www.cadanda.com
Computer-Aided Design AND APPLICATIONS

Interior Ball-Pivoting on Point Clouds for Offsetting Triangular Meshes

Tathagata Chakraborty[1]

[1]HCL Technologies, tathagata.chakr@hcl.com

Corresponding author: Tathagata Chakraborty, tathagata.chakr@hcl.com

Abstract. Triangular mesh offsets are useful for hollowing 3D models, generating tool paths, for clearance analysis and robot path planning among other applications. Smaller offsets are also used to account for shrinkage in casting and rapid prototyping. It is however difficult to compute mesh offsets that are simultaneously accurate, defect free and preserve sharp corners.

The present work includes a brief survey of common mesh offsetting techniques and describes our easy to implement sampling-based offset method that preserves sharp corners. Our approach consists of first approximating the mesh offset using a uniformly distributed point cloud that has been trimmed to eliminate self-intersections. The trimmed point cloud is then triangulated using the Ball-Pivoting Algorithm (BPA) modified to pivot the ball in the interior of the point cloud. We show that the BPA is a robust algorithm when applied to a uniform point cloud. While our method does not always guarantee defect-free meshes, it is only because we need a more robust method for estimating the normals at the trim boundary of sharp corners.

Keywords: offset, triangular mesh, point cloud, surface reconstruction
DOI: https://doi.org/10.14733/cadaps.2022.662-676

1 INTRODUCTION

Mesh offsets are useful for tolerance analysis, clearance testing, hollowing [18], modeling of coatings and etchings, and of shrinkage in casting and rapid prototyping, cutter path generation for NC machine tools [15], and path planning for robot motion. However, offsets in general are difficult to compute in two dimension and in 3D the problem is even more challenging [26].

There are three broad category of techniques for offsetting meshes. In the first category are techniques that obtain the offset by computing the Minkowski sum of the mesh with a sphere of offset radius [31]. A constructive geometry approach like this reduces the problem to computing a large number of booleans [30]. However computing 3D booleans is also not a trivial problem [3]. Moreover numerical instability is expected when computing booleans between the cylinder, spheres and prisms of the Minkowski sum since these will often intersect tangentially when calculating an offset.

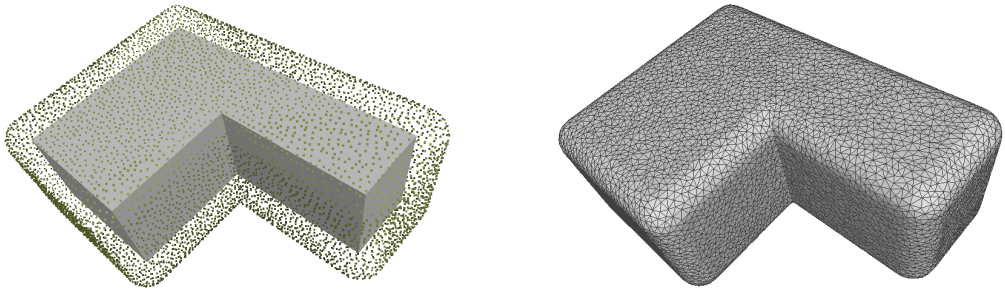


Figure 1: Left: Original mesh with trimmed point cloud at offset distance; Right: Triangulation of the offset point cloud using the modified BPA

The second category comprises techniques where either the triangles individually or the vertices independently are moved along a normal to create the offset. When the triangular faces are moved individually, they invariably create self-intersections in concave regions and gaps in the convex regions [14] [32]. These self-intersections then need to be trimmed and the gaps patched with cylindrical and spherical surfaces. Alternatively, where the vertices are moved independently while maintaining the topology of the original model, the offset approximation becomes increasingly inaccurate for larger offsets [28].

A third group of techniques are based on volumetric or surface sampling followed by surface reconstruction [24] [31] [19] [21]. Here the offset surface is approximated using a distance field or a point cloud and an appropriate surface reconstruction algorithm is used to triangulate the offset surface. In explicit surface reconstruction the local surface connectivity is estimated and the space between points interpolated to create triangles. Implicit approaches on the other hand require a signed distance field and use the Marching Cubes [22] or the Dual Contouring [12] algorithm or variants of these to triangulate the surface.

1.1 Our Approach in Brief

Our approach belongs to the third group and involves uniform surface sampling and explicit reconstruction. Uniform surface sampling is harder than it appears, and surface reconstruction offers a variety of possible explicit and implicit approaches. Explicit surface reconstruction techniques are often elegant and easy to implement, but they do not guarantee a watertight triangulation. Implicit techniques on the other hand guarantee watertight meshes, but they are not very accurate [9] - their accuracy depends on the voxel resolution used which is limited by available memory. In spite of this, implicit surface reconstruction techniques are more popular because they robustly handle noise and non-uniformity typically present in device acquired point-cloud data [13].

We use an explicit surface reconstruction technique called the Ball-Pivoting Algorithm (BPA) [2] which is ideal for our artificially constructed point cloud that contains no noise. Our primary interest was in computing relatively small, accurate and NC machinable positive offsets to account for shrinkage during casting. It was therefore necessary to compute and preserve sharp concave corners accurately. It is generally more difficult to preserve sharp features using implicit approaches [17] [12].

In brief, we first compute a uniform point cloud approximating the mesh offset, then trim it to remove self-intersections and finally use the BPA to triangulate the trimmed point cloud (see Fig. 1). The rest of the paper describes the process in detail and is organized as follows. We start with an brief survey of mesh offsetting techniques in Sec. 2. In Sec. 3 we describe how to generate a uniform point cloud approximation of

the offset, including how to trim it of self-intersections and how to compute extrapolated trim boundary points for accurately reconstructing sharp concave corners. Sec. 4 describes the Ball-Pivoting Algorithm (BPA), our modifications and some limitations. Lastly in Sec. 5 we analyze the complexity and performance of the algorithm. Sec. 6 presents and discusses some results. Sec. 7 concludes our discussion with directions for future research.

2 BRIEF SURVEY OF EXISTING TECHNIQUES

Offsets of synthetic curves and surfaces have been investigated since the early days of CAD/CAM (see [26] and [23] for early surveys). As noted in [26] computing the offset is not easy and most research has been focused around finding good approximations. Early methods used a combination of analytic and iterative techniques, but we also see investigations into discrete and sampling-based methods (see [16], [6] and [27]). Sampling and subdivision based techniques have remained essential for computing offsets ever since.

Although a triangular mesh is already a sampled and discretized surface it is thereby still no easier to offset. Negative offsets can be used to hollow 3D models to make them faster to print using rapid prototyping. The idea in an early solution [18] was to move the vertices inward by the offset distance along their average normal to create the negative offset. This would often generate self-intersections. However, instead of resolving these intersections in 3D, the model is sliced along the material deposition direction and the self-intersections are then corrected in 2D which is much easier to do. Similar techniques are used for resolving self-intersection when computing offset-based tool paths (see [15]). These techniques however cannot be readily extended to compute uniform offsets and are therefore limited in their application.

A more complicated technique that allows for both positive and negative offsets moves the vertices in the direction of the average normal [28] or along multiple normals [14]. In [28] the topology is preserved as is by moving the vertices independently along the average normal. No gaps and only rare self-intersections are thereby created in the offset but the offset accuracy decreases rapidly for larger offset values. Somewhat differently in [14] each convex vertex is offset along multiple normals creating gaps which are then patched with cylindrical and spherical surfaces. In both cases however the self-intersection problem is left unresolved and delegated to downstream processes.

Volumetric and sampling based methods have become increasingly popular because they avoid the need for complex intersections and heuristics. The offset of a 3D polygonal model can be construed as the Minkowski sum of the model with a sphere of offset radius. Computing the Minkowski sum can be very expensive, however in [31] a fast approximation of the Minkowski sum is computed by combining multiple signed distance fields over a voxel grid and then the Marching Cubes iso-surface extraction algorithm is used to create the offset mesh. In [5] a voxel grid and a point sampling technique are used to identify surface voxels in the offset and then the Dual Contouring algorithm is used to reconstruct the offset mesh. In [21] a similar technique is used where the space is sampled in a uniform grid, intersections between the offset surface and the grid edges are computed and a modified version of the Dual Contouring algorithm is used to extract the offset mesh surface. The method presented in [4] is also similar.

The techniques in [31] [5] [21] [4] all rely on an implicit surface reconstruction technique. Reconstruction from an implicit representation gives a watertight model, although it may sometimes contain self-intersections. Implicit techniques however take up a lot of memory and thus do not scale well with the size of the model. Larger voxel grids and finer octrees require exponentially more memory and even when such memory is available the algorithm can become very slow due to memory access overheads.

Explicit surface reconstruction techniques on the other hand are not popular because they do not work well on noisy and non-uniform point cloud data. An explicit technique can however handle very large models without running out of memory. Unfortunately, not all explicit reconstruction technique are robust and they usually cannot guarantee watertight meshes. However we show that the BPA, which is an explicit reconstruction algorithm, performs robustly given a uniformly distributed point cloud as input.

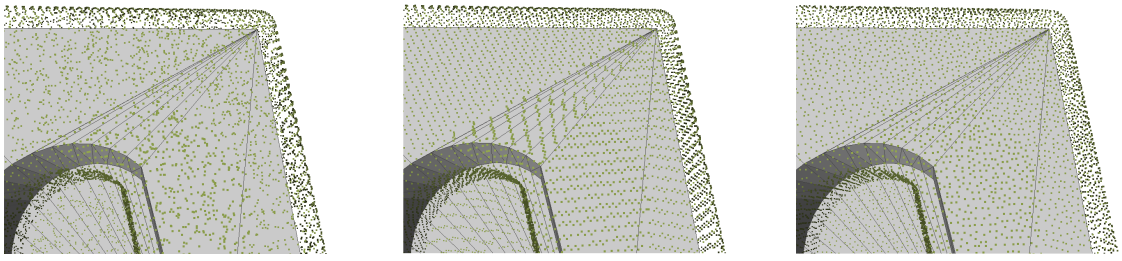


Figure 2: Left: Point cloud using random point distribution; Center: Point cloud using pseudo-random point distribution; Right: Point cloud using the Poisson-disk decimator

3 POINT-CLOUD GENERATION

Point-clouds are typically acquired using 3D scanners and sometimes generated from the information extracted from 2D depth cameras. In our case however the point-cloud data is artificially generated and therefore has very different characteristics. Point clouds acquired using 3D scanners have a lot of noise and non-uniformity. Preprocessing techniques to remove the noise from the data are frequently employed, and a reconstruction algorithm is chosen which it is invariant to some amount of noise present in the data.

Our offset point cloud is generated by sampling the surface of the model directly and moving the points normal to the surface. This creates gaps in the convex corners which are patched using point cloud approximations of cylindrical and spherical surfaces. These patch surfaces are created conservatively to cover a surface area somewhat larger than the actual gap since at complex corners it is difficult to accurately estimate the extent of the required patch surface beforehand. In concave regions and near the boundaries where the patch surfaces meet there are therefore self-intersections. These self-intersections are identified and trimmed to create an accurate and noise-free point cloud.

Traditionally mesh sampling has been used for remeshing and to reduce the dimension of the problem for statistics and machine learning algorithms. Our implementation required a uniform mesh sampling technique with feature preservation and was inspired from these studies [8] [11].

3.1 Uniform Point Cloud Generation

We use the Ball-Pivoting Algorithm (BPA) to triangulate the offset point cloud [2]. While the BPA is quite elegant and simple, it has not been widely investigated possibly because the algorithm was, until recently, under the shadow of a patent which discouraged broad usage. Our early experiments showed that the BPA works well given a fairly uniform point cloud. While it is theoretically possible execute the BPA iteratively using balls of increasing radii to triangulate a non-uniform point cloud, practically this is inadvisable because it is difficult to predetermine the number and range of ball radii required for these iterations. Highly non-uniform point clouds may require too many iterations of the algorithm to triangulate fully.

Generating a globally uniform point distribution however is harder than it initially appears. Real-world 3D models are non-uniformly tessellated which makes it difficult to ensure a globally uniform point density when sampling each triangle independently. Locally this is because the boundary between neighboring triangles get approximated twice and globally because the model is likely to contain very large triangles as well as small and sliver triangles, which results in regions that have a much higher concentration of boundary edges and therefore also a higher density of points.

Generating a uniform point distribution inside a triangle is equally difficult. A random sprinkling of points inside a triangle is far from uniform and not ideal for a triangulation algorithm (see Fig. 2). A more appropriate

method is to use a quasi-random distribution of points. Quasi-random distributions [29] give good results for most triangles but show aliasing effects on narrow triangles. Another method is to overlay a regular grid of points on top of the triangle and discard the points that fall outside the triangle. However, the triangulation resulting from such a distribution looks sterile. Triangulation of a quasi-random distribution on the other hand is aesthetically more pleasing.

The number of points generated for a triangle is proportional to its area. It is therefore possible that no points are generated for very small triangles. This is not a problem for an isolated triangle, but when sizeable regions are covered by a large number of small or sliver triangles, the point distribution in these regions may become sparse. We therefore also sample the triangle edges and include the vertices of each triangle in the point cloud.

3.2 Poisson-disk Sampling

To get a globally uniform point distribution we first generate a denser point cloud and then decimate it using Poisson-disk sampling [7] (see Fig. 2). We implemented a randomized parallel version of the algorithm which is similar to the constrained sampling method described in [8]. We draw random points from the original point cloud, keep that point and remove the points in the r-neighborhood of the point where r is the desired resolution of the point cloud. The random selection of points is necessary to avoid any aliasing effects and it produces aesthetically pleasing triangulation. Our implementation of the algorithm is given in pseudo-code below.

Algorithm 1: Parallel Poisson-disk sampling

Input: original point list and decimation resolution r
Output: decimated points list
while *points remaining in the original list* **do**
  pick a random point from the list;
  find the points in the r-neighborhood of this point;
  add the picked random point to the decimated point list;
  remove the picked point and the points in the r-neighborhood from the original list;
end

Generating the dense point cloud takes time proportional to the surface area of the mesh. For every triangle we generate quasi-random points proportional to the area of the triangle. The number of points to generate per unit area is determined based on the maximum chord tolerance that is acceptable in the output offset mesh. For a typical mesh several million points are generated, and often the time taken to generate the points is overshadowed by the time taken to allocate and access the memory for storing them. Memory allocation and access can be made more efficient by predetermining the total memory required and preallocating it in right sized chunks. The memory footprint of our method is driven by the size of the point cloud but even for large models it rarely exceeds the gigabyte mark.

Poisson-disk decimation requires $O(n \log m)$ time. During decimation we have to go through a subset of points in the point cloud and for each point find the points in its r-neighborhood which takes on an average $O(\log m)$ time, where m is the size of the mesh space when discretized using a resolution r.

3.3 Trimming Self-Intersections

Before Poisson-disk decimation, we trim the self-intersections in the point cloud. Additionally, we also explore the neighborhood of invalid points (i.e. points in the self-intersecting regions) to find points on the boundary of the trimmed surfaces (see Sec. 3.4). To trim the self-intersections we find the closest distance of each point to the input mesh. To do this efficiently we overlay the model space with a offset distance resolution voxel

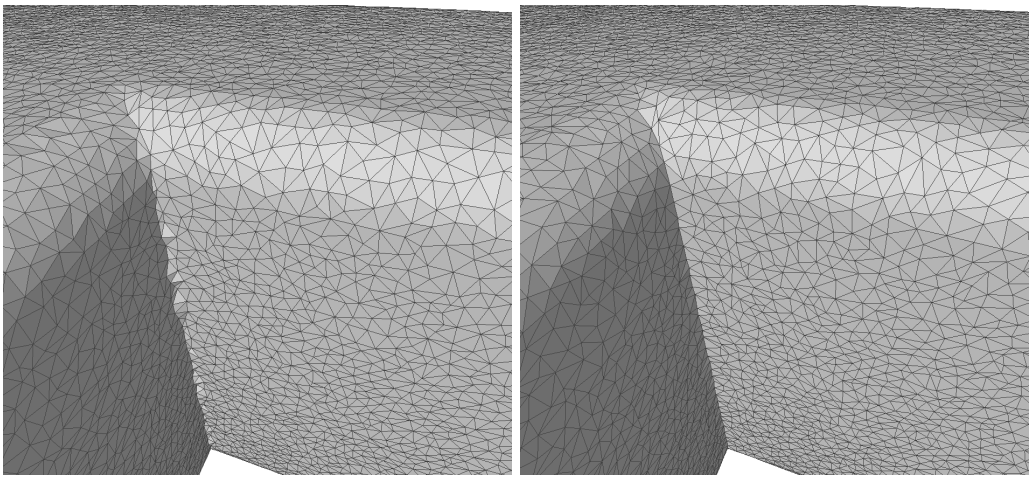


Figure 3: Left: Triangulation with no trim points at sharp corners; Right: Triangulation with trim points

grid, where each voxel stores the list of triangles intersecting with the voxel. For every point in the offset point cloud, we find the voxel in which it contained, and then find all the triangles passing through the 1-voxel neighborhood of this voxel. Next we compute the closest distance of the offset point under consideration from the set of the triangles found passing through the 1-voxel neighborhood. If the closest distance is less than the offset distance we discard the point.

This requires $O(n \log m)$ time. We have to go through all the n points to determine if they are valid and for each point find the voxels in the neighborhood of the point which takes on an average $O(\log m)$ time where m is the number of voxels in the grid. The algorithm is given in pseudo-code below where both the for loops can be executed in parallel.

Algorithm 2: Trimming Self-Intersections

```
Input: original point list with self-intersections and voxel grid
Output: point list without self-intersections
for each point in the original point list do
    find the triangles in the 1-voxel neighborhood of this point;
    for each triangle found in the neighborhood do
        find the minimum distance of the point to the triangle;
        if minimum distance is less than offset distance then
            remove the point from the list;
        end
    end
end
```

Trimming self-intersections and finding exact boundary points are the most computationally expensive steps in our approach. Most of the time is taken up in computing the minimum distance of a point from the mesh. In practice most of the points are already exactly on the offset surface and only a much smaller subset of the points are within the self-intersecting regions. For points that are already on the offset surface it is not only redundant to compute the minimum distance from the mesh but it is also computationally more expensive since for these points we need to compute the distance to each and every one of the neighboring triangles before we can be sure that they are valid.

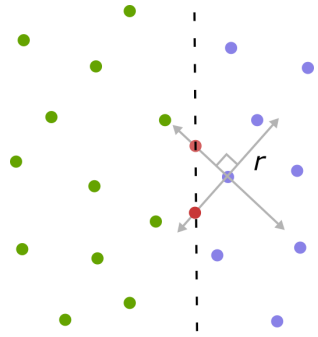


Figure 4: Finding trim boundary points (red) by binary searching along orthogonal directions from invalid points (purple)

A quick method of filtering out the points that are already on the offset surface is to render the points as surfels [25] from different directions with the depth buffer and depth testing enabled and then to note which points get rendered. The points must be rendered with a radius slightly greater than the point cloud resolution to ensure that the self-intersection points cannot be seen through the gaps. The points which are rendered on the framebuffer can then be filtered out as valid. The percentage of valid offset points thus detected will depend on the geometry of the model and the number of directions from which the point cloud is rendered. In our experiments around 70-80% of the valid points are detected when the point cloud is rendered from the positive and negative standard axis directions.

3.4 Finding Exact Boundary Points

During trimming we explore the neighborhood of each invalid point to see if we can find a valid point nearby that lies on the trim boundary. Finding points on the trim boundary is essential for preserving sharp concave corners in the offset. Without these trim boundary points and a surface reconstruction method which includes them, the triangulation in the concave corners will be visibly jagged (see Fig. 3). Jagged corners are problematic in domains like manufacturing and especially for tool path generation.

Algorithm 3: Finding Exact Trim Boundary Points

```
Input: original point list with self-intersections
Output: boundary trim point list
for each point in the original point list do
    find if the point is valid;
    if point in not valid then
        find two orthogonal directions on the plane tangential to the offset at the point;
        create four segments of resolution length along these two directions;
        for each segment with a valid end point do
            try to find nearest valid point on the segments using binary search;
            if valid point found then
                add point to the boundary trim point list;
            end
        end
    end
end
```

As described in the algorithm above, to find these trim points, we do a binary search along multiple line or arc segments passing through each invalid point to discover the nearest valid points if any. If found these valid points are appended to the point cloud and marked as trim points. In our current implementation we choose two arbitrary orthogonal directions and mark off four small segments of length equal to the point cloud resolution on both sides of these orthogonal directions. If the end points of any of these segments turns out to be valid, we do a binary search for the nearest valid points along that segment (see Fig. 4). For invalid points that originate from the original triangles we extend line segments lying on the offset plane and along the orthogonal directions. For invalid point originating from cylindrical patch surfaces we extend a line segment along the cylindrical axis and extend an arc orthogonal to the line both lying on the cylindrical surface. Similarly for invalid points originating from spherical patch surfaces.

Only invalid points near the trim boundary are useful for finding valid trim boundary points. We can therefore improve performance by filtering out invalid points which are too far from the offset envelop. Our method also finds trim boundary points on convex corners which result in trim points on smooth boundaries. These trim points are not useful for the preservation of sharp corners. Another possible improvement therefore is to restrict the computation of trim boundary points to those entities (triangles, edges or vertices) among whose connected neighbors we can find at least one concave entity. However, this would fail to capture trim points on concave corners created by globally distant intersections.

3.5 Limitations

We would like to apply Poisson-disk sampling only on the geodesic neighborhood of a point on the offset surface, however our implementation removes points from the volumetric r-neighborhood. In narrow channels and small holes therefore, where two or more surfaces are close together, the Poisson-disk sampling method can remove unintended points during decimation. A partial solution to this problem is to only remove those points from the r-neighborhood whose normals have a positive dot product with the normal of the random point selected for preservation. In spite of this, sometimes narrow channels and small holes may have ugly triangulations due to the sparseness of the point cloud in these regions. A more adaptive sampling technique needs to be investigated to resolve these issues.

4 TRIANGULATION WITH MODIFIED BPA

Point cloud reconstruction has been extensively studied (see [20] [1] [13] for recent surveys). Different techniques can be used depending on how unevenly distributed, noisy or sparse the point cloud is. We chose the Ball-Pivoting Algorithm (BPA) because of its robustness and simplicity. Lack of available data on the effectiveness of the BPA was a deterrent but it did not dampen our curiosity for exploring a robust technique the patent on which had just expired. More importantly, the BPA had heuristic steps which could be tweaked and extended.

In the sections below we describe our implementation of the BPA which is somewhat different from that proposed in the original paper [2] and in part inspired from the implementation detailed in [10]. We explain our modifications to the BPA and highlight limitations with the method. We also briefly describe a parallel implementation of the BPA.

4.1 Ball-Pivoting Algorithm

We start by selecting a random seed point and then find a point nearby which is not already interior to the triangulation (i.e. either it is outside the triangulation or on its boundary). These two points form the seed edge about which we pivot a ball of a predefined radius, and try to find another point in the neighborhood of the seed edge such that we can get the ball on rest on these three points. If certain conditions are met then we create a triangle using these three points and add it to our triangulation.

We maintain a half-edge data-structure to store the triangulation. A half-edge data-structure enables us to quickly check whether a vertex or an edge is interior to the triangulation or on the boundary. In general for a closed solid all pivots must succeed. However sometimes a pivot fails because there are no nearby points close enough to create a triangle with the ball radius with which we are pivoting. A range of balls are therefore used and the BPA is executed repeatedly on the failed pivot edges using balls of increasing radii till there are no edges left to pivot on.

Algorithm 4: The Ball Pivot Algorithm

```
Input: offset point cloud
Output: triangulated offset mesh
find a seed pivot edge and insert it into the stack;
while there are pivot edges remaining in the stack do
    pop an edge from the stack;
    if edge is not an interior edge then
        find points in the 2r-neighborhood of this edge;
        for each point in the neighborhood do
            if point is not an interior point in the triangulation then
                see if we can rest the ball on these three points;
                if we can rest the ball then
                    check if there are any other points inside the rested ball;
                    if no points are inside the ball then
                        create a triangle using these three points and;
                        compute the dot product of the triangle normal and the vertex normals;
                        if the dot produts are all positive then
                            add the triangle using these three points to the triangulation;
                        end
                    end
                end
            end
        end
    end
end
```

The details of our ball pivot implementation are shown in the algorithm above. Given the edge about which to pivot, we find its mid-point first, and then find points in the $2r$-neighborhood of this point (where r is the resolution of the point cloud). We then sort these points in ascending order of distance from the pivot mid-point. We iterate over this sorted list of points till we find a point on which we can rest a sphere of the ball radius, such that no other points in the neighborhood lie inside this sphere.

While iterating through the list of points we check if the point is already a part of the triangulation, in which case we additionally check if the point is an interior point or a point on the boundary of the triangulation. In case the point in an interior point (we can find this out by checking if all the coedges connected with the vertex for this point have partner coedges), we ignore the point. If we find a non-interior point on which we can rest the ball then we create a new triangle with the pivot edge and the found point. The edges of the this triangle become new potential pivot edges, which we maintain in a stack. We keep pivoting on the edges from the stack edges till the stack is empty. While popping edges from the stack we check if the edge is interior to the triangulation (we can do this by checking if both the coedges of the edge are present) and only pivot on the edge if the edge is boundary edge. This edge check can also be performed before pushing an edge onto the pivot edge stack.

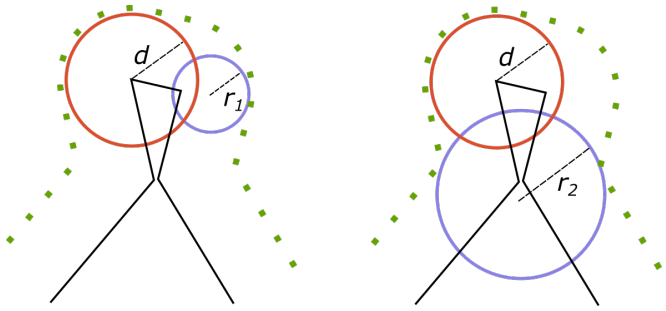


Figure 5: BPA ball radius r (purple) must be less than offset distance d so that it can access all points when rolling in the interior of the point cloud

4.2 Modifications to the BPA

When checking whether a ball of a given radius can be balanced on the three points, two other checks are performed. First we also check if the dot products of the normal of the triangle and the vertex normals of the three points of the triangle are all positive. Secondly we check that no other neighboring point is inside the sphere when it is resting on the three points. Both these checks may fail, especially at sharp concave corners. Our initial attempts at preserving sharp concave corners therefore revolved around heuristic relaxation of these checks in the neighborhood of such corners. However it was difficult to handle all possible cases using such heuristics.

Our main insight into resolving this issue was to roll the ball in the interior of the point cloud instead of outside it as it typically done. This converts sharp concave corners into convex corners, which are easier to handle. This choice forces us to use a ball radius less than the offset distance (see Fig. 5), which in turn requires us to use a higher point cloud density, but in practice this is not a problem. Rolling the ball inside the point cloud also enables us to properly triangulate narrow channels and gaps which the standard BPA, and most other surface reconstruction technique for that matter, cannot easily handle.

Rolling the ball inside the point cloud however doesn't completely resolve the issue. The BPA can handle sharp convex corner only if the points at the trim boundary forming the sharp corner have properly averaged and accurate normals. Otherwise the positive dot product check of the triangle normal with the vertex normals is likely to fail resulting in gaps and irregular triangulation in these regions. We try to accurately estimate the normal of the trim point by setting it to the average of the neighborhood point normals, taking care to count approximately equal number of points in the various surrounding directions to eliminate any bias. If the normals are estimated accurately then it is impossible for a vertex normal to have a negative dot product with the fitted triangle. However in very sharp corners accurate normal estimation is difficult and this results in small gaps in the triangulation.

4.3 Problems with Very Sharp Corners

Very sharp corners typically occur when two opposite offset surfaces just meet each other, for example in the case of a narrow channel with width twice the offset distance (see Fig. 6). In these cases it becomes very difficult to estimate the normal of the trim points on the boundary of the intersecting surfaces based solely on the local neighborhood of the point.

Normal estimation in general is a difficult problem. Most of the limitations in our approach are caused by erroneous normal estimation in such borderline cases. Inaccurate normals are a serious problem since they lead to small gaps and jagged triangulation in the offset mesh. Often the gaps are small enough that a triangulation

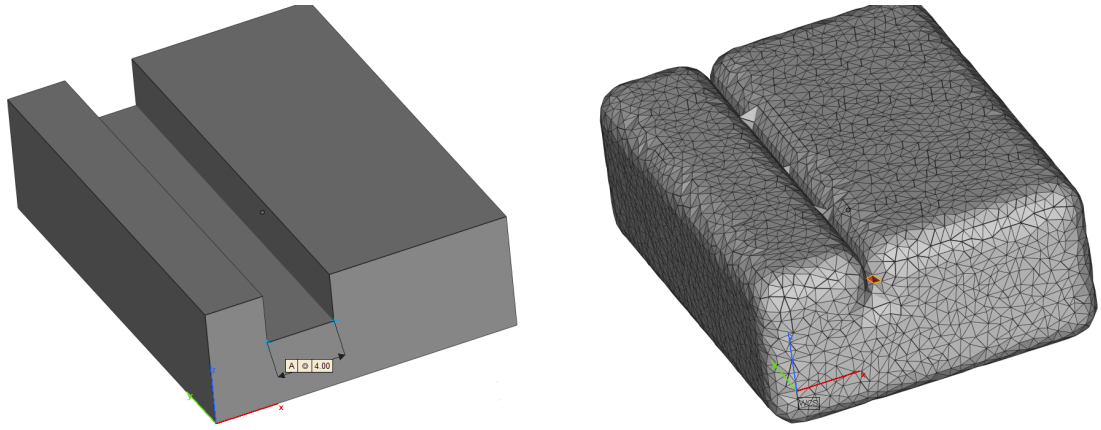


Figure 6: Left: Model with a narrow channel of width 4; Right: Offset at 2 units with problems resulting from inaccurate normal estimation

can be forced in these regions using a simple hole patching algorithm. However, the jagged triangulations are more difficult to fix. We are still investigating how to robustly resolve this issue.

4.4 Parallel BPA

The BPA can be parallelized by splitting the point cloud into multiple exclusive regions along its major axis. The number of split regions will depend on the number of available processors cores. The BPA can then be executed on each of these regions in parallel to generate as many triangulations. Since these regions will not be watertight, in each case we will also end up with a list of failed pivot edges. The multiple half-edge triangulation data structures are easily merged into one. The BPA is then run on all the failed pivot edges with the whole of the point data to patch the gaps between the different triangulations. In case we have a large number of parallel processors this two step process can be further broken down into a multi-step process where each intermediate step involves a parallel merging of two or more adjacent split regions.

With the parallel BPA we can sometimes get very narrow triangles and triangular gaps in the joins between the split triangulations. These narrow gaps require a much larger ball to triangulate. To patch these we need a separate triangular hole patching step which can force the triangulation in these gaps. However note that this problem does not occur with the serial version of the algorithm.

4.5 Negative Offsets

We were only interested in computing relatively small positive mesh offsets to account for shrinkage in casting and for generating NC machinable tool paths for such cast products. Our method therefore works only for computing positive offsets. The whole idea of rolling a ball in the interior of the point cloud which converts hard to triangulate sharp concave corners into convex corners, does not hold when computing negative offsets. Negative offsets will contain both sharp convex and concave corners no matter on which side of the point cloud we roll the ball. One possible way of extending the BPA so that it preserves both sharp concave and convex corners would be to roll the ball on both sides of the point cloud and in each side take care to avoid rolling over sharp concavities. While this seems theoretically possible, such an approach will surely involve several more and different challenges.

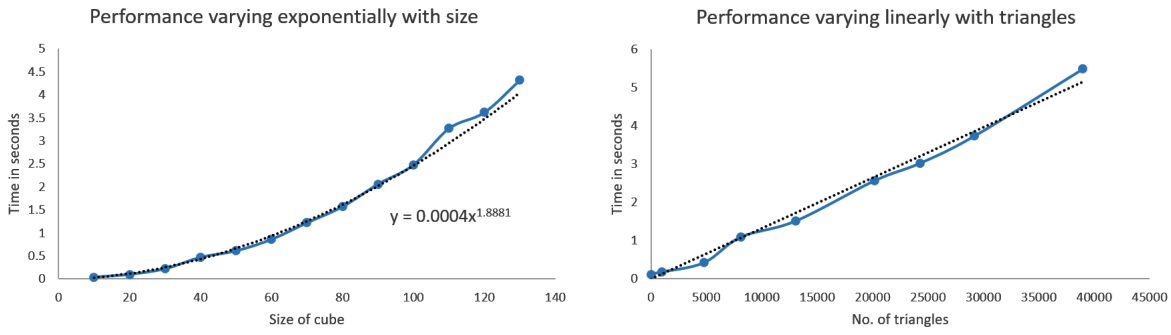


Figure 7: Left: Performance scaling with model size with number of triangles fixed; Right: Performance scaling with number of triangles with model size fixed

5 PERFORMANCE CHARACTERISTICS

In general sampling-based methods do not scale well with model size. As the size of the model increases the point cloud density required to maintain the same tolerance increases quadratically (see Fig. 7). Even with a low overall complexity of $O(n \log n)$, but with n increasing quadratically the algorithm slows down significantly with increasing model size. However since all the steps in our approach can be easily parallelized, we can significantly improve the runtime of the algorithm when executing on modern multi-core processors. This brings down the overall time taken to a reasonable amount for moderately sized models, but it is still too large for integration into interactive applications.

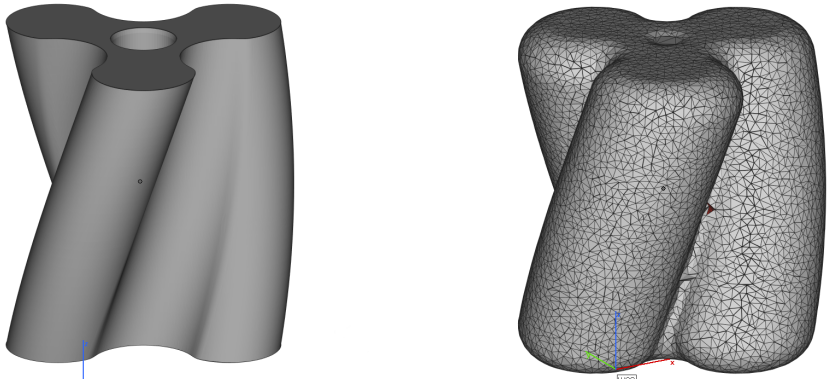
The algorithm scales only linearly with the number of triangles (see Fig. 7), but in our case this is still bad. Typically sampling-based methods should be invariant to the density of the underlying representation, since once the original surface has been sampled there is rarely any need to refer back to it. We however need to access the original mesh surface to trim the self-intersection in the point cloud and to compute the trim boundary points and this creates an overall linear dependency. The additional linear dependency compounds the performance problem since an increase in the size of a model is usually accompanied by an increase in triangle count.

6 RESULTS

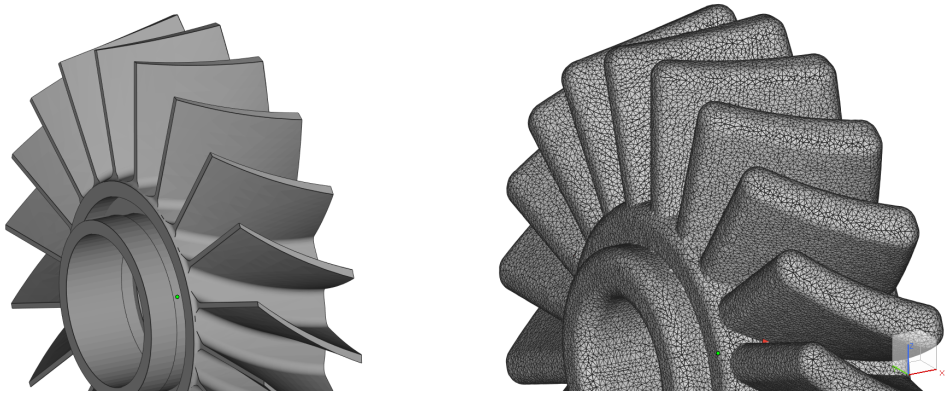
Fig. 8 shows some results from our method. Note that our method produces aesthetically pleasing Delaunay-like triangulation. Some minor gaps at concave corners can be seen (marked in red) in both cases and these are due to inaccurate normal estimation and are typically representative of the type of problem encountered with our method. The time taken is half a minute in both cases. In general, given a fixed tolerance and thus a fixed point cloud resolution, the time taken by the algorithm can vary between a few seconds going up to a few minutes for large complicated models.

7 CONCLUSIONS

Robustly offsetting triangular meshes is a difficult problem. A Minkowski sum approach can work but requires a robust mesh boolean algorithm, which is equally difficult to implement. Commonly used are volumetric approaches where the offset is approximated by a signed distance field and reconstructed using an implicit surface reconstruction algorithm. However the accuracy of these volumetric approaches depends on the voxel resolution and increasing accuracy requires exponentially more memory. In this paper we have described an unorthodox technique for computing accurate offsets of triangular meshes using the relatively uncommon



(a) Around 41k triangles, size 22x21x23, time taken to offset by 3 units was around 36 seconds



(b) Around 57k triangles, size 34x113x113, time taken to offset by 3 units was around 20 seconds

Figure 8: Some examples of offset on non-trivial parts

Ball-Pivoting Algorithm for surface reconstruction. We have shown how the method can be extended to overcome inherent limitations of the algorithm and to preserve sharp concave corners in the offset. Our results indicate that the method performs robustly given a uniform point cloud with reasonably accurate normals at trim boundaries. Our method requires much less memory as compared to volumetric techniques, however performance remains a problem for very large models and a more robust normal estimation technique needs to be found.

ACKNOWLEDGEMENTS

This research was undertaken as a part of the CAMWorks project. CAMWorks is a popular CAM software used by small and medium sized workshops and industries. Thanks to Vivek Govekar and Nitin Umap, product and project managers respectively for CAMWorks for providing us the opportunity to explore and prototype the ideas presented here. Thanks to Swadhin Bhide for suggesting various research directions, and to Hariharan Krishnamurthy for reviewing several drafts of the paper.

Tathagata Chakraborty, http://orcid.org/000-0000-2752-2533

REFERENCES

[1] Berger, M.; Tagliasacchi, A.; Seversky, L.; Alliez, P.; Levine, J.; Sharf, A.; Silva, C.: State of the art in surface reconstruction from point clouds. In Eurographics 2014-State of the Art Reports, vol. 1, 161–185, 2014.

[2] Bernardini, F.; Mittleman, J.; Rushmeier, H.; Silva, C.; Taubin, G.: The ball-pivoting algorithm for surface reconstruction. IEEE transactions on visualization and computer graphics, 5(4), 349–359, 1999. http://doi.org/10.1109/2945.817351.

[3] Chen, M.; Chen, X.Y.; Tang, K.; Yuen, M.M.: Efficient boolean operation on manifold mesh surfaces. Computer-Aided Design and Applications, 7(3), 405–415, 2010. http://doi.org/10.3722/cadaps.2010.405-415.

[4] Chen, Y.; Wang, C.C.: Uniform offsetting of polygonal model based on layered depth-normal images. Computer-aided design, 43(1), 31–46, 2011. http://doi.org/10.1016/j.cad.2010.09.002.

[5] Chen, Y.; Wang, H.; Rosen, D.W.; Rossignac, J.: A point-based offsetting method of polygonal meshes. ASME Journal of Computing and Information Science in Engineering, 1–21, 2005.

[6] Choi, B.K.; Park, S.C.: A pair-wise offset algorithm for 2d point-sequence curve. Computer-Aided Design, 31(12), 735–745, 1999. http://doi.org/10.1016/S0010-4485(99)00060-3.

[7] Cline, D.; Jeschke, S.; White, K.; Razdan, A.; Wonka, P.: Dart throwing on surfaces. In Computer Graphics Forum, vol. 28, 1217–1226. Wiley Online Library, 2009. http://doi.org/10.1111/j.1467-8659.2009.01499.x.

[8] Corsini, M.; Cignoni, P.; Scopigno, R.: Efficient and flexible sampling with blue noise properties of triangular meshes. IEEE transactions on visualization and computer graphics, 18(6), 914–924, 2012. http://doi.org/10.1109/TVCG.2012.34.

[9] De Araújo, B.R.; Lopes, D.S.; Jepp, P.; Jorge, J.A.; Wyvill, B.: A survey on implicit surface polygonization. ACM Computing Surveys (CSUR), 47(4), 1–39, 2015. http://doi.org/10.1145/2732197.

[10] Digne, J.: An analysis and implementation of a parallel ball pivoting algorithm. Image Processing On Line, 4, 149–168, 2014. http://doi.org/10.5201/ipol.2014.81.

[11] Geng, B.; Zhang, H.; Wang, H.; Wang, G.: Approximate poisson disk sampling on mesh. Science China Information Sciences, 56(9), 1–12, 2013. http://doi.org/10.1007/s11432-011-4322-8.

[12] Ju, T.; Losasso, F.; Schaefer, S.; Warren, J.: Dual contouring of hermite data. In Proceedings of the 29th annual conference on Computer graphics and interactive techniques, 339–346, 2002. http://doi.org/10.1145/566654.566586.

[13] Khatamian, A.; Arabnia, H.R.: Survey on 3d surface reconstruction. Journal of Information Processing Systems, 12(3), 2016.

[14] Kim, S.J.; Lee, D.Y.; Yang, M.Y.: Offset triangular mesh using the multiple normal vectors of a vertex. Computer-Aided Design and Applications, 1(1-4), 285–291, 2004. http://doi.org/10.1080/16864360.2004.10738269.

[15] Kim, S.J.; Yang, M.Y.: Triangular mesh offset for generalized cutter. Computer-Aided Design, 37(10), 999–1014, 2005. http://doi.org/10.1016/j.cad.2004.10.002.

[16] Kimmel, R.; Bruckstein, A.M.: Shape offsets via level sets. Computer-Aided Design, 25(3), 154–162, 1993. http://doi.org/10.1016/0010-4485(93)90040-U.

[17] Kobbelt, L.P.; Botsch, M.; Schwanecke, U.; Seidel, H.P.: Feature sensitive surface extraction from volume data. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 57–66, 2001. http://doi.org/10.1145/383259.383265.

[18] Koc, B.; Lee, Y.S.: Hollowing stl objects with biarcs fitting to improve efficiency and accuracy of rapid prototyping processes. In IIE Annual Conference. Proceedings, 1. Institute of Industrial and Systems Engineers (IISE), 2002.

[19] Lien, J.M.: Covering minkowski sum boundary using points with applications. Computer Aided Geometric Design, 25(8), 652–666, 2008. http://doi.org/10.1016/j.cagd.2008.06.006.

[20] Lim, S.P.; Haron, H.: Surface reconstruction techniques: a review. Artificial Intelligence Review, 42(1), 59–78, 2014. http://doi.org/10.1007/s10462-012-9329-z.

[21] Liu, S.; Wang, C.C.: Fast intersection-free offset surface generation from freeform models with triangular meshes. IEEE Transactions on Automation Science and Engineering, 8(2), 347–360, 2010. http://doi.org/10.1109/TASE.2010.2066563.

[22] Lorensen, W.E.; Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. ACM siggraph computer graphics, 21(4), 163–169, 1987. http://doi.org/10.1145/37402.37422.

[23] Maekawa, T.: An overview of offset curves and surfaces. Computer-Aided Design, 31(3), 165–173, 1999. http://doi.org/10.1016/S0010-4485(99)00013-5.

[24] Pavić, D.; Kobbelt, L.: High-resolution volumetric computation of offset surfaces with feature preservation. In Computer Graphics Forum, vol. 27, 165–174. Wiley Online Library, 2008. http://doi.org/10.1111/j.1467-8659.2008.01113.x.

[25] Pfister, H.; Zwicker, M.; Van Baar, J.; Gross, M.: Surfels: Surface elements as rendering primitives. In Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 335–342, 2000. http://doi.org/10.1145/344779.344936.

[26] Pham, B.: Offset curves and surfaces: a brief survey. Computer-Aided Design, 24(4), 223–229, 1992. http://doi.org/10.1016/0010-4485(92)90059-J.

[27] Piegl, L.A.; Tiller, W.: Computing offsets of nurbs curves and surfaces. Computer-Aided Design, 31(2), 147–156, 1999. http://doi.org/10.1016/S0010-4485(98)00066-9.

[28] Qu, X.; Stucker, B.: A 3d surface offset method for stl-format models. Rapid prototyping journal, 2003. http://doi.org/10.1108/13552540310477436.

[29] Roberts, M.: Evenly distributing points in a triangle. http://extremelearning.com.au/evenly-distributing-points-in-a-triangle/, 2019. [Online; accessed 24-March-2021].

[30] Rossignac, J.R.; Requicha, A.A.: Offsetting operations in solid modelling. Computer Aided Geometric Design, 3(2), 129–148, 1986. http://doi.org/10.1016/0167-8396(86)90017-8.

[31] Varadhan, G.; Manocha, D.: Accurate minkowski sum approximation of polyhedral models. In 12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings., 392–401. IEEE, 2004.

[32] Yi, I.L.; Lee, Y.S.; Shin, H.: Mitered offset of a mesh using qem and vertex split. In Proceedings of the 2008 ACM symposium on Solid and physical modeling, 315–320, 2008. http://doi.org/10.1145/1364901.1364945.