BSP-ASSISTED CONSTRAINED TETRAHEDRALIZATION

Bhautik Joshi [1 2] Sébastien Ourselin [1]

[1] *BioMedIA Lab, CSIRO ICT Centre, Sydney, Australia.*
[2] *University of New South Wales, Sydney, Australia.*
Bhautik.Joshi@csiro.au, Sebastien.Ourselin@csiro.au

ABSTRACT

In this paper we tackle the problem of tetrahedralization by breaking non-convex polyhedra into convex subpolyhedra, tetrahedralizing these convex subpolyhedra and merging them together. We generate a Binary Space Partition (BSP) tree from the triangular faces of a polyhedron and use this to identify the convex subpolyhedra in the polyhedron. Each convex subpolyhedron is tetrahedralized individually. Using an original merging process, the boundaries between these subpolyhedra are joined and tetrahedralized, ensuring that no tetrahedra are created outside of the original polyhedron in this merging process.

Keywords: computational geometry, mesh generation, tetrahedralization, BSP, Delaunay

1. INTRODUCTION

Closed polyhedra in 3D can be described using one or more non-self-intersecting, closed boundaries. The boundaries themselves are often constructed using triangular faces. These polyhedra have an interior and an exterior, and have a finite volume. Tetrahedralization of them involves finding a set of tetrahedra that completely fill the polyhedron and that lie exactly on or inside of its boundaries.

Generally, this tetrahedralization can be performed using a Delaunay tetrahedralization algorithm. Given a set of vertices in a polyhedron, the Delaunay tetrahedralization algorithm creates a convex set of tetrahedra with these vertices. None of the tetrahedra intersect each other, and the minimum angle (between edges or faces) in the mesh is maximised.

However, a Delaunay triangulation or tetrahedralization algorithm generates only convex meshes. This means that it can fail to recover the boundary of the polyhedron due to local non-convex regions, such as dents and holes, which tend to get meshed over and sealed. A generic approach to this problem involves first generating a Delaunay tetrahedralization of the entire domain and removing tetrahedra that lie outside it [1]. However, this does not guarantee that all the boundary faces can be recovered.

To partially solve this, tetrahedralization of a non-convex polyhedron typically involves the placement of extra points on and inside it. These extra points, commonly called Steiner points, enable the algorithm to completely cover the domain at the added cost of extra complexity in the generated mesh. In 2D, several reliable methods for preserving the boundary with minimal point insertion have been proposed [1]. However, in 3D, an algorithm that can create a tetrahedralization preserving boundaries and with an acceptable degree of extra complexity is still a current and challenging problem.

Over the past ten years, two dominant approaches to preserve boundaries in 3D meshing have emerged: *conforming* and *constrained* tetrahedralizations. In the following, we describe two major techniques then briefly review some of the other methods available in the literature [1].

Conforming Delaunay tetrahedralization is a strict Delaunay tetrahedralization of the input polyhedron. If necessary, Steiner points are inserted into the mesh to ensure that all boundaries of the input mesh are preserved, while complying strictly with the Delaunay empty sphere criteria. In 2D, a clear upper bound on the number of added

[1] We recommend to the reader an exhaustive survey of tetrahedralization methods by Bern *et al.* [2].

Steiner points to perform a conforming triangulation has been established as $\mathcal{O}(n^3)$ for n input verticies [3]. However, conforming tetrahedralization can come at a higher cost in terms of number of Steiner points inserted.

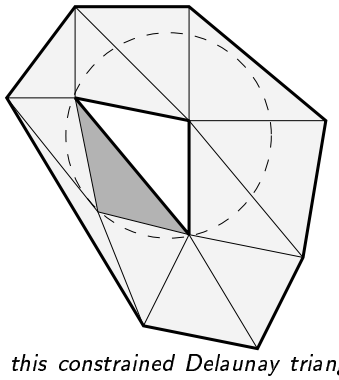


Figure 1: *In this constrained Delaunay triangulation, the boundary edges are indicated by the darkened lines. To preserve boundary edges, the highlighted triangle with its circumcircle (dashed lines) is not Delaunay.*

Constrained Delaunay tetrahedralizations are not strictly Delaunay. To assist boundary recovery, the empty circle theorem is relaxed to allow for elements that are locally Delaunay [4]. Elements that lie on a boundary edge or face are allowed to be created even if they violate the empty circle theorem (see Figure 1). Constrained tetrahedralizations have fewer restrictions on the input meshes. Although they are not a strict Delaunay tetrahedralization, they still help to generate good quality meshes and require insertion of fewer points than conforming algorithms [5, 6, 7].

An extension of constrained Delaunay tetrahedralizations has been proposed by Shewchuk, that is able to tetrahedralize non-convex polyhedra by using a Delaunay refinement approach [8]. However, it is restricted to working with meshes whose faces form angles of greater than 90 degrees with each other.

The first group to create an algorithm that was able to perform a conforming Delaunay tetrahedralization without any restrictions on the input mesh were Murphy *et al.* [9]. The method described extended a 2D conforming triangulation method to 3D [10]. They stated that the number of Steiner points it added to the mesh was too large to be practical. The paper proved that there does exist an upper bound on the number of Steiner points needed to tetrahedralize a polyhedron, but the value of this bound is yet to be determined [9, 11].

By adapting the algorithm to the local features of the geometry being tetrahedralized, Cohen-Steiner *et al.* were able to reduce the number of Steiner points added for a 3D conforming Delaunay triangulation [11]. The algorithm makes use of the Delaunay refinement approach, which adds Steiner points to the mesh until the original boundaries have been recovered. In practice, the ratio of number of added vertices to input vertices varied between 3 to 1 and 10 to 1. However the technique did not state any bounds on this ratio.

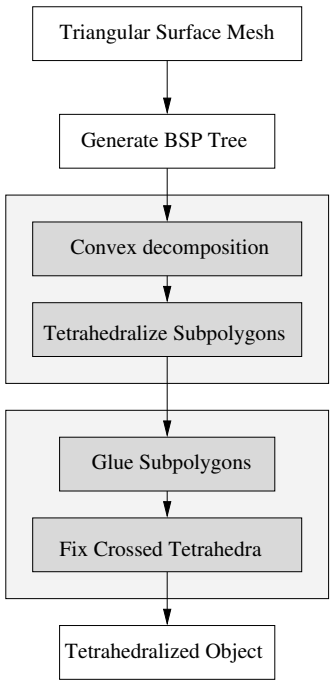


Figure 2: *The overall BSP-assisted constrained tetrahedralization algorithm.*

Another approach performs tetrahedralization by using the existing Delaunay triangles in a boundary [12]. It made use of the triangles to accelerate a Delaunay tetrahedralization algorithm; however, it did not indicate if it was able to cope with non-convex polyhedra.

By partitioning a polyhedron into subpolyhedra, the problem of tetrahedralization is simplified as it allows for individually tetrahedralizing each subpolyhedron and merging the results together. It has been shown that the task of partitioning a polyhedron into the minimum number of convex subpolyhedra without Steiner points is NP-complete. However, good algorithms for polygon paritioning exist [13], with many more that allow for Steiner point insertion [14].

A common approach for polyhedral decomposition is to use an pre-defined grid, often made up of orthogonal planes to divide the polyhedron into finite sized cells. However, these techniques can be limited by the local geometry of the polyhedron [15], especially when complex features are smaller than the size of the cell size.

To tackle this problem and take advantage of the simplicity of grid-based mesh generation, we propose using a Binary Space Partition (BSP) tree to decompose an input mesh into convex regions. Each of these regions are individually tetrahedralized and then merged together with additional tetrahedra to reconstruct the original polyhedron. This technique deals effectively with non-convex polyhedra, and unlike other grid-based techniques, is not

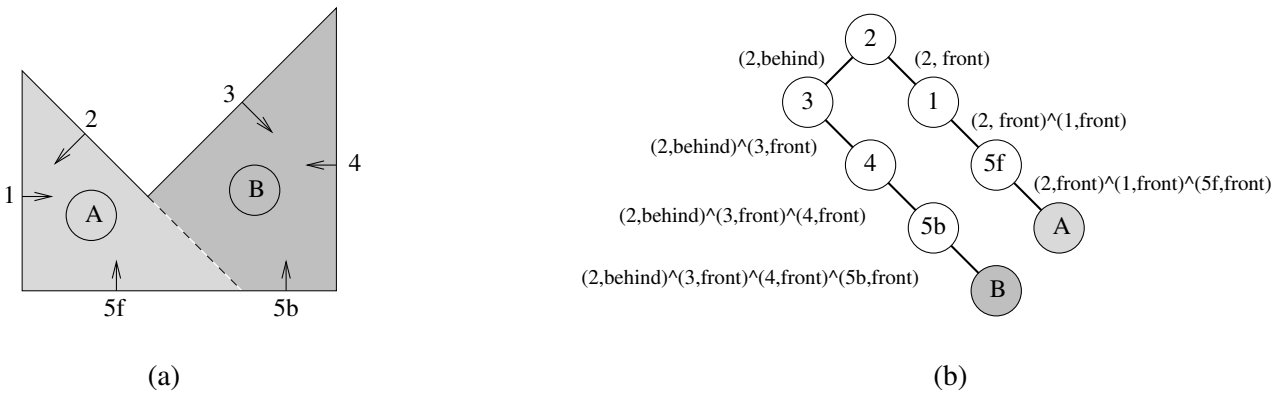


Figure 3: *Creation of a BSP tree and convex subpolyhedron identification for a simple polygon. The polygon is shown in (a) and the corresponding BSP tree is shown in (b).*

limited by grid resolution because the grid conforms to the faces of the original polyhedron itself.

The proposed algorithm serves as an initialisation for quality mesh generation. Once it creates an efficient covering tetrahedralization of the polyhedron, the generated tetrahedra can be easily subdivided without the need for edge flips, providing a guarantee that the input boundary topology is preserved.

The algorithm described can triangulate non-convex polyhedra in 2D as well; however, this paper is focused on its 3D applications. Our proposed algorithm is described in detail in Section 2, with preliminary results using non-convex polygons presented in Section 3. In Section 4, we discuss advantages and limitations of the current technique, and finally propose several paths for future research.

2. METHODOLOGY

Given an input polygon in 2D or 3D, a BSP tree of it can be created. Using this tree, convex subpolyhedra within the polyhedron are established [16]. Sets of points in each of these convex subpolyhedra are identified and tetrahedralized using an incremental Delaunay algorithm. Using the BSP tree, these convex sites are *glued* together using an adaptation of the same incremental Delaunay algorithm. The BSP tree can be traversed recursively to efficiently perform this gluing process with only two subpolyhedra at a time. An additional step to correct tetrahedra that are not coincident with each other is necessary to complete the tetrahedralization.

A flowchart of the BSP assisted tetrahedralization algorithm is shown in Figure 2. For clarity, convex decomposition and tetrahedralization of convex subpolyhedra are shown to be grouped as a single step, and the gluing and fixing of the subpolyhedra as another. In practice, all four operations can be performed as part of a single recursive algorithm.

2.1 Generating BSP Trees

BSP trees can be regarded as the most general spatial subdivision technique, easily adapted for 2D, 3D and higher dimensions. BSP trees came to the fore in computational geometry and computer graphics as a solution for the painters algorithm [17]. We will not go into the details of BSP tree creation here, and recommend a good introduction to BSP trees from Bruce Naylor [18].

Using BSP trees to find convex subpolyhedra of polyhedra is a well known property of BSP trees [18, 16]. However, because BSP trees split faces and insert points, it is not a strict convex decomposition of the polygon. Ruppert and Sidel have shown that the problem of determining if a given polygon can be tetrahedralized without Steiner points is NP-complete [19]. Other spatial subdivision techniques such as kD-trees or octrees are not used because in polyhedra with boundaries that have high curvature, a high degree of subdivision of the initial grid may be necessary, producing a large number of added points [20]. In addition, as mentioned in the introduction, unlike other spatial subdivision techniques, this decomposition is not limited by grid resolution because the grid is conforming to the faces of the original polyhedron itself.

2.2 BSP Trees for Convex Decomposition

The polygonal face at the root of a BSP tree divides space into two subpolyhedra. As one traverses the BSP tree in the front or behind subpolyhedra, these subpolyhedra are divided again by the faces of the leaf nodes. These leaf nodes may serve as roots of sub-trees which further divide the subpolyhedra, which may go on to have leaves that split these subpolyhedra and so on. In fact, convex subpolyhedra in the polyhedron are the intersection of groups of these subpolyhedra, and a structured traversal of the BSP tree can be used to identify them.

To clarify the process of convex decomposition using BSP trees, we present in Figure 3 a 2D example. The polygon in Figure 3a has five faces. Each face has an orientation;

the side with the arrow represents $front$ and without represents *behind*. These faces are added one at a time into a BSP tree, shown in Figure 3b. In the tree, relative to a face, the right leaf is the front halfspace and the left leaf is the behind halfspace. If the plane that a face lies in intersects a leaf, then the leaf face is split. Hence face 2 splits face 5 into $5f$ and $5b$. As the tree is traversed, a list of tuples representing the face traversed and the direction taken relative to the root face is kept. If a front face is encountered that is empty, then the list is stored as a convex subpolyhedron. The convex subpolyhedron A is represented by the intersection of the halfspaces in front of 2, 1 and $5f$. B is represented by the intersection of the halfspaces in front of 3, 4, $5b$ and behind 2.

2.2.1 Creation of BSP points

Not all of the implicit vertices of the subpolyhedra identified by the BSP tree convex decomposition algorithm are guaranteed to have a corresponding vertex in the original polyhedron. However, during the BSP tree creation process, faces that lie across the plane of a parent node are *split*. This splitting process introduces new points on to the boundary of the polyhedron being tetrahedralized. These *BSP points* help ensure that a complete tetrahedralization is performed. However, it is still unclear if there is a relationship between BSP points and Steiner points.

2.3 Tetrahedralization of subpolyhedra

Once a convex subpolyhedron has been identified, a list of points inside[2] it can be determined. These points can be tetrahedralized, forming a mesh of the convex subpolyhedron. A randomized point insertion Delaunay tetrahedralization algorithm is used. However, it is not guaranteed that this set of points completely describe the boundary of the convex subpolyhedron. This commonly occurs when a node of a BSP tree intersects the plane of a child node but does not intersect the child node itself.

A 2D example of subpolyhedra that are not completely described is shown in Figure 4. The polygon in Figure 4a and one possible corresponding BSP tree, Figure 4b, can be split into two convex subpolyhedra, A and B. Although the plane that the child node 3 lies in intersects the parent node, 1, the face itself does not intersect the parent node. This means that for the convex subpolyhedra, there is no vertex at the intersection of bounding planes 1 and 3, and hence the subpolyhedra do not have a vertex at every corner of the subpolyhedra. Figure 4c illustrates how a triangulation of the subpolyhedra will not recover the whole polygon.

Uncorrected, this problem will lead to missing tetrahedra in the tetrahedralization. To recover these missing

[2]In this paper, a point lying exactly on the boundary of a subpolyhedron is classified as inside the subpolyhedron

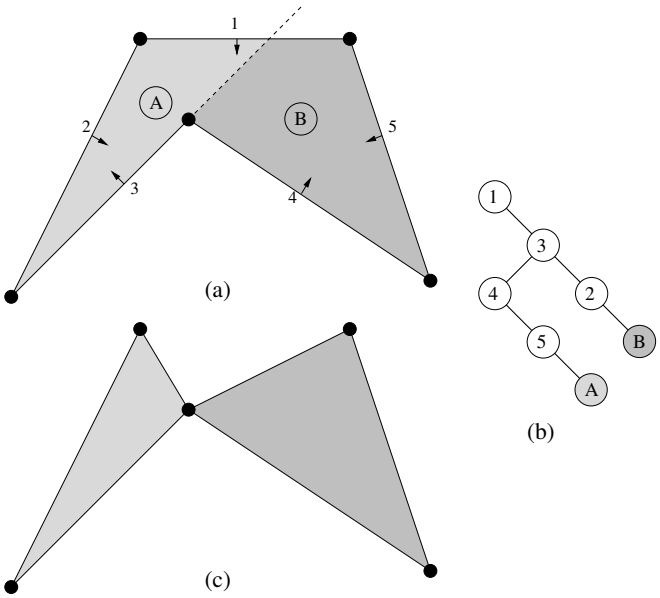


Figure 4: *(a) shows a problem polygon where the convex subpolyhedra are not completely described by points. The corresponding BSP tree is shown in (b). The polygon with its subpolyhedra triangulated is shown in (c).*

tetrahedra, a *glue* algorithm has been devised to merge subpolyhedra together.

2.3.1 Pseudocode Implementation

The tetrahedralization algorithm operates by traversing the BSP tree. A list of tuples representing the node traversed and the direction taken - for example, ($node_n$,[front,behind]) - is necessary to identify the convex subpolyhedra. Once a node has been visited, it is removed from the list. This psuedocode implementation is described in Algorithm 1.

2.4 Gluing subpolyhedra

When two subpolyhedra are merged using the glue algorithm, they are separated exactly by a single splitting plane, which is the root node of the current position in the BSP tree. As such, the algorithm recursively merges two growing subpolyhedra at a time until all subpolyhedra in the tree have been merged. Tetrahedra need to be generated that fill the space in-between the merging subpolyhedra. This is not a new problem; Bern and Marshall have demonstrated that it is possible to tetrahedralize the region between two convex polyhedra without the addition of Steiner points [21].

One way to generate the tetrahedra in-between the subpolyhedra is to merge the points in both the front and behind subsets and perform a Delaunay tetrahedralization on this merged set of points. A simple cross test then can be used to reject tetrahedra that do not merge the tetrahedralization. This merging set of tetrahedra is

Algorithm 1 Traverse and tetrahedralize convex subpolyhedra within a BSP tree

```
1: bsp_tetrahedralize(bsp_tree, traversal_list)
2: frontNode = bsp_tree_front
3: behindNode = bsp_tree_behind
4: if frontNode = empty then
5:    points = getConvexPoints(bsp_tree, traversal_list)
6:    return Delaunay(points)
7: end if
8: push (frontNode, front) on traversal_list
9: front_tetrahedralization=bsp_tetrahedralize(frontNode, traversal_list)
10: pop traversal_list
11: push (behindNode, front) on traversal_list
12: behind_tetrahedralization=bsp_tetrahedralize(behindNode, traversal_list)
13: pop traversal_list
14: return glue(front_tetrahedralization, behind_tetrahedralization)
```

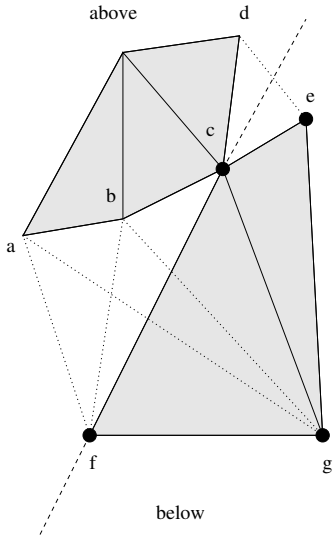


Figure 5: *Merging two convex subpolyhedra.*

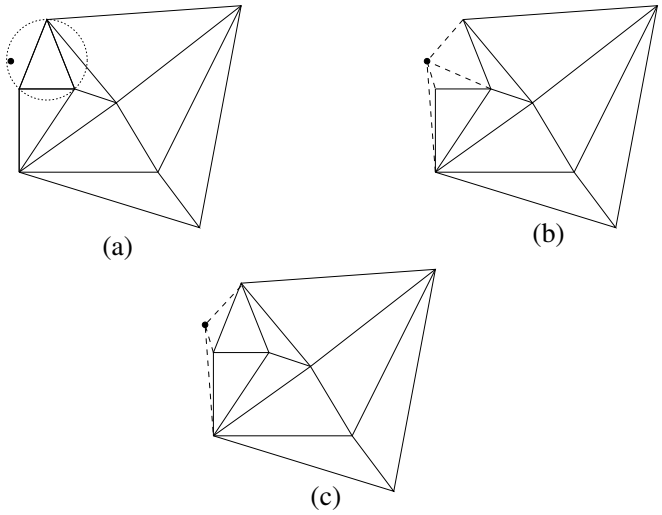


Figure 6: *Randomized Point Insertion: adding a point outside the triangulation when the point is inside the circumcircles of one or more existing triangles is shown in (a). (b) and (c) demonstrate two alternative solutions for triangulation.*

then added to the set of already generated tetrahedra for the front and behind subpolyhedra.

For a tetrahedra to pass the cross test, it must satisfy three criteria. Firstly, all its vertices must not lie completely on and above or on and below of the joining plane. Secondly, at least one edge of the tetrahedra must cross the joining plane. Lastly, none of the edges of the tetrahedra may intersect an existing face on the joining plane.

A 2D example is described in Figure 5. In this merge, the joining plane is represented by the dashed line and is generated from the edge cf. Using the cross test, triangle $\triangle cde$ is legal, as its vertices lie both above and below the joining plane, an edge crosses the joining plane and none of the edges intersect cf. $\triangle afb$ is illegal because all its points above the joining plane, and $\triangle abg$ is illegal because it has edges that cross edges on the joining plane cf.

Unfortunately, the above algorithm does not work for all cases, sometimes failing to completely mesh the region in-between the subpolyhedra. To solve this problem we propose a glue algorithm inspired by the Bowyer-Watson randomised point insertion (RPI) algorithm [22, 23]. It works by joining triangles from the boundary (or *hull*) of the front subset, *front hull*, to the points of the hull below that face those triangles, *below hull points*.

Tetrahedralization of the front and behind subpolyhedra are performed independently of each other. Because of this, there is no guarantee that a point in one set would not violate the Delaunay empty sphere criteria of a tetrahedra in the other set. Hence the region in-between the two sets would not be guaranteed to be tetrahedralized with Delaunay tetrahedra. Therefore a non-Delaunay tetrahedralization algorithm is necessary. A version of the traditional RPI algorithm has been modified so that it performs the gluing tetrahedralization without check-

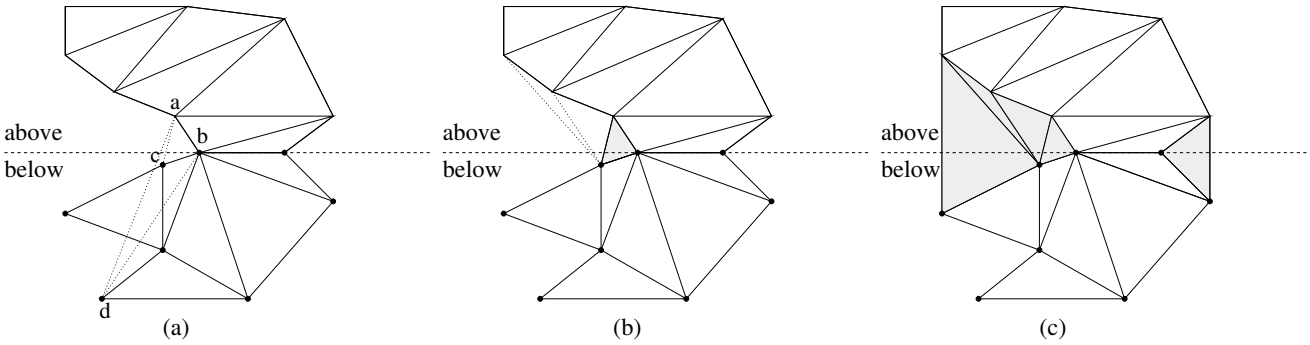


Figure 7: *Here two triangulated subpolyhedra are to be merged along a merging plane (the dashed horizontal line).*

ing the Delaunay in-sphere criteria.

In RPI, a point may be added to a tetrahedralization that lies outside the tetrahedra but infringes upon the circumcircles of some tetrahedra. Conventionally, the tetrahedra that are infringed are deleted and re-meshed and the remaining edges for which the point is behind make new tetrahedra. With the new point insertion algorithm, infringed tetrahedra are not deleted.

A 2D example of this type of point insertion is shown in Figure 6. Triangles whose circumcircles contain the point are deleted and re-meshed with the new point (Figure 6a). Once this is completed, the edges of the hull are used to create new triangles (same as if the point was lying completely outside the triangulation and circumcircles)(Figure 6b). With the glue algorithm, as shown in Figure 6c, no tetrahedra are deleted and triangles are created from edges on the hull and the exterior point. The glue algorithm operates in a similar fashion, but the tetrahedra that are infringed are not deleted and tetrahedra are created from any faces for which the point lies behind.

The *below hull points* are added individually to the *front hull* using this technique. An example is shown in Figure 7. The halfspace above the line is indicated simply as *above*, below as *below*. The boundary edges in *above* are identified and the points on the boundary of *below* are also identified, as described in Figure 7a. For each of the boundary edges in *above*, a test triangle for each boundary point in *below*.

For the triangle to be allowed, it must not intersect any edges on the joining plane or intersect any of the boundary edges in *below*. So, $\triangle abc$ is allowed, but $\triangle abd$ is not because one or more of its edges intersect the below boundary. Once a legal triangle is created, it is added to the *above* set and further edges may attempt to create triangles with the added point as in Figure 7b. When all the points are added to the *above* subpolyhedron, the region between *above* and *below* is triangulated as delineated as the darkened area in Figure 7c.

2.5 Fixing Crossed Tetrahedra Edges

While the *glue* algorithm completely fills the space between the *above* and *below* subpolyhedra, it is not guaranteed that the gluing process creates tetrahedra that have edges that are all coincident to each other, leading to edges that may intersect. These intersections can possibly occur on any of the planes defined by the faces of the *below* hull.

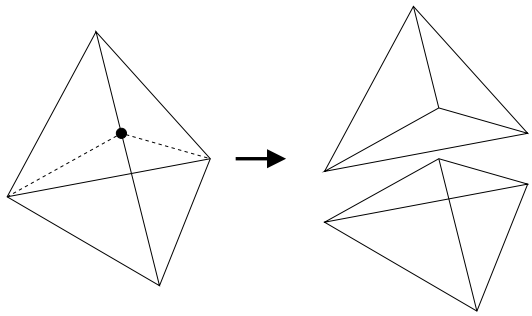
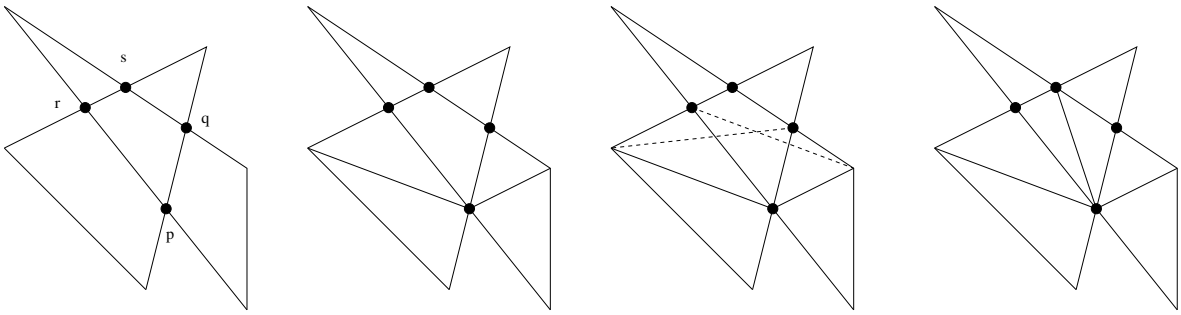


Figure 8: *The addition of a point to the edge of a tetrahedron on the left leads to the creation of two new tetrahedra, shown separated, on the right.*

This can usually be corrected by inserting a point at the site of an intersection and splitting the tetrahedra for which this added point lies on. Each point added to the edge of a tetrahedron leads to the creation of two new tetrahedra, as described in Figure 8.

A whole face of a tetrahedron may lie on a plane for which there are edges that are crossed. A case can occur where adding the intersected point generates an edge which in turn generates a new intersection, leading to an infinite loop of new intersection points being generated. This can be avoided by checking to see if the intersected point generates an edge which generates more invalid crossed edges. If this is the case, the intersected point is pushed to the end of the list of intersected points to be added. In this fashion, all the intersected points can be added.

Figure 9 describes an example of this process. Two tetrahedra have triangular faces that lie on a common plane. Their vertices and edges are not coincident and they lie across each other as shown in Figure 9a, creating four in-



(a) (b) (c) (d)

Figure 9: *Fixing crossed edges.*

tersection points - p,q,r,s. These intersection points are pushed onto a list. p is the first point to be added to the tetrahedra, and does not create any new intersections, as shown in Figure 9b. However, when q is popped off the top of the list it creates edges that create new intersection points, as shown in Figure 9c. q is placed at the bottom of the list to be handled later. The same applies to r. s is now popped off the top list, and its addition does not create any new intersecting edges, as shown in Figure 9d. q and r are (respectively) popped of the list now, and can safely split the tetrahedra without creating any new intersected edges.

3. PROOF OF CONCEPT

To test the BSP assisted tetrahedralization algorithm, four non-convex 3D polyhedra were constructed. The first is a cube which had one corner indented, the second is the well known Schönhardts polyhedron, the third is a cube with a hole through it on one axis and lastly a rectangular prism with opposing angular cuts on two opposite faces, making a dented polygon.

These polyhedra are closed surface meshes constructed from triangular faces. Our BSP assisted tetrahedralization algorithm has been applied to this set. During this process, we recorded several pertinent variables, such as the initial number of points in the polyhedra, the number of points added, and the final number of tetrahedra. We summarise these results in Table 1. In Table 2, we present the different polyhedra at each step of the tetrahedralization process.

4. DISCUSSION

4.1 Theoretical Bounds

Current algorithms for tetrahedralizations produce a large number of Steiner points that need to be added. In the two dimensional case, a lower bound of $\mathcal{O}(n^3)$ has been found [3]. However, a lower bound for the number of inserted Steiner points in 3D has yet to be firmly established [11]. One could reasonably expect that 3D techniques that extend 2D would only increase the lower bound (*i.e.* greater than $\mathcal{O}(n^3)$).

Given a 3D polyhedra made of f triangular faces and n vertices, in the worst case, a naive BSP tree generation algorithm will produce a tree made from $\mathcal{O}(f^3)$ faces. In practice, however, empirical results have shown that BSP trees of 3D polyhedra tend to produce trees with close to $\mathcal{O}(flog(f))$ faces [18]. Paterson and Yao demonstrated that it is possible to construct 3D BSP trees with a provable upper bound of $\mathcal{O}(f^2)$ and lower bound of $\mathcal{O}(f^{3/2})$ [24]. It is expected that a BSP would produce BSP polygons with a number of vertices in the order of $\mathcal{O}(nlog(n))$.

4.2 BSP and Steiner Points

During the process of creating a BSP tree from the faces of an input mesh, many of the faces are split. Each time a split occurs, two new points are introduced into the mesh. Rather than discard these *BSP points*, they are retained as part of a new mesh with no split faces as defined by the BSP tree. In practice, these points appear to fulfil the role of Steiner points, helping to ensure that convex regions can be tetrahedralized. However, a proof that links BSP points and Steiner points is yet to be established.

4.3 Practical Considerations

It is possible to construct many different BSP trees from the same input mesh. To help make the BSP trees generated more consistent, it was decided that a selection criteria for choosing root nodes in BSP tree construction was necessary. The metric used was the number of leaf nodes split if a particular root node was chosen. In practice, meshes tended to have fewer added points and tetrahedra when the choice of BSP root node was one that minimised the number of split leaf nodes.

The mesh generator does not guarantee the quality of any of the tetrahedra generated. However, the mesh generation process is not limited by the size of local geometrical features and as such is scale independent.

Numerical Results				
Variable	Dented box	Schönhardts polygon	Box with hole	Dented polygon
Initial Number of Points in Polygon	8	6	16	26
Total Number of Points Added	0	6	1	7
Number of Tetrahedra	9	18	29	70

Table 1: *Numerical results on non-convex polyhedra.*

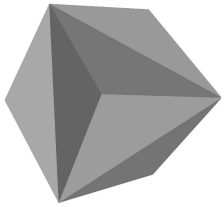
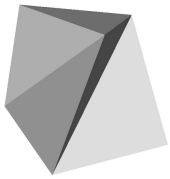
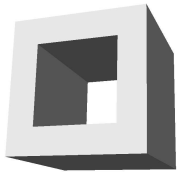
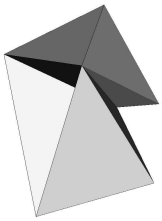
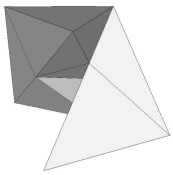
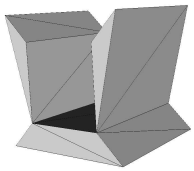
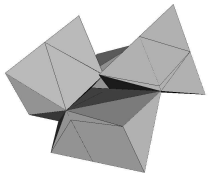
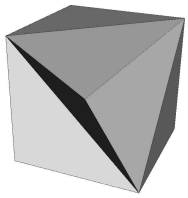
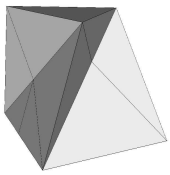
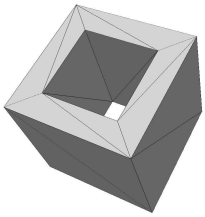
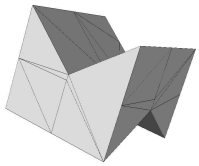
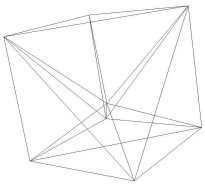
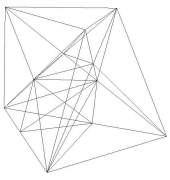
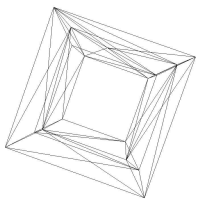
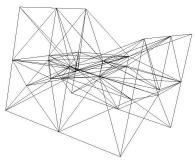
Graphical Results Table				
Type	Dented box	Schönhardts polygon	Box with hole	Dented polygon
Original				
Before gluing				
After gluing				
Wireframe				

Table 2: *Graphical results table for non-convex polyhedra.*

Because convex subpoloygons are tetrahedralized individually, and only the hulls of these subpolyhedra are necessary for the bulk of the computation, completed subpolyhedra are swapped out-of-core into a cache on disk. In this way, it may be possible to tetrahedralize very large meshes by swapping out completed portions of the mesh.

As with many tetrahedralization algorithms, practical implementation is plagued by problems with computational precision.

4.4 Limitations

The weakest part of the algorithm is in the gluing process, and the author believes that it will require more development. The main problem lies in the fact that the set of generated tetrahedra between the subpolyhedra is non-Delaunay, and reliable properties and attributes of these tetrahedra have yet to be determined.

Related to this is the problem of finding the set of faces in the above subpolyhedron that face the below subpolyhedron in the gluing process. At the moment, there is no sure way to prevent faces from the other side of the subpolyhedron that are simply just facing in the right direction to be treated as candidates for gluing to the bottom subpolyhedron.

5. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we have presented a covering tetrahedralization algorithm providing an efficient initialisation for a quality mesh generator. As such, these tetrahedra can be subdivided and refined without any edge flips which guarantees to preserve the topology of the polyhedron input boundary.

The main innovation of this is bringing together BSP trees and Delaunay Tetrahedralization, along with a tetrahedral mesh generator that is not limited by the local complexity of the geometry.

Based on our experiments, the number of added points has been quite low. Future testing on more complex meshes will yield more conclusive information about the properties of this algorithm and how it may perform in practice.

The number of points added by a BSP decomposition of a polygon has already been quite well established. If a link between BSP points and Steiner points can be found, is may be possible that bounds on the number of extra points necessary to tetrahedralize any polyhedron can be found.

So far, the meshes generated are Constrained Delaunay Tetrahedralizations, and no consideration is given to the quality of the meshes generated. Preliminary work has shown that it is, in theory, possible to convert the constrained tetrahedralizations to conforming ones via point insertion and Delaunay refinement. This is work currently in progress.

The algorithms in this paper were implemented in C++ and made use VTK for rendering output [25]. The computational geometry functions implemented and the BSP assisted tetrahedralization algorithm will be made available online.

6. ACKNOWLEDGEMENTS

The authors would like to thank R. Li and N. Killeen for proofreading of this article, and J. Ables and B. Doyle for their support.

References

[1] Frey P.J., George P.L. *Mesh Generation Application to Finite Elements*. Hermes Science Publishing, 2000

[2] Bern M., Eppstein D. "Mesh generation and optimal triangulation." D.Z. Du, F.K.M. Hwang, editors, *Computing in Euclidean Geometry*, no. 4 in Lecture Notes Series on Computing, pp. 47–123. World Scientific, second edn., 1995

[3] Edelsbrunner H., Tan T.S. "An Upper Bound for Conforming Delaunay Triangulations." *Proc. 8th Symposium on computational geometry, Universitat Politècnica de Catalunya, Barcelona, Spain*, pp. 53–62. ACM Press, Jun. 1992

[4] Chew L.P. "Constrained Delaunay Triangulations." *Algorithmica*, vol. 4, 97–108, 1989

[5] Shewchuk J.R. "Mesh generation for domains with small angles." *Proceedings of the sixteenth annual symposium on computational geometry, Hong Kong, China*, pp. 1–10. ACM Press, 2000

[6] Shewchuk J.R. "Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery." *Proceedings of the 11th International Meshing Roundtable, Ithaca, New York, USA*, pp. 193–204. ACM Press, 2002

[7] Shewchuk J.R. "What is a Good Linear Element? Interpolation, Conditioning, and Quality Measures." *Proceedings of the 11th International Meshing Roundtable, Ithaca, New York, USA*, pp. 115–126. ACM Press, 2002

[8] Shewchuk J.R. "Tetrahedral Mesh Generation by Delaunay Refinement." *Fourteenth Annual Symposium on computational geometry, Minneapolis, MN, USA*, pp. 86–95. ACM Press, 1998

[9] Murphy M., Mount D.M., Gable C.W. "A Point-Placement Strategy for Conforming Delaunay Tetrahedralization." *International Journal of Computational Geometry and Applications*, vol. 11, no. 6, 669–682, 2001

[10] Chew L.P. "Guaranteed-Quality Triangular Meshes." Tech. Rep. 89-983, Department of Computer Science, Cornell University, 1989

[11] Cohen-Steiner D., de Verdière E.C., Yvinec M. "Conforming Delaunay Triangulations in 3D." *Proc. of the Eighteenth Annual Symposium on Computational Geometry, Barcelona, Spain*, pp. 199–208. ACM Press, 2002

[12] Choi S. "The Delaunay tetrahedralization from Delaunay triangulated surfaces." *Proc. of the Eighteenth Annual Symposium on Computational Geometry, Barcelona, Spain*, pp. 145–150. ACM Press, 2002

[13] Chazelle B., Palios L. "Triangulating a non-convex polytype." *Proceedings of the fifth annual symposium on Computational geometry*, pp. 393–400. ACM Press, 1989

[14] Keil J.M. *Handbook of Computational Geometry*, chap. 11, pp. 491–518. Elsevier, 2000

[15] Owen S.J. "A Survey of Unstructured Mesh Generation Technology." *Proceedings, 7th International Meshing Roundtable, Sandia National Lab., USA*, pp. 239–267. Oct. 1998

[16] Baldazzi C., Paoluzzi A. "From Polyline to Polygon via XOR tree." Tech. Rep. INF-04-96, Dip. Disc. Scient., Universita Roma Tre, Rome, Italy, 1996

[17] Fuchs H., Kedem Z.M., Naylor B.F. "On visible surface generation by a priori tree structures." *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques, Seattle, Washington, USA*, pp. 124–133. ACM Press, 1980

[18] Naylor B. "Constructing Good Partitioning Trees." *Graphics Interface '93, Toronto Canada*, pp. 181–191. May 1993

[19] Ruppert J., Seidel R. "On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra." *Proceedings of the fifth annual symposium on Computational geometry*, pp. 380–392. ACM Press, 1989

[20] Mitchell S.A., Vavasis S.A. "Quality Mesh Generation in Three Dimensions." *Proc. 8th Symposium on computational geometry, Universitat Politècnica de Catalunya, Barcelona, Spain*, pp. 212–221. ACM Press, 1992

[21] Bern M. "Compatible tetrahedralizations." *Proc. 9th Annual Symp. Computational Geometry, San Diego, USA*, pp. 281–288. ACM Press, 1993

[22] Bowyer A. "Computing Dirichlet tessellations." *The Computer Journal*, vol. 24, no. 2, 162–166, 1981

[23] Watson D. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes." *The Computer Journal*, vol. 24, no. 2, 167–172, 1981

[24] Paterson M.S., Yao F.F. "Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling." *Discrete and Computational Geometry*, vol. 5, 485–503, 1990

[25] Schroeder W., Martin K., Lorenson B. *The Visualisation Toolkit - An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, 2nd edn., 1998