Automatic Convexification of Space using BSP-trees

João L. D. Comba* UFRGS

Cláudio T. Silva† OHSU

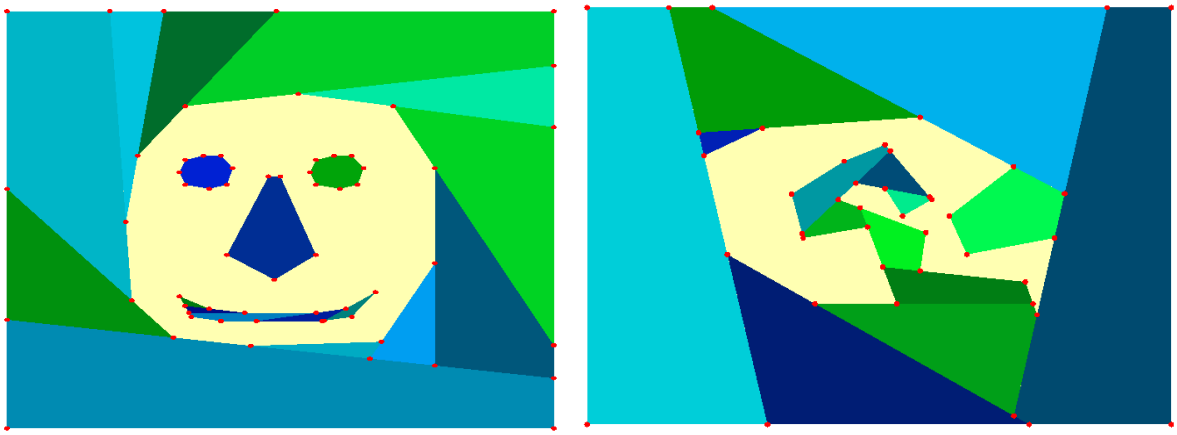

Figure 1: Convexification of space using a BSP-based filling algorithm. The boundary faces of a non-convex region of space are used as cuts in a BSP-tree. Enumeration of regions associated with outside leaves of the BSP-tree fills the space in a simple and numerically stable way.

Abstract

Convex representations of shapes have several nice properties that can be exploited to generate efficient geometric algorithms. At the same time, extending algorithms from convex to non-convex shapes is non-trivial and often leads to more expensive solutions. An alternative and sometimes more efficient solution is to transform the non-convex problem into a collection of convex problems using a convexification approach. In this paper, we address the issue of building convexification of 3D spatial domains. This process is non-trivial, since it might lead to many convex cells and the computations are subject to numerical errors. In particular, we give the first fully automatic convexification technique of non-convex polyhedral meshes that leads to most of the time to a small increase in the number of convex cells. The basic idea of our technique is to use the leaves of a binary space partition tree (BSP-tree) to create the cells that we use for filling up the space between the non-convex polyhedral mesh and its convex hull. We show an application of our ideas to volume rendering of unstructured grids.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms

Keywords: BSP-trees, Space Filling

1 Introduction

In this paper, we address the issue of building convexification of 3D spatial domains. Given a set of polyhedral boundaries, our algorithm creates a convex envelope that completely fill the space outside of such boundaries. The basic idea of our technique is to use the leaves of the binary space partition tree used in BSP-XMPVO [1] to create the cells that fill up the space between the boundary of the grid and its convex hull. Unfortunately, such a naive solution does not work: geometric degeneracies cause such a computation to be all but impossible. Instead, we make novel use of properties of the BSP-tree to robustly compute all the necessary information.

This kind of research has direct applications in visualization, in the form of optimizing the volume rendering of unstructured grids. In particular, using our technique, it is possible to perform fully automatic convexification technique for unstructured volumetric grids. Previous work in this particular topic by Williams [11] and Kraus and Ertl [4] lead to partial solutions.

This paper is organized as follows. We first describe some related work in Section 2. In Section 3, we describe our new algorithm. In Section 4, we discuss the issues in using our convexification algorithm for volume rendering of unstructured grids. In Section 5, we report some experimental results. We finish the paper in Section 6 with final remarks.

2 Related Work

The computational geometry and mesh generation community have done most of the work on topics related to the space triangulation of the outside of the boundary of polyhedral domains [3]. Particularly relevant are the works in the computation of constrained Delaunay triangulations [8, 7], which has in fact been the class of solutions advocated by Williams [11], who was possibly the first to encounter this problem in the context of visualization. Computing such geometrical structures is no easy task, and is often limited to datasets of reasonably small sizes because of both memory and computational constraints. Furthermore, the implementation requires carefully designed exact-arithmetic primitives [6]. The algorithms proposed in this paper are considerably simpler than this more general solution.

*comba@inf.ufrgs.br
†csilva@cse.ogi.edu

Another solution, proposed by Kraus and Ertl [4], is to use semi-automatic techniques. That is, often, the non-convexity can be solved by the careful addition of a few new vertices. This process can be achieved interactively by a semi-automatic tetrahedrilization scheme, where a user would place points, and specify the area to be triangulated.

3 BSP FILLER ALGORITHM

3.1 Using BSP-trees to fill space

We claim that the BSP-tree is a good supporting structure to capture the geometry and topology of empty space. As the geometry of empty space adjacent to a mesh is given by the boundary faces of the model, it suffices to build a BSP-tree with only the boundary faces of the model as cuts in space. One reason for choosing BSP-trees is that cuts in space only use normals that come from boundary faces, which greatly reduces numerical problems that usually arise if new normals are created. In addition, known normals makes it more stable to recover the topological relations among cells. Finally, computation is extremely fast by using the search structure properties of the BSP-tree.

In Figure 2 we illustrate the basics of the algorithm. Given an input mesh, shown in Figure 2a, the boundary faces are extracted and used as cuts in the BSP-tree. The resulting BSP-tree is represented by its decomposition in Figure 2b (for convenience enclosed in a bounding box), and its tree structure, shown in Figure 2c. An important property of BSP-trees is that each node corresponds to a convex region of space, defined by the intersection of the halfspaces in a path from the node of interest until the root of the tree. A related property is that the union of all convex regions associated with all leaves in the tree fills the space. This can be observed in the example, where the leaves of the BSP-tree are colored accordingly to the convex region they represent in the decomposition. Enumerating only the convex regions outside the input model requires separating between inside and outside cells. We assume that the face normals in the input model always point outside the model, and we encode this information implicitly in the tree by having the right subtree always correspond to the region facing the normal. As a result of this convention, leaf nodes that descend in left (right) subtrees correspond to internal (external) regions of the model. Therefore, the filler set correspond to the convex regions associated with all right descendants leaf nodes in the BSP-tree. The resulting input mesh augmented with the filler cells is shown in Figure 2d.

3.2 Generating filler cells

The hierarchical representation of convex regions encoded in a BSP-tree suggests a traversal-type of algorithm to enumerate the filler regions of the outside cells. Our algorithm performs a depth-first traversal of the BSP-tree, keeping at each visited node the convex region associated with the node. In Figure 3 we illustrate this traversal and the regions obtained in all nodes in the BSP-tree of Figure 2c. The convex region associated with the root of the tree is unbounded, and for convenience, we replace it by a bounded region that is guaranteed to enclose the input mesh. The region is represented by a list of faces (or list of edges in the case of the 2D example of the figure). The algorithm proceeds recursively as follows. For each node, the cell associated with the node is partitioned by the hyperplane used to cut the space. This operation is simply a partition of the cell (a collection of faces) by a plane, generating two set of faces, one for each halfspace. Note that an important face is missing, the one defined by the node itself, which comes from the intersection of its convex cell with the hyperplane of the node. Each new cell formed is passed into the subtree that

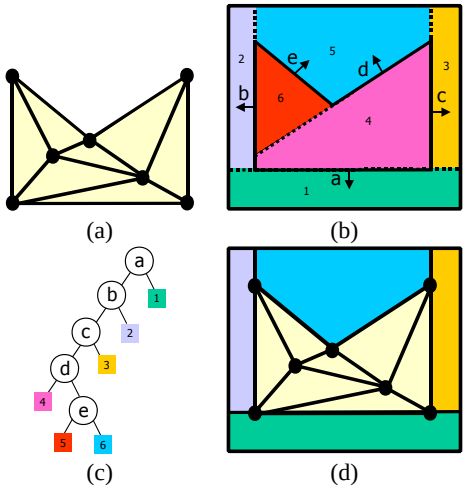

Figure 2: **BSP-Filler Algorithm**. (a) Input non-convex mesh. (b) BSP decomposition using the boundary faces of input mesh. (c) Corresponding BSP tree (d) Input mesh augmented with cells generated by the filler algorithm.

corresponds to the halfspace associated with the cell, and computation proceeds as described. Once an outside leaf node is reached, a new fille cell is added to the filler set and we proceed to compute adjacency relations.

3.3 Finding adjacency relations in the BSP-tree

In order to connect the input polyhedral mesh with the cells generated during convexification, it is necessary to build an adjacency graph between the two types of cells. A common procedure to find adjacency information in the BSP-tree is to query which leaves contain a given geometric entity. In the case of points, for instance, it corresponds to a simple point location algorithm, that starts at the root of the tree, and follows the point down the tree, choosing the subtree to continue the search depending on which side with of a node's hyperplane the point lies.

If the queried entity is a face (represented as a collection of points), a similar procedure can be applied, comparing each point of a face against the hyperplane of a given node in the tree. If some points lie in opposite sides, the face is split in two faces, generating two sub-faces where computation proceeds recursively. Unlike points, it is likely that more than one leaf node may be reached by sub-faces, which means that more than one cell of the BSP-tree contains the queried face. We simply call this a *face-location* algorithm, receiving as input a query face and a node indicating where the search starts, and returns a list of the cells reached by the face.

There are three types of adjacencies relating filler and mesh cells that need to be computed by the filler algorithm: filler face to mesh cell (fM), filler face to filler cell (fF), and mesh face to filler cell (mF).

Filler to Filler Adjacencies

We start describing how to obtain fF adjacencies. In the moment that the traversal in the filler algorithm reaches an outside leaf node, all faces of the cell corresponding to this node are available. For each filler face, we look for adjacent cells in the BSP-tree using the face-location algorithm described above. Instead of starting the

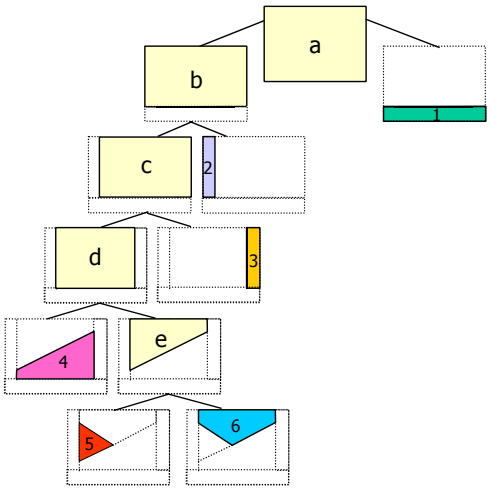

Figure 3: **Extracting convex regions from BSP-trees.**

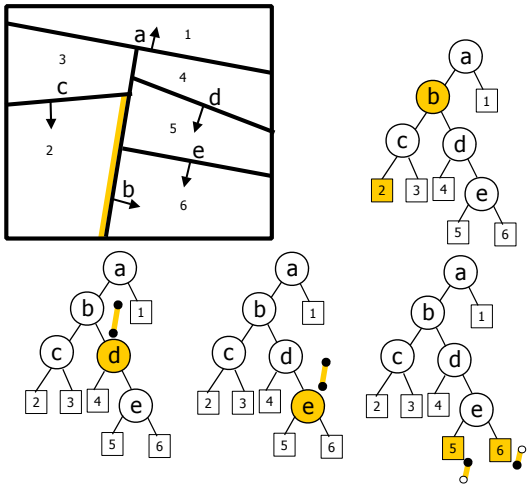

Figure 4: **Face Location algorithm example**. Find the cells adjacent to the marked face in cell 2, defined on cut b, facing the left subtree of b. Insert face in the right subtree of b. Each node tested against the face is highlighted. The test against node d passes the face to its right subtree, while against the node e splits the face in two. Leaf nodes (5, 6) reached by sub-faces are the adjacent cells.

face-location search at the root of the tree, we start at the deepest node in the tree that is guaranteed to contain all adjacent cells. In this case, if the face was defined in the left halfspace of a node n in the tree, then this node corresponds to the right child of n (and vice-versa). The starting node for face-location is found quickly because we keep, for each face, a pointer to this node whenever a new face is generated in the filler algorithm. Finaly, we change the result of the face-location algorithm to discard inside cells, because filler to cell adjacencies will be treated elsewhere. In Figure 4 we describe an example of the face-location algorithm in action.

Mesh to Filler Adjacencies

The computation of mF relations preceeds the filler algorithm, and saves information at the leaves of the tree that are later used to recover fM adjacency relations. For each boundary face of the mesh, we use a face-location algorithm starting at the root of the tree, looking for all outside leaf nodes that are reached by the face. For each node obtained, we establish the mF adjacency relation. Also, we keep at each node a list of boundary faces that reaches the node, call this $MFLIST(node)$.

Filler to Mesh adjacencies

Now consider fM adjacencies. Because of coplanarity among boundary faces of the mesh, it is possible that a face of a filler cell maybe adjacent to more than one cell of the input mesh. When the filler algorithm reaches an outside cell, we process the $MFLIST$ associated with the node. For each boundary face m in this list, we compare m against the list of faces f that define the convex region of the node. For each f that lies on the same supporting hyperplane of m, we create a fM adjacency relation between f and the mesh cell associated with the boundary face m. Note that this *lies-on-hyperplane* operation is numerically stable because we compare ids of hyperplanes.

4 Application: Volume Rendering

An effective technique for exploring graphics hardware for volume rendering is the Projected Tetrahedra (PT) algorithm proposed in [9]. The main idea is to break a volumetric grid into a collection of tetrahedra, which are then rendered by *splatting* its faces on the screen. In order to apply PT, one needs to compute a visibility-ordering of the cells. One way to compute such an ordering for convex meshes is to partially order the cells based on adjacency information and the orientation of faces and do a search through the resulting graph to determine a correct depth ordering, as for example in the Meshed Polyhedra Visibility Ordering (MPVO) algorithm described by Williams [11, 5] (see Figure 5).

The MPVO sorting algorithm is both fast and accurate: it runs in linear time with low computational overhead and uses linear space for its data structures. Unfortunately, many data sets violate the convex mesh constraints of MPVO. For example, cells may be in a nonconvex mesh, or there may be multiple disconnected components to the mesh (see Figure 6). In the case of disconnected components, there are cells which cannot be related by any transitive chain of in-front relationships across shared faces and yet which may occlude each other, so an ordering based purely on such relationships may incorrectly order the cells.

Recently, several extensions of MPVO for general meshes have been proposed [10, 1, 2]. These techniques are based on augmenting the visibility graph used by MPVO with extra *relations*, while keeping the same nodes (i.e., cells). Computing extra relations can be costly, and modify the underlying MPVO algorithm. An alternative approach (originally proposed in [11]) is to use a convexification algorithm, such as the one proposed in this paper, to fill the space between the convex hull of the grid, and its (possibly multi-connected) boundary with extra convex cells, as to complete the adjacency information. In other words, a transitive chain of in-front relationships between occluding cells would always exist. Note that the extra cells are used for sorting purposes only, and ignored during rendering.

The filler cells and its adjacencies need to be inserted into the adjacency graph in MPVO. Creating nodes in this graph is trivialy done by adding a node for each cell in the filler set. Edge insertions are more complex because they require adding adjacency relations between mesh and filler cells. Unlike the input mesh, where adjacencies were defined from one cell to one cell by a single face (an edge in the graph), the new adjacency relations in the presence of filler cells can be from one cell to various cells by each face.

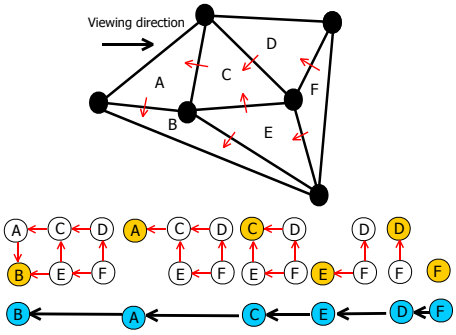

Figure 5: **MPVO algorithm**. The adjacencies define an adjacency graph. A topological sorting of this graph produces the cells in visibility ordering with respect to the viewing direction.

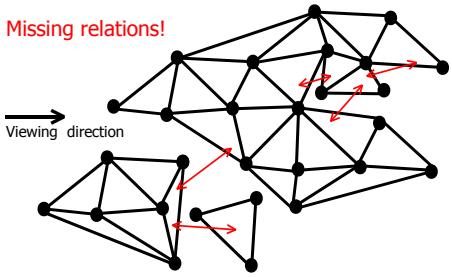

Figure 6: **MPVO missing relations.**

Discussion

Our convexification algorithm automatically fills the empty space surrounding a non-convex model with cells (also called filler cells) in order to form a single convex model, allowing the MPVO algorithm to be used. This approach is advantageous when the complexity added by the additional cells can be compensated by the speed-up gained by using a faster algorithm for convex cells (e.g., MPVO) instead of a non-convex algorithm (e.g., SXMPVO or BSP-XMPVO).

One of the limitations of our approach is that it does not guarantee to generate convexifications that are cycle free. In some datasets, such as the SPX, there are cycles formed that include a sequence of mesh-filler-mesh or mesh-filler-mesh adjacencies. This can be explained by the fact that BSP-trees imposes a total ordering only at the convex cells it creates, but not at objects that are split by BSP-tree cuts. Therefore, the choosing of cuts in the BSP-tree is critical to avoid the creation of such cycles.

5 Results

All results produced in this section were obtained in a Pentium 4 1.4GHz with 1GB of memory (a Dell Precision 330). We explored several BSP-tree construction techniques in the BSP-tree used by the filler algorithm. Overall, our goal was to produce a BSP-tree that has the fewer number of outside leaf nodes, while being as balanced as possible. We used the following heuristic to pick the next cut in the BSP-tree Let $c_{in}(f)$, $c_{out}(f)$ and $c_{cross}(f)$ count the number of faces that lie in the inside, outside and both hyperplanes of a candidate cut. Let $score(f) = 2 * c_{cross} + abs(c_{in} - c_{out})$. Among a user-defined number of candidates we chose the one that minimized this score.

Filler Statistics					
Mesh	Faces	FillCells	FillFaces	Avg(Adj)	Max(Adj)
DODEC	12	12	42	3.1	4
MUSH	240	396	2269	14	19
BUNNY	948	810	5063	5.68	15
SPX	2760	3032	17698	9.00	50
SPX,k=3	2760	2787	16283	8.25	43
SPX2,k=3	5520	5845	25872	8.75	40
SPX4,k=3	11040	11417	67333	7.21	59
BLUNT	13516	62	284	0.015	21

Figure 8: Filler construction statistics for various datasets. Unless mentioned, the number of kdtree levels (k) is 0. Note that for SPX we obtain fewer cells with 3 kdtree levels. SPX2 contains two copies, side by side, of the SPX dataset. SPX4 contains four copies.

We also experimented with adding external cuts parallel to coordinate axes, in the same way that a kd-tree does. These cuts are added before the boundary cuts in the BSP-tree. We chose to add kd-tree cuts until a certain level of the tree, which is a parameter to the construction. We use as heuristic to choose the kd-tree axis as the one that has the greatest variance in the vertices of the faces, and cuts the space at the median of chosen axis.

The boundary faces input to the filler algorithm are assumed to be defined along a consistent orientation (clockwise or anti-clockwise), which is used to define inside and outside cells. We tested the performance of the algorithm with the boundary extracted from a volumetric dataset (BLUNT), and with other boundary files like a dodecahedron, a mushroom and the Stanford bunny. We display results in Figure 8.

We used the filler data to augment the MPVO adjacency graph to obtain the visibility ordering of the BLUNT dataset. We call it the SF-MPVO algorithm (Space Filled MPVO). For the BLUNT dataset, a model with 187K tetrahedron, fewer filler cells were added to the models as it is almost convex. For this dataset, we produce an ordering in 0.71s, which is comparable with the results of MPVONC, which corresponds to MPVO augmented with an additional sort step that works for most but not all cases. It is important to note that the increase in the MPVO adjacency graph is not substantial, which explains the fact that the sorting times among MPVONC and SF-MPVO are almost equivalent. In figures 9 and 10 we illustrate our algorithm in action, with the display of the input mesh and filler cells generated, followed by a sequence of images that show the ordering produced by the SF-MPV0 algorithm. A video showing an animation of this process is also included with this submission.

6 Conclusions and Future Work

In this paper we presented an automatic and numerically stable way to the convexification of space problem, used to accelerate the volume rendering of unstructured grids. We chose a BSP-tree as the supporting structure for this problem, using the boundary faces as cuts in space. Enumerating the geometry and topology of empty space can be done very efficiently with the filler algorithm. The resulting convex cells that form empty space are incorporated into the MPVO algorithm with little overhead in the number of cells. Results shown that the performance obtained is compatible with the performance of convex algorithms. An alternative technique would be to instantiate the convex cells generated by the BSP-tree completely numerically, and to use these cells as input to MPVO. This has shortcomings. First, the construction of such cells is computationally expensive. Also, the numerical instabilities associated with such a process make it hard to use the output, (e.g., to apply MPVO,

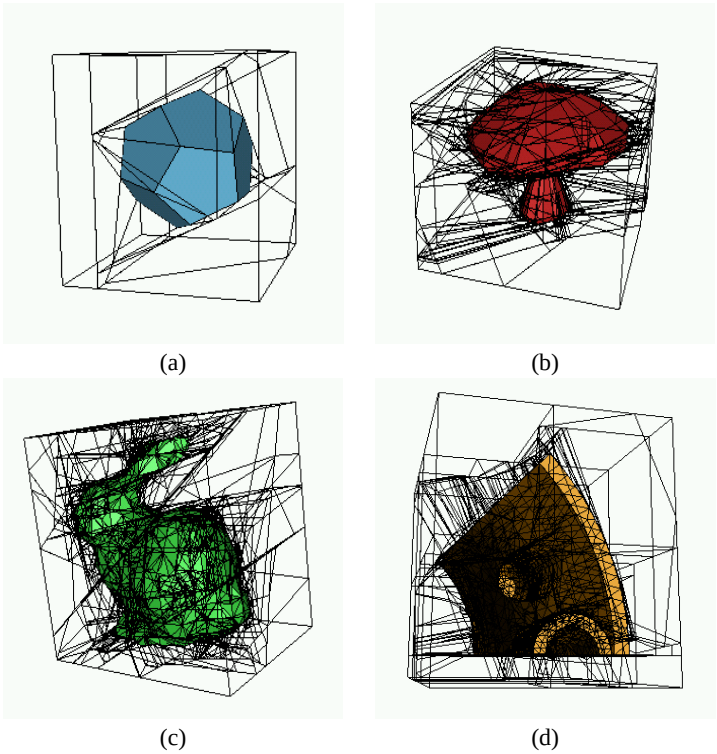

Figure 7: (a)-(d) Examples of filler decompositions in the dodecahedron, mushroon, bunny and SPX datasets.

it is necessary to compute connectivity information from possibly inexact cells).

In future work, we are exploring extensions of the algorithm for dynamic scenes, and ways to simplify even further the number of cells generated by the filler algorithm. There are interesting theoretical questions associated with the filling process. An important question is to compute non-trivial theoretical bounds on the number of filling cells. Another important questions is related to the ordering properties of the complex after convexification. It is possible to design examples which show that our technique can potentially generate cycles as part of the filling process. Although not actually a problem in practice[1], it would be nice to understand this better.

REFERENCES

[1] J. Comba, James T. Klosowski, Nelson Max, Joseph S. B. Mitchell, C. T. Silva, and Peter L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):??–??, September 1999.

[2] Richard Cook, Nelson Max, Claudio T. Silva, and Peter Williams. Efficient, exact visibility ordering of unstructured meshes. Tech Report, Lawrence Livermore National Labs, 2001.

[3] E. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge, 2001.

[4] M. Kraus and T. Ertl. Cell-projection of cyclic meshes. In *IEEE Visualization 2001*, pages 215–222, October 2001. ISBN 0-7803-7200-x.

[5] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 27–33, November 1990.

[6] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[7] Jonathan R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 86–95, 1998.

[8] Jonathan Richard Shewchuk. Sweep algorithms for constructing higher-dimensional constrained Delaunay triangulations. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 350–359, 2000.

[9] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 63–70, November 1990.

[10] C. T. Silva, Joseph S. B. Mitchell, and Peter L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *IEEE Symposium on Volume Visualization*, pages 87–94. IEEE, ACM SIGGRAPH, 1998.

[11] Peter L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[1] MPVO reports when cycles occur. In that case, it would be easy to revert to a slower algorithm, e.g. BSP-XMPVO, and still maintain correctness.

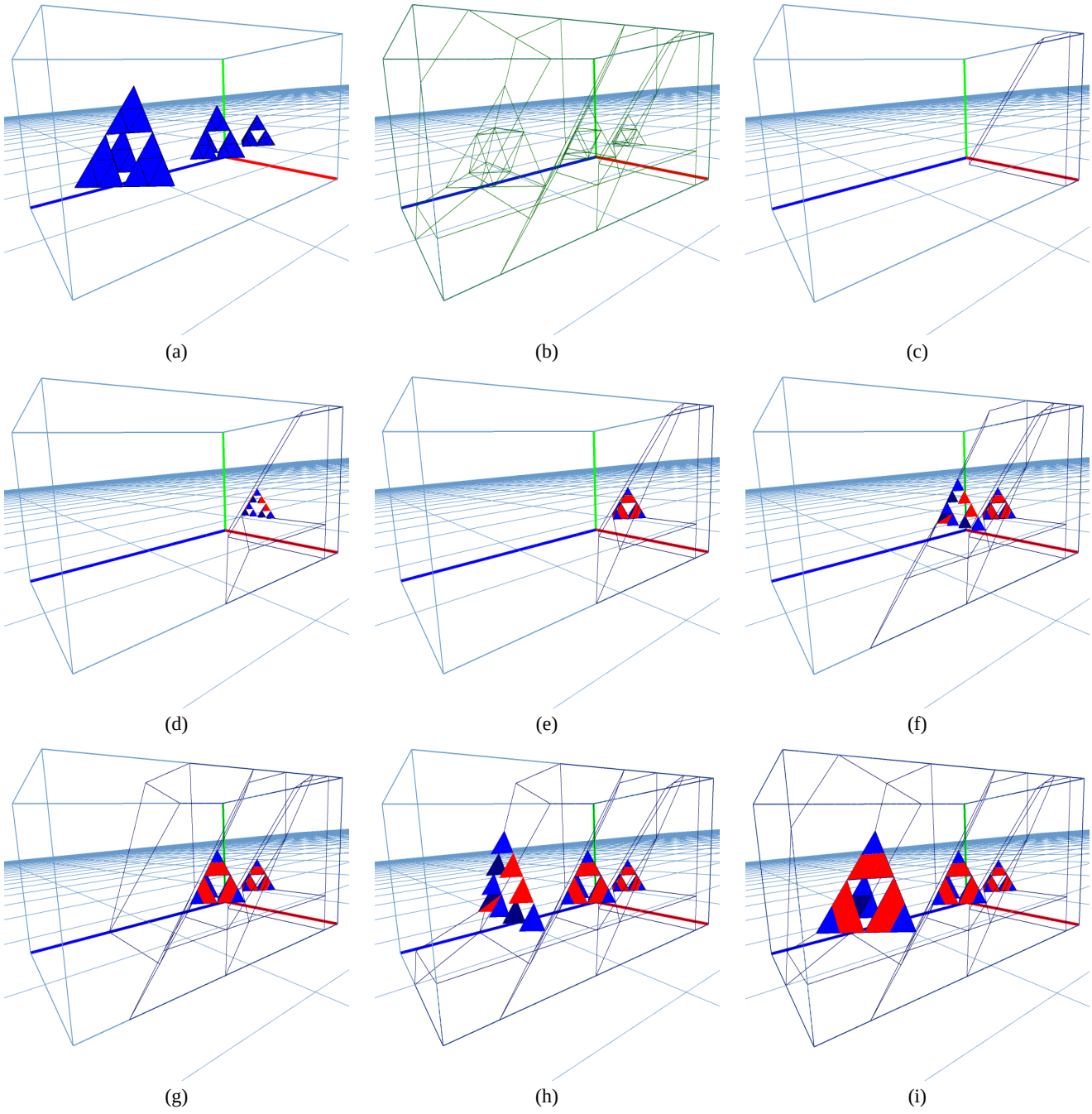

Figure 9: SF-MPVO in a dataset with disjoint parts and holes. (a) Input mesh (b) BSP-tree subdivision (c)-(i) Several instances of the algorithm showing the cells projected so far (filler meshes are drawn in wireframe)

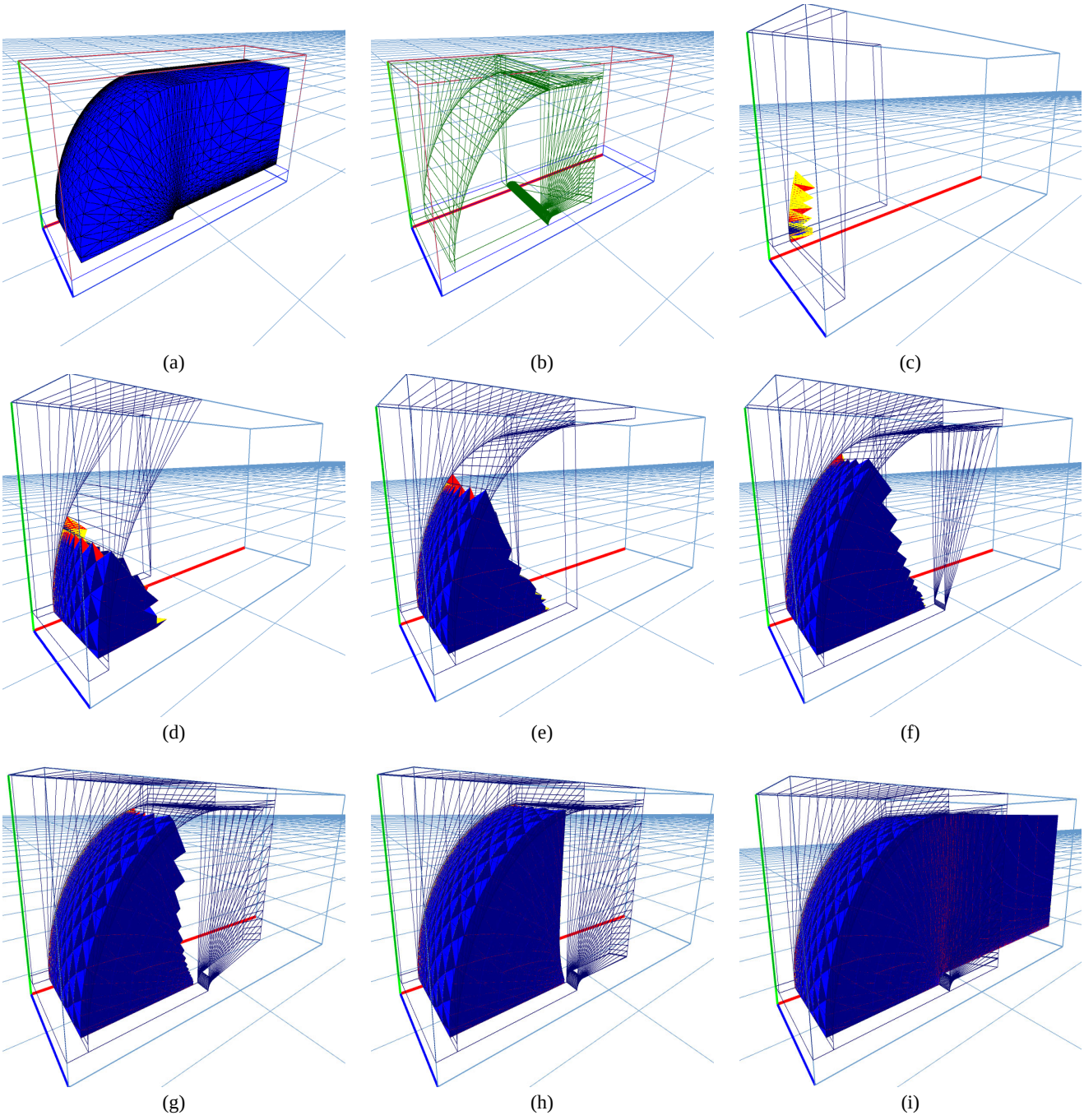

Figure 10: SF-MPVO in the BLUNT dataset. (a) Input mesh (b) BSP-tree subdivision (c)-(i) Several instances of the algorithm showing the cells projected so far (filler meshes are drawn in wireframe)