average only a few would intersect a slicer would result in a very large set of sub-surfaces. We grouped a number of slicers together to form a band for the purpose of decomposition only. In other words, given the layer thickness λ we introduced a band defined as $\beta = c\lambda$ where $c = \{2,\ 3,\ 4,\ \ldots\}$. The value for the constant c must be determined for each type of object, e.g. simple surfaces or highly free-form surfaces. The examples below will be provided for the following settings:

$$\lambda = 0.1mm \quad \beta = 3\lambda = 0.3mm$$

4.2. Surface extents

In order to produce a sensible surface decomposition, the estimation of surface areas is required. Such area is estimated by the approximate surface extents in each of the u- and v-directions. Since a precise area is not required, we use the following approach:

- Perform a knot refinement based on a level of refinement. That is, for level zero, no knots are inserted. For level one, one knot is inserted into each non-zero knot span. For level two the spans are refined twice, one knot is inserted into the middle of each span resulting in two sub-spans which in turn are refined as well. In general, a level k refinement produces $2^k - 1$ new knots in each span. For a good surface extent computation a level one or two refinement is quite adequate.
- Compute the maximum length of the control polygons along the rows and the columns.

The output is $\{L_u,\ L_v\}$ the approximate extents in u- and v-directions. We will make use of these quantities in the next subsection to generate the sub-patches.

4.3. Sub-patch computation

This part of the algorithm creates small sub-patches out of each bounding B-spline surface. Each B-spline surface consists of a set of Bezier patches, obtained by inserting the interior knots multiple times so that the total multiplicity is equal to the degrees. While these Bezier patches are smaller than the B-splines, they are still much larger than what the printer requires, so they need to be further subdivided. This is how we do it. Assume that the number of non-zero knot spans in u- and v-direction are n_u and n_v, respectively. These non-zero knot spans are the ones over which the Bezier surfaces are defined. The approximate Bezier patch extents are

$$LB_u = \frac{L_u}{n_u} \quad LB_v = \frac{L_v}{n_v}$$

We want the subdivided sub-patches to be roughly of size β so we insert

$$k_u = \left\lfloor \frac{LB_u}{\beta} + 0.5 \right\rfloor \quad k_v = \left\lfloor \frac{LB_v}{\beta} + 0.5 \right\rfloor$$

number of knots into each non-zero knot span. Figure 1 shows an example of a head model with X-Y-Z dimensions (32 x 43 x 45 mm). The model of the head contains 601 B-spline surfaces. After refinement for $\beta = 0.3$ mm band thickness the number of sub-patches becomes 57,975. This is two orders of magnitude higher than the original set of surfaces, clearly underlying the importance of proper decomposition. Improper subdivision could generate either an unnecessarily high number or produce surfaces that are inadequate in meeting manufacturing requirements.

Note that the number of sub-patches are very manageable for engineering tolerances, however, when the accuracy gets too high, the decomposition will results in quite some number of surfaces, as shown in the Table 1 below.

4.4. Surface lists

One of the many challenges of dealing with massive amounts of data is storage and ease of access. Storage

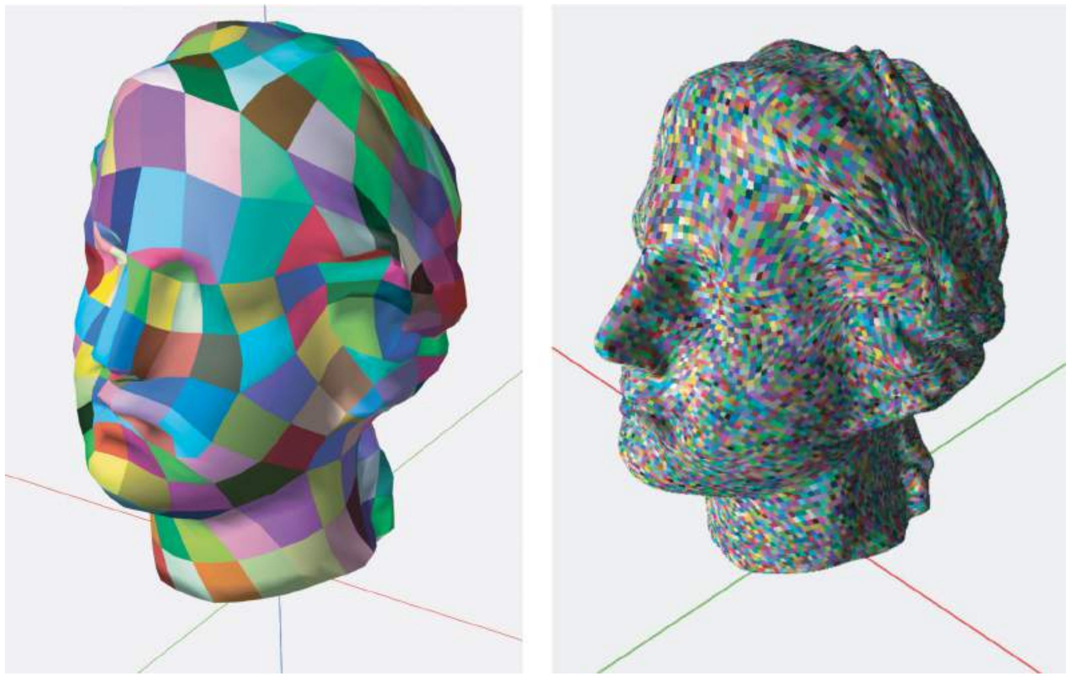

Figure 1. Surface decomposition: (left) object covered by B-spline surfaces, and (right) sub-patches after decomposition for $\beta = 0.3mm$ (model courtesy of Direct Dimensions, Inc.).

Table 1. Number of sub-patches as a function of the layer thickness.

λ	0.001	0.01	0.1	0.2	0.3	0.4	0.5
No. of patches	8,387,525	196,000	57,975	37,075	19,975	15,700	15,075

requires memory allocation and deallocation and if these operations are performed repeatedly (as in linked entities of complex data structures) the algorithm can be quite slow (it may not be all that well known but memory allocation can be more expensive than number crunching). Operating on a global data structure is out of the question so we introduced a dynamic list that is constantly updated for each slicer position.

First, we put all the sub-patches into a global storage so that surfaces can be selected for each slicer. Since the size of that list is not known until the decomposition is done, we estimate it as follows. Let the area of the bounding box of the entire object be A. The decomposition aims at generating surfaces that are about the size of the slicer band β. The estimated number of sub-patches and the patches per slicer are given below

$$N_{patches} = \frac{A}{\beta^2} \quad n_{patchonslicer} = \frac{N}{No\ of\ slicers}$$

Memory is allocated once based on the estimates above. If we run out, memory is reallocated in chunks, instead of one-by-one.

The critical part is the creation of the local list that is dynamically updated as the slicer moves up. For each slicer the method works as follows:

- The surfaces on the global list are sorted by their bounding boxes' minimum z-value.
- For each slicer we add surfaces from the global list if their bounding boxes intersect the slicing plane, and drop surfaces currently on the list if they no longer intersecting.
- Each time we add new surfaces, we check if the memory is sufficient or not. If not, we reallocate the memory and increase the current account by a larger chunk. Since the estimates are quite accurate, reallocation is rarely needed.

Figure 2 shows an example. The cut is slice number 104 of the head above (the neck area) produced a list of 279 patches. Notice how the sizes of the patches vary:

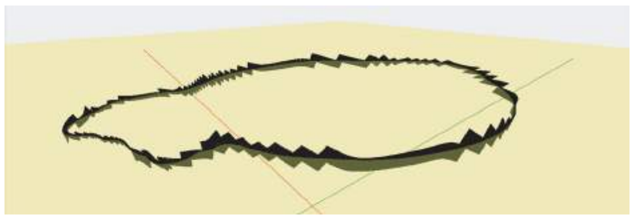

Figure 2. Surfaces on the local list corresponding to slice number 104.

high curvature areas generate small patches, whereas small curvature areas produce large patches.

A more complicated slice is shown in Figure 3: slice number 228 through the nose containing 582 patches. The yellow is the slicing plane and black depicts the surface patches.

The crux of the point-based algorithm lies in proper decomposition as well as the management of the dynamic list. If the surface is not properly decomposed, then some surfaces remain on the list too long while others drop out quickly. Figure 4 shows how the local list is utilized. The yellow line represents the number of surfaces for each slice. The green line shows the number of newly added surfaces and the red one illustrates the number of removed surfaces. It is quite revealing that the number of added and removed surfaces are roughly the same and they are proportional to the number of active surfaces on the list. It is also evident that adding and removing is not dependent of how complex the slice is, i.e. the surface decomposition does a good job regardless of the complexity of the object. A sample of numerical data is shown in Table 2 below.

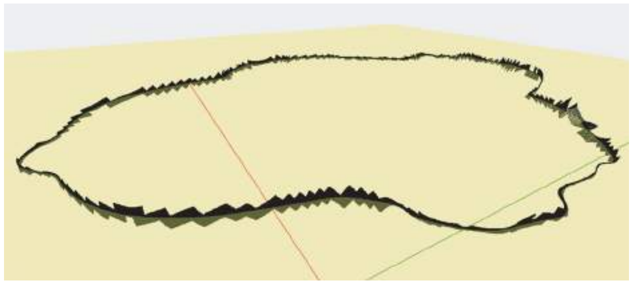
Figure 3. Surfaces on the local list corresponding to slice number 228.

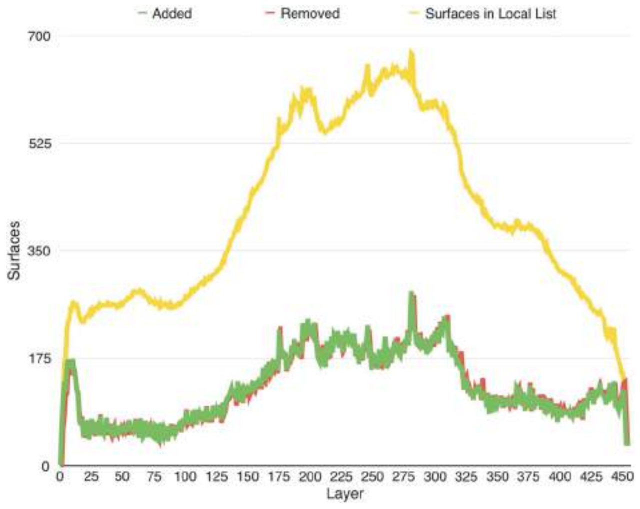

Figure 4. Surface patch utilization of the local list: the number of added and removed surfaces are shown in green and red, respectively, while the total active surfaces are marked in yellow.

Table 2. Sample slicer data: added, removed and total number of surfaces.

Slice	0	1	2	10	104	228	291	394	456
Added	1	64	107	174	71	220	206	81	36
Removed	0	1	60	159	70	208	197	87	36
Surfaces	1	64	111	269	279	583	589	334	36

5. Surface sampling

Once the surface patches are buffered into the dynamic list, they are ready to be sampled. Given a set of surfaces $S_k(u, v)$, $k = 0, \dots, K$, the required tolerance ε, we are seeking to obtain a sample of points Q_i, $i = 0, \dots, N$ on each surface so that for each point Q_j the following relationship holds: $|Q_j - Q_l| < \varepsilon$, where Q_l, $l = 0, \dots, M$ are the immediate neighbors of Q_j (the immediate neighbors form a polygon and the point Q_j is in the interior of that polygon).

There are a number of ways one can sample a surface given a tolerance. We discuss three different methods in this section.

5.1. Sampling based on derivatives

The simplest method involves computing a set of points at uniform parameter locations so that the resulting surface points, when triangulated, would not deviate from the surface in more than the allowed tolerance. To accomplish this, we need to determine the number of parametric locations (the number of sampling points). Assuming equal number of points in each direction, then according to [1], the formula is

$$n = \sqrt{\frac{1}{8\varepsilon}(M_1 + 2M_2 + M_3)} \quad M_1 = \max(S_{uu})$$

$$M_2 = \max(S_{uv}) \quad M_3 = \max(S_{vv})$$

While this seems like a simple solution, there are a number of problems:

- Computing bounds on derivatives is expensive especially if it needs to be done on a large number of surfaces (see the data in Table 2).
- The resulting point distribution is too high as this method oversamples the surface to achieve a global accuracy.
- Flat areas receive a small number of points whereas curved areas receive a lot more. That is, the point cloud can have large gaps in flat areas that is a no-no in 3-D printing.

We want a sampling technique that produces points in quasi-uniform manner, i.e. the distance condition above must be satisfied irrespective of the curvature of the surface.

5.2. Sampling based on divide-and-conquer

This method uses recursion to compute a set of points based on distance as well as flatness conditions. The outline of the algorithm is as follows:

- Subdivide the surface into four patches at the parametric midpoints.
- Push the four patches onto a stack.
- While the stack is not empty do:
  - Pop the stack
  - Check the surface for flatness; if flat, continue, otherwise subdivide and put the four sub-surfaces on the stack.
  - Check the six distances formed by the four corners; if they satisfy the accuracy requirement (and the flatness condition above), output the four corners, otherwise subdivide.
- Continue with the subdivision until no surfaces are left in the stack.

A few notes are in order:

- The algorithm needs to satisfy both the flatness condition as well as the distance condition. That is, the small patches should not deviate from a plane more than the tolerance ε to guarantee that there are no bumps in the middle of the patch. On the other hand, the corner points, forming the samples we are looking for, must satisfy the distance condition, i.e. for each point, all neighboring points must be within the tolerance.
- Care must be exercised when creating the output as a single point may be incident upon more than one sub-patch giving rise to multiple output. Sample points can be identified by their parameter pairs which can carry a flag to avoid multiple output.
- There are several ways one can check if a surface patch is flat enough (based on a strong convex hull property of splines [5]). The simplest is to compute a best-fitting plane to the four corners and check the distances of the control points from this plane. A more complicated one can compute a best-fitting plane to the entire set of control points and then check the distances.

While this algorithm is not the cheapest, it is tightly coupled with the geometry of the local surfaces and hence produces good quality results with a reasonable amount of overshoots, i.e. overly dense point distributions.

5.3. Sampling based on a hybrid method

This approach uses the ideas in the previous two methods but avoids expensive recursions. Let $\{L_u, L_v\}$ be the approximate extents of the small sub-patch to be sampled. Compute

$$n_u = \frac{L_u}{\varepsilon} \quad n_v = \frac{L_v}{\varepsilon}$$

number of points in each parametric direction at equal parametric locations. This generates a grid of points that, in turn, will be subjected to the flatness and distance conditions. That is, each quad

$$[Q_{i,j},\ Q_{i+1,j},\ Q_{i,j+1},\ Q_{i+1,j+1}]$$

in the grid computed at the parametric intervals

$$[u_i,\ u_{i+1}] \quad [v_j,\ v_{j+1}]$$

is considered as a small surface patch and is processed for flatness and for the six distances. If the tests fail, the quad is subdivided. Given that the sub-patches are quite small, subdivision is rarely needed. Table 3 and Figure 5 below show the number of points for various slicers. For one-tenth of a millimeter, the total number of points per slicer is quite small, in the neighborhood of under 50,000. For one-hundredth of a millimeter, this number jumps by two orders of magnitude to under about 2 million! This may sound too many, however, in today's computing power and memory storage 2 million points is a drop in

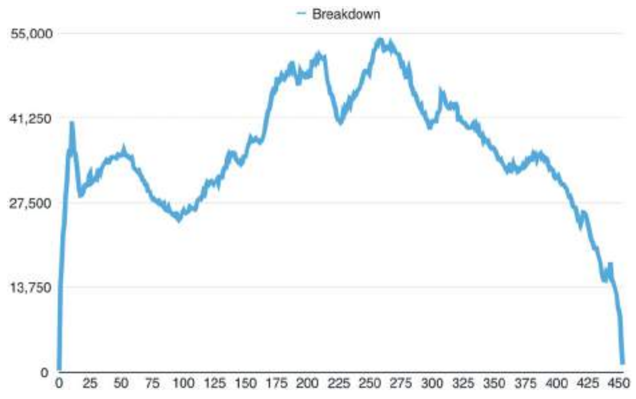

Figure 5. Number of points for each slicer computed at $\varepsilon = 0.1$.

Table 3. Number of sampling points based on recursive subdivision.

Slice	0	1	2	10	104	228	291	394	453
$\varepsilon = 0.1$	289	13986	17599	40687	26142	42227	43281	32972	1214
$\varepsilon = 0.01$	16641	1065024	1668899	1912570	500548	1212010	895429	516049	152100

the bucket (we live in the world of big data and 2 million points is in fact quite small).

A few improvements to the above algorithm are possible:

- Simple surfaces, such as planes and cylinders, do not require flatness and distance checking. All we need to do is sample them based on the tolerance. What it needs is knowing for each initial surface what their types are.
- We could reduce the thickness of the slicer band to produce small sub-patches. In the extreme, we could subdivide the surfaces to tiny pieces requiring no further sampling and checking at all. This, of course, would increase the data, however, if would eliminate further computational burdens.

Figure 6 below shows the result of sampling of the entire head model. The varying density is due to local surface characteristics and the angle of visualization.

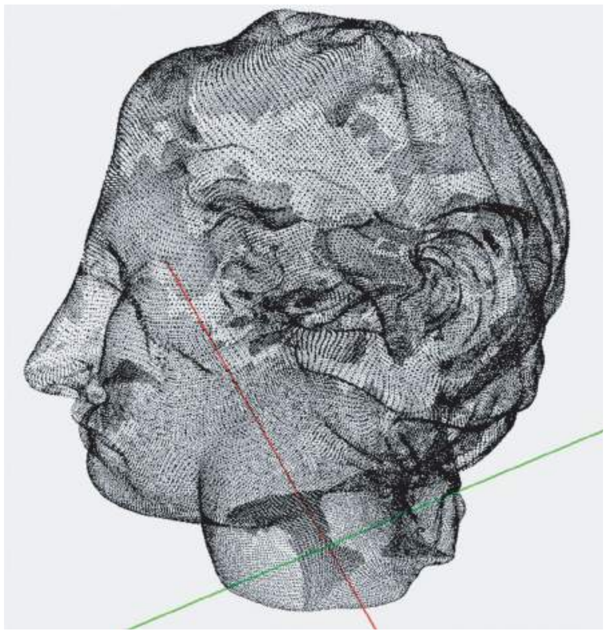

Figure 6. Point cloud computed at $\varepsilon = 0.5$.

6. Conclusions

This paper presented several ideas on how to prepare complex objects for slicing for 3-D printing. In order to avoid the myriad of problems with STL models and brain-numbing numerical issues, we elected to use the simplest of all geometric entities: the point. One of the biggest pros of the method is that it is extremely simple and nearly error free. The price to be paid for this simplicity is large data. However, living in the age of big data, this does not seem to be a burden. In fact, the algorithm runs smoothly on a plain vanilla laptop with off-the-shelf components. Each slice is processed in a matter of a few seconds, far outperforming the printer that takes many minutes to complete one layer. In other words, the slicer can be run in parallel with the printing process, instead of having to pre-process the entire object.

ORCID

Les A. Piegl http://orcid.org/0000-0003-0629-8496
Paul Rosen http://orcid.org/0000-0002-0873-9518
Khairan Rajab http://orcid.org/0000-0002-1260-5854

References

[1] Filip, D.; Magedson, R.; Markot, R.: Surface algorithms using bounds on derivatives, Computer Aided Geometric Design, 3, 1986, 295–311. http://dx.doi.org/10.1016/0167-8396(86)90005-1

[2] Ma, W.; But, W.-C.; He, P.: NURBS-based adaptive slicing for efficient rapid prototyping, Computer-Aided Design, 36, 2004, 1309–1325. http://dx.doi.org/10.1016/j.cad.2004.02.001

[3] Oropallo, W.; Piegl, L.: Ten challenges in 3D printing, Engineering with Computers, 32(1), 2016, 135–148. http://dx.doi.org/10.1007/s00366-015-0407-0

[4] Pandey, P.; Reddy, V.; Dhande, S.: Slicing procedures in layered manufacturing: a review, Rapid Prototyping Journal, 9(5), 2003, 274–288. http://dx.doi.org/10.1108/13552540310502185

[5] Piegl, L.; Tiller, W.: The NURBS Book, Springer-Verlag, New York, NY, 1997. http://dx.doi.org/10.1007/978-3-642-59223-2

[6] Piegl, L.; Rajab, K.; Smarodzinava, V.; Valavanis, K.: Fault-tolerant computing in a knowledge-guided NURBS environment, Computer-Aided Design and Applications, 6(6), 2009, 809–823. http://dx.doi.org/10.3722/cadaps.2009.809-823

[7] Rajab, K.; Piegl, L.; Smarodzinava, V.: CAD model repair using knowledge-guided NURBS, Engineering with Computers, 29(4), 2013, 477–486. http://dx.doi.org/10.1007/s00366-012-0264-z

[8] Shah, J. J.; Ameta, G.; Shen, Z.; Davidson, J.: Navigating the tolerance analysis maze, Computer-Aided Design and Applications, 4(5), 2007, 705–718. http://dx.doi.org/10.1080/16864360.2007.10738504

[9] Sikder, S.; Barari, A.; Kishawy, H.: Effect of adaptive slicing on surface integrity in additive manufacturing, Proc. ASME International Design Engineering Technical Conference, DETC2014-35559, 2014. http://dx.doi.org/10.1115/detc2014-35559

[10] Sun, S.; Chiang, H.; Lee, M.: Adaptive direct slicing of a commercial CAD model for use in rapid prototyping, International Journal of Advanced Manufacturing Technology, 34, 2007, 689–701. http://dx.doi.org/10.1007/s00170-006-0651-y

[11] Topcu, O.; Tascioglu, Y.; Unver, H.: A method for slicing CAD models in binary STL format, Sixth International

Advanced Technologies Symposium, Elazig, Turkey, 141–145, 2011.

[12] Wong, K.; Hernandez, A.: A review of additive manufacturing, International Scholarly Research Network, ISRN Mechanical Engineering, 2012, ID 208760.

[13] Yau, H.-T.; Kuo, C.-C.; Yeh, C.-H.: Extension of the surface reconstruction algorithm to the global stitching and repairing of STL models, Computer-Aided Design, 35, 2003, 477–486. http://dx.doi.org/10.1016/S0010-4485(02)00078-7

[14] Zhang, L.-C.; Han, M.; Huang, S.-H.: An effective error-tolerance slicing algorithm for STL files, International Journal of Advanced Manufacturing Technology, 20, 2002, 363–367. http://dx.doi.org/10.1007/s001700200164