An Efficient Geometric Algorithm for Clipping and Capping Solid Triangle Meshes

Aaron Scherzinger, Tobias Brix and Klaus H. Hinrichs
Department of Computer Science, University of Münster, Münster, Germany

Keywords: Clipping, Two-Manifold Triangle Meshes, Geometry Processing.

Abstract: Clipping three-dimensional geometry by arbitrarily oriented planes is a common operation in computer graphics and visualization applications. In most cases, the geometry used in those applications is provided as surface models consisting of triangles which are called meshes. Clipping such surface models by a plane cuts them open, destroying the illusion of a solid object. Often this is not desirable, and the resulting mesh should again be a closed surface model, e.g., when generating cross-sections in technical visualization applications. We propose an algorithm which performs the clipping operation geometrically for a given input mesh on the GPU. The intersection edges of the mesh and the clipping plane are then transferred to the CPU, where a cap geometry closing the mesh is computed and eventually added to the clipped mesh. Our algorithm can process solid (i.e., closed two-manifold) triangle meshes, or sets of non-intersecting solids, and has a worst-case runtime of $O(N + n\log n)$ where N is the number of triangles in the input geometry, and n is the number of input triangles intersecting the clipping plane.

1 INTRODUCTION

In computer graphics and visualization, applying clipping planes to a given geometry is a standard operation. It is usually implemented in rendering systems and application programming interfaces (APIs), as clipping geometric objects against the planes of a viewing frustum is essential for the rasterization process. However, for some applications it is desirable to provide the option of performing additional clipping operations with user-defined clipping planes. For instance, such functionality is required in technical visualization like *computer-aided design* (CAD), or in medical visualization. Moreover, these application domains often require interactive frame rates, allowing the user to modify the parameters of the plane in real-time, while generating high-quality images.

Usually, the geometry used in computer graphics applications is provided as surface models composed of triangles which are called meshes. Such meshes are restricted to surface representations and thus do not contain any information about the interior of an object. Clipping such surface models by a plane cuts them open, and although it might be intentional in some cases to obtain an open geometry, for several application domains it is desirable to maintain the illusion of a solid object, which requires closing the mesh after performing the clipping operation. Especially in CAD applications, this is a common requirement when producing cross-sections of solid geometric objects to resolve spatial occlusion, allowing to examine the internal structures of assemblies. This process is referred to as *solid-clipping* (see Fig. 1). However,

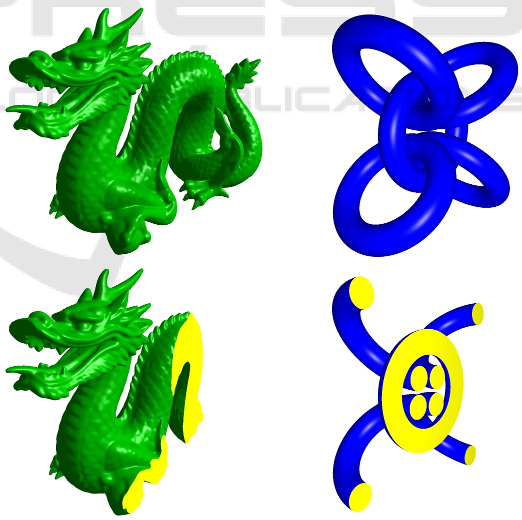


Figure 1: Example of solid-clipping the Stanford dragon (left) and a test mesh consisting of several tori (right). In both cases, the closing geometry is highlighted in yellow.

Scherzinger A., Brix T. and H. Hinrichs K.
An Efficient Geometric Algorithm for Clipping and Capping Solid Triangle Meshes.
DOI: 10.5220/0006097201870194
In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017)*, pages 187-194
ISBN: 978-989-758-224-0

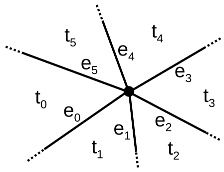


Figure 2: In a closed two-manifold triangle mesh, the triangles incident to a vertex form a closed triangle fan and can be arranged in a cyclic order.

closing a model after performing the clipping operation requires special effort, as additional faces have to be created and rendered, and the methods for achieving this are usually not general, requiring specific data structures and sophisticated algorithms.

In this paper, we propose an algorithm which performs the solid-clipping process geometrically for input meshes that are closed two-manifold triangle meshes (see Sec. 2). Our algorithm performs the clipping of the input mesh on the GPU. The intersecting edges between the triangles of the mesh and the clipping plane are then transferred to the CPU, while the clipped mesh can be kept in graphics memory. From the set of edges, a *cap mesh* is computed on the CPU which closes the clipped mesh. Possible applications of the algorithm include clipping of meshes for stereolithography, CAD applications, and isosurfaces extracted from volumetric data sets, where Lewiner et al. (Lewiner et al., 2003) have proposed an extension to the marching cubes algorithm to guarantee topological consistency for the resulting surface.

2 FUNDAMENTAL NOTIONS

This section includes some fundamental notions which will be required in the remainder of the paper.

Definition 1 (Closed Two-Manifold Triangle Mesh). *A finite triangle mesh is called a* closed two-manifold triangle mesh *if every edge of the mesh is shared between exactly two of its triangles, and if for every vertex of the mesh the triangles incident to the vertex form a closed fan, i.e., the edges e_i and triangles t_j incident to the vertex can be arranged in a cyclic order $t_0, e_0, t_1, e_1, \dots, t_{n-1}, e_{n-1}$ without repetitions such that edge e_i is shared between t_i and t_{i+1} (indices taken* mod n*). Other than in shared vertices or edges, there is no intersection of triangles in the mesh. A triangle mesh that is a closed two-manifold is also called* solid.

An example of the triangle and edge arrangement around a vertex in Def. 1 is depicted in Fig. 2. In addition to the constraints given in Def. 1, we assume

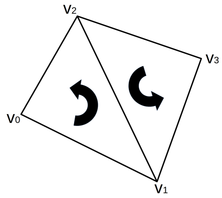


Figure 3: The two CCW triangles (v_0, v_1, v_2) and (v_3, v_2, v_1) are consistently oriented, since the shared edge has orientation $\overline{v_1 v_2}$ in the first triangle and orientation $\overline{v_2 v_1}$ in the second triangle.

that the triangles in the mesh are *consistently oriented*, which we will define as follows.

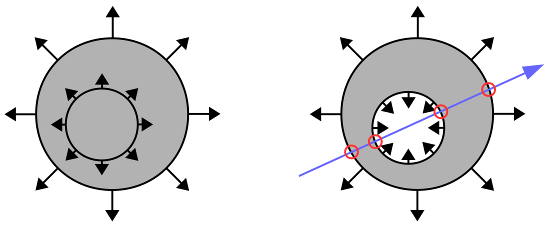
Definition 2 (Consistently Oriented Triangle Mesh). *A triangle mesh is called* consistently oriented *if two triangles sharing an edge induce opposite directions on the shared edge.*

As a convention, we will assume that the orientation of each triangle is chosen so that its vertices are ordered counter-clockwise (CCW) with respect to the triangle's front face normal vector, i.e., the direction of the surface normal of a triangle (v_0, v_1, v_2) is given by the cross product $(v_1 - v_0) \times (v_2 - v_0)$. An example of a consistent orientation is depicted in Fig. 3.

It should be noted that the definition above allows meshes to consist of multiple unconnected components, where some may constitute holes or hollows within a solid object. While our algorithm is inherently able to handle such cases, we assume that the orientation of the triangles is consistent with the surface normals when constituting holes and exclude cases where this condition is not fulfilled (which would contradict the intuition of modeling solid objects, since an interior surface representing a hole would have an outwards normal). An example would be a solid sphere within a larger solid sphere where the surface normals of both spheres point to the same outward direction, which is depicted in Fig. 4(a). This additional constraint can be formalized as follows.

Definition 3 (Solid Triangle Mesh Constraint). *For each ray R which intersects a consistently oriented two-manifold triangle mesh $\mathcal{M}$ holds that it is either a degenerate case, i.e., R intersects at least one edge or vertex of a triangle of $\mathcal{M}$, or the set of triangles of $\mathcal{M}$ that are intersected by R and ordered along the direction of R are given by $t_0, t_1, \dots, t_k$ so that for each triangle $t_i, i = 2 \cdot m$, the angle between the direction vector of the ray d_R and the triangle's normal n_i is greater than $90°$, i.e., $d_R \cdot n_i < 0$, while for each triangle $t_j, j = 2 \cdot m + 1$, the angle is less than $90°$, i.e., $d_R \cdot n_i > 0$.*

Intuitively, the above constraint corresponds to the idea that a ray intersecting the geometry will alternate



(a) Example of two solids stacked into each other. The surface normals are depicted by arrows.

(b) A ray intersecting a solid alternates between being outside of the solid and within the solid.

Figure 4: Solid modeling constraints.

between being outside of the solid and being inside the solid each time it intersects the surface given by the triangles of the mesh (see Fig. 4(b)). If this assumption holds, cases such as in Fig. 4(a) can also be handled by the algorithm when introducing an additional processing step without changing the overall asymptotic runtime (see Sec. 4.4).

3 RELATED WORK

Several approaches to realize solid-clipping have been proposed. Probably the most common method is an image-based technique which utilizes the stencil buffer of the graphics hardware to render additional geometry, thus conveying the impression of solid objects (McReynolds and Blythe, 2005). This approach is often referred to as *capping*, and the geometric objects embedded in the clipping plane which are rendered additionally to close the mesh are called *cap polygons*. For technical visualization, other image-based techniques have been proposed, such as the interactive view-dependent cutaway rendering by Burns and Finkelstein (Burns and Finkelstein, 2008). Trapp and Döllner (Trapp and Döllner, 2013) have proposed a technique which allows the use of more complex clipping surfaces by applying an offset map to the plane. Despite the fact that capping techniques allow to efficiently render images of the clipped objects, a general drawback is that they only construct the cap polygons in image space and do not allow to retrieve a clipped and closed version of the surface model for further geometric computations.

An alternative approach which solves the problem of solid-clipping geometrically is the use of *constructive solid geometry (CSG)*, which provides a set of operations on *boundary representations (b-reps)* of the geometry (Foley et al., 1996). While providing a flexible and general toolbox for geometry processing, those methods often require a specific representation of the geometric objects explicitly storing topological information such as the *winged-edge* data structure or similar representations.

Weiskopf et al. (Weiskopf et al., 2003) have proposed a technique for interactive clipping of volumetric data sets in texture-based volume rendering applications. Using volumetric representations of both the input geometry and the clipping objects, voxel-based methods provide a lot of flexibility regarding the shape of the clipping objects and allow to retrieve the new geometry after performing the clipping process. Unfortunately, in order to apply these techniques to a triangle mesh, they require the input mesh to be converted to a voxel representation via a *voxelization* method such as the ones proposed in (Huang et al., 1998) or (Schwarz and Seidel, 2010). Additionally, if a surface model representation of the clipped geometry is required afterwards for further processing, the obtained voxel representation has to be converted back again after performing the clipping.

Erleben and Henriksen (Erleben and Henriksen, 2006) have proposed a geometric algorithm for clipping a solid mesh composed of convex faces and closing the geometry afterwards. However, their approach requires an input mesh representation similar to a *half-edge* data structure. Moreover, the runtime of their algorithm is $O(n^2)$, which might be problematic in real-time applications.

We propose a geometric algorithm for interactively clipping solid triangle meshes and subsequently closing these objects. The output of our algorithm is again a solid triangle mesh which allows to store or further process the geometry data after the solid-clipping operation. Our algorithm does not need any pre-processing phase to convert the input geometry to a specific representation as the clipping is performed on the individual triangles and solves the problem in a worst-case runtime of $O(N + n \log n)$ where N is the number of triangles in the input geometry and n is the number of triangles in the input geometry that actually intersect the clipping plane. Since the clipping is performed in parallel on the GPU, the $O(N)$ step can be computed efficiently in practice.

4 PROPOSED ALGORITHM

4.1 Workflow

An overview of the workflow for our proposed method is depicted in Fig. 5. First, the clipping of the input mesh against a plane is performed in parallel for each individual triangle on the GPU (see Sec. 4.2). We use OpenGL's *transform feedback* functionality to

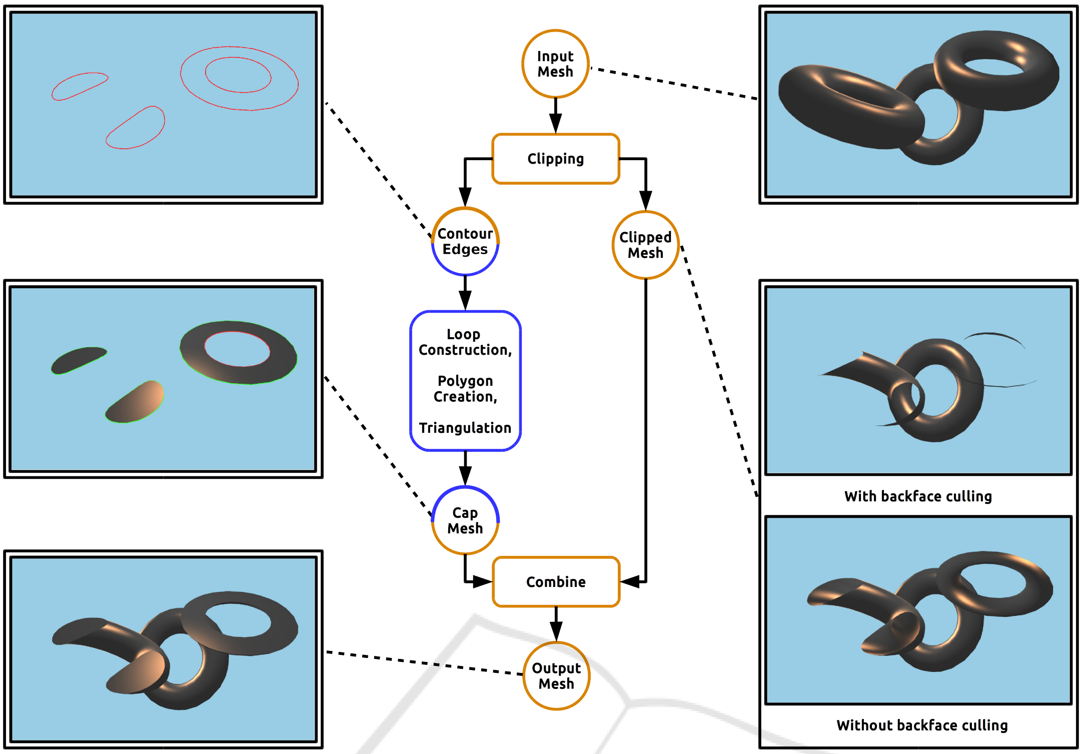


Figure 5: Workflow of our proposed algorithm. Boxes denote operations of the algorithm while circles correspond to meshes, which are illustrated by the rendered examples. Operations and data on the GPU are depicted using the orange color while operations and data on the CPU are depicted in blue. Circles with both colors denote data transfer between CPU and GPU.

retrieve both the clipped mesh as well as the intersection geometry of the mesh and the clipping plane into separate buffers. The intersection corresponds to a set of *contour edges* which are downloaded to the CPU for creating the cap mesh while the clipped version of the input mesh is kept on the GPU.

From the (unordered) set of edges, we then construct a set of *loops*, i.e., closed polygonal chains which describe the boundaries of polygonal regions embedded in the clipping plane (see Sec. 4.3). Depending on its orientation, each loop may constitute either the outer boundary of a polygon or a hole within such a polygon. In the next step, a hierarchy of the loops is computed which is then traversed to compute the actual set of polygons (potentially containing holes) which correspond to the planar geometry necessary to close the clipped input mesh and thus correspond to the capping polygons (see Sec. 4.4). Afterwards, a triangulation of those polygons is performed to construct the actual triangle cap mesh (see Sec. 4.5). This mesh can then be transferred to GPU memory and closes the clipped input mesh, so that their combination constitutes the solid-clipped mesh which again is a solid triangle mesh. Examples of the geometries resulting from the different stages of the workflow are depicted in Fig. 5.

4.2 Clipping

Clipping a triangle against a plane with the normal $n = (n_x, n_y, n_z), |n| = 1$, and (signed) distance d from the origin computes the intersection of the triangle with the half-space $(x, y, z) \cdot n - d > 0$ bounded by the clipping plane, which can either be the empty set, a triangle, or a quadrilateral (which can be decomposed into two triangles). It should be noted that we define the half-space as being strictly positive which avoids special cases like triangles lying exactly in the clipping plane or the existence of degenerated triangles after the clipping operation.

We assume that an input mesh is given as a GPU representation for rendering, e.g., an OpenGL buffer object, and compute the intersection of the triangle and the positive half-space using a geometry shader. The implementation relies on the GPU triangle clipping method proposed by McGuire in (McGuire, 2011), which is based on an algorithm proposed by Sutherland and Hodgman (Sutherland and Hodgman, 1974). For triangles, their method can be reduced to a few specific cases which are depicted in Fig. 6. Triangles can either be discarded or completely retained, or have to be clipped against the plane (cases (3) and (4) in Fig. 6). When clipping a triangle, two vertices v_1'

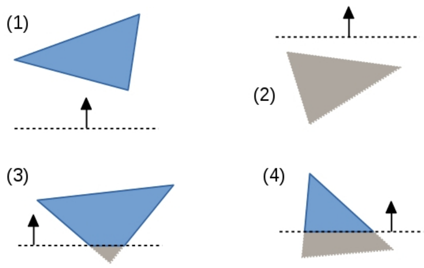


Figure 6: Examples of the four cases that may occur when a triangle is clipped against a plane. In case (1), the whole triangle is retained, while in case (2), it is completely discarded. In the other two cases the triangle is actually clipped, resulting in a quadrilateral in case (3) and in a triangle in case (4). The arrow represents the direction of the plane normal.

and v_2' corresponding to the intersection of two edges of the triangle and the clipping plane are computed using linear interpolation along the triangle's edges to construct either a quadrilateral or a triangle. For the actual shader code, we refer to the triangle clipping listing in McGuire's paper. When clipping a triangle, we linearly interpolate the vertex attributes such as colors, normals, or texture coordinates between the original vertices in the same way as the vertex positions. If the clipping operation yields a quadrilateral (case (3) in Fig. 6), we decompose it into two triangles with consistent orientation. As stated above, degenerated triangles are avoided by the convention of using the strictly positive half-space.

When a triangle is clipped against the plane, the two vertices v_1' and v_2' form an edge which is embedded in the clipping plane and corresponds to the intersection of the plane and the triangle. The geometry shader writes those edges and the clipped triangles into different output streams, storing the clipped mesh and the intersection edges in separate buffer objects. While we retain the original orientation of the clipped triangles, we invert the direction of the intersection edges so that a polygon constructed from those edges is consistently oriented to the clipped input mesh.

Since we compute the clipping operation on the GPU, the triangles can be processed very efficiently in parallel. However, if the mesh is not already present on the GPU or should not be uploaded for later rendering, the clipping can also be performed on the CPU by processing the individual triangles sequentially in $O(N)$ time. It should be noted that a trivial upper bound for the number n of triangles that intersect the plane, which corresponds to the number of intersection edges, is $O(N)$, although in practice the number of edges is usually much smaller than the overall number of triangles N.

4.3 Loop Construction

After performing the GPU clipping operation, we download the intersection edges to the CPU while keeping the triangles of the clipped mesh in graphics memory. Due to the parallelity of the clipping operation and the potentially random order of triangles in the input mesh, the list of edges is not given in any specific order. For a closed two-manifold triangle mesh, the set of intersection edges always constitutes a set of closed loops for which is guaranteed that the loops are not self-intersecting and that there are no intersections between different loops.

Theorem 1.
The intersection of a plane P and a finite consistently oriented closed two-manifold triangle mesh $\mathcal{M}$ is either the empty set, or a finite set of edges embedded in P, which form one or more closed loops without self-intersections. No pair of two edges contained in different loops intersects.

Proof. Let S be the set of edges that resulted from clipping $\mathcal{M}$ against P. Since each edge $s \in S$ is the intersection of a triangle $T \in \mathcal{M}$ and the plane P, it has to be embedded in P, i.e., $s \subset P$. The rest of the proof consists of two parts.

(i) *For each clipping edge $s = \overline{pq}, s \in S$, a successor edge $s' = \overline{qr}, s' \in S$, can be found.*

Let $A = (v_0, v_1, v_2)$, $A \in \mathcal{M}$, be a triangle for which at least one of its vertices $v \in \{v_0, v_1, v_2\}$ lies in the positive half-space bounded by P, and at least one of its vertices $w \in \{v_0, v_1, v_2\}, w \neq v$, lies in the negative half-space. Then the intersection of P and A constitutes a line segment $s \in S, s = \overline{pq}$, where p and q are the intersections of two edges e, f of A and P. W.l.o.g. let q be the intersection point of e and P.

Because of the characteristics of a consistently oriented two-manifold triangle mesh, there exists exactly one triangle $B \in \mathcal{M}, B \neq A$, which shares the edge e with A. Then the intersection of B and P also constitutes a line segment with q as one of its endpoints. Because of the consistent orientation, the edge e is inversely directed in B compared to A and thus s' has to be directed from q to a point r, i.e., $s' = \overline{qr}$.

Since for each clipping edge s a successor s' can be found and the mesh is finite, each edge is contained in a closed loop. The second part of the proof now considers the intersections between line segments.

(ii) *Except for two consecutive edges in a loop, which intersect in their common endpoint, no pair of two edges $e, f \in S, e \neq f$, intersect.*

Each edge $s \in S$ is the intersection of a triangle $T \in \mathcal{M}$ and P, which implies $s \subseteq T$. Let $A, B \in \mathcal{M}, A \neq B$ be two triangles that intersect the plane, and let $e, f \in S, e \neq f$ be the two corresponding intersection edges with the plane P. Then $e \cap f \neq \emptyset \Rightarrow A \cap B \neq \emptyset$, and if this is true, one of two cases has to apply:

(1) A and B share a common edge, which intersects P as in (i), and e and f are consecutive edges of a loop. ✓

(2) $\mathcal{M}$ is not a two-manifold triangle mesh, as it is self-intersecting. ↯

Since no pair of line segments is intersecting, except for the consecutive edges of a loop, and every edge is contained in a closed loop, Thm. 1 is correct. □

It should be noted that the orientation of the resulting loops is consistent with the mesh, meaning that a loop is CCW regarding the inverted normal of the clipping plane, i.e., the actual surface normal of the resulting cap polygon, if it constitutes the outer boundary of a polygon, and clockwise (CW) if it constitutes a hole. This follows directly from the convention we have chosen for the triangle orientation of the consistently oriented mesh, and the fact that the orientation of the edges retrieved from the clipping process is consistent with the clipped triangles of the input mesh. This property will be relevant for the subsequent steps presented later on.

Now that the existence of loops in the set of edges, and the membership of each edge in one of the loops, has been established, the loop construction algorithm will be outlined. Basically, the algorithm starts to create a new loop by picking an arbitrary edge from the set of contour edges. Afterwards, it searches for its successor in the set of edges, i.e., for an edge that has a start vertex with a position identical to that of the end vertex of the current edge, and appends it to the loop. If the end vertex of the new edge closes the loop, the process is started again with one of the remaining edges and a new loop. Otherwise, the new edge becomes the current edge, for which a successor has to be found. The process is repeated until no edge is left that is not already part of a loop. To reduce the search time for a successor, in a first step the algorithm sorts the edges lexicographically in ascending order with respect to the position of their start vertices. This allows to perform binary search to find an edge by its start vertex position. Algorithm 1 shows the complete loop construction procedure.

Since Thm. 1 states that the number of edges is finite, each edge has a successor, and every loop is closed, the algorithm eventually terminates, as it examines each edge exactly once before appending it to a loop and removing it from the input list. If the lexicographical sorting is implemented by inserting all edges into a balanced tree (e.g., an *AVL tree*), the algorithm has a worst case time complexity of $O(m \log m)$ for m input edges. This upper bound can be established by analyzing the actions the algorithm takes for each edge:

Algorithm 1: Loop Construction.

```
Input: Vector<Edge> edges
Output: List<Loop> loops

LexicographicSortByStartPosition(edges)
while edges not empty do
    Loop l = new empty Loop
    Edge currentEdge = edges.front()
    l.append(currentEdge.start)
    edges.erase(currentEdge)
    while currentEdge.end.position ≠
    l.front().position do
        Edge next =
        findSuccessor(edges,currentEdge)
        currentEdge = next
        l.append(currentEdge.start)
        edges.erase(currentEdge)
    loops.append(l)
```

1. Each edge is selected by either using `front` (which selects the first edge in the list, i.e., leftmost element in the tree), or `findSuccessor`. In an AVL tree, both of these operations can be executed in $O(\log m)$ time.
2. Each edge is erased from the tree, which takes $O(\log m)$ time.
3. Each edge is appended to the end of a list (i.e., the current loop), which can be realized in $O(1)$ time.

Since each edge is only handled once and is directly erased from the tree afterwards, processing an edge takes $O(\log m)$ time. All of the m edges are thus processed in $O(m \log m)$ time. Inserting the edges into the tree at the start also takes $O(m \log m)$ time. The complete loop construction algorithm runs therefore in $O(m \log m)$ time. Since the number m of edges corresponds to the number n of triangles intersecting the clipping plane, this can also be written as $O(n \log n)$.

4.4 Polygon Creation

For convenience, we transform all of the edges, i.e., their vertices, into the xy-plane for the next two steps, which allows us to perform the polygon creation and triangulation steps in $\mathbb{R}^2$. This can be realized by a rotation matrix which can be computed from the clipping plane normal. Optionally, an additional translation and scaling of the vertices can be performed

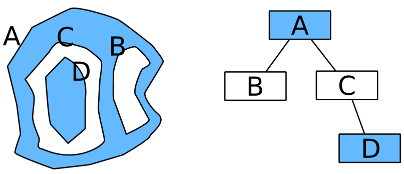


Figure 7: The contour A corresponds to the outer boundary of a polygon, while the contours B and C correspond to holes in that polygon. The contour D is nested within a hole and corresponds to a new polygon.

to normalize the coordinates. It should be noted that along with the projected vertices, we retain the original vertex positions in $\mathbb{R}^3$ which will be used for the output mesh to avoid numerical issues that might arise from transforming the vertices back and forth.

Since some of the loops reconstructed from the set of edges may constitute the outer contours of polygons while others may correspond to holes within those polygons, we need to compute the hierarchical ordering of the loops, which is called *polygon nesting* (see Fig. 7). Note that not all contours have to be nested in a single hierarchy. Instead, there may exist several trees like the one in the example, which form a *forest of trees*. Such a forest always exists and the hierarchical ordering is unambiguous, since no intersections between two different loops exist. Each loop is thus either the root of a tree or has a parent loop which completely contains it. Moreover, each root corresponds to the outer contour of a polygon, since the outermost loop cannot constitute a hole.

Due to the characteristics of a consistently oriented solid triangle mesh and the additional constraint given in Def. 3, each contour at the root of a tree as well as each contour with an even number of predecessors has a CCW orientation and corresponds to the outer boundary of a polygon, while each contour with an uneven number of predecessors has a CW orientation and corresponds to a hole. This becomes apparent when examining rays intersecting the geometry as in Def. 3 which lie in the clipping plane. Each of those rays alternates between the outside and the inside of the solid each time it crosses one of the edges.

To efficiently compute the polygon nesting structure, we apply the algorithm proposed by Bajaj and Dey (Bajaj and Dey, 1990). Their method computes the hierarchy of the polygons in $O(k+(l+r)\ log(l+r))$ where k is the number of input vertices of the polygons, l is the number of polygons, and r is the number of *reflex vertices*, i.e., vertices with an inner angle $> 180°$, in the set of vertices. Since l and r are much smaller than k, the algorithm runs faster than $O(k \log k)$ in practice. It should be noted that the number k of vertices equals the number n of contour edges, so that the runtime of the algorithm is in $O(n \log n)$.

First, the algorithm breaks all of the loops into *subchains*, i.e., it partitions each loop into x-monotone sequences of vertices, which can be done in $O(k)$ time. These subchains are sorted lexicographically by their endpoints from left to right. The algorithm then performs a *plane sweep* using a sweep line L while maintaining the vertical ordering $\mathcal{O}$ of the subchains induced by L. The sweep line stops at each endpoint of a subchain and updates the ordering $\mathcal{O}$ to determine the parent of each loop. The output of the algorithm is a directed acyclic graph G which contains a node for each loop L and corresponds to the nesting structure of the contours. For details about the algorithm, we refer the reader to the original paper by Bajaj and Dey. After computing the nesting structure, it can easily be traversed to extract each individual polygon together with its holes. During this traversal, an additional step could be integrated to correct the orientation of inner loops corresponding to holes when dealing with cases such as the one depicted in Fig. 4(a) (see Sec. 2) by inverting the orientation of the loops at each other level of the hierarchy, producing an alternating CCW-CW order in the sequence of levels of the nesting structure. This only requires $O(n)$ additional computation time and thus does not change the overall runtime complexity.

4.5 Triangulation

Polygon triangulation is one of the fundamental problems in computational geometry. Chazelle was the first to propose a linear-time algorithm for triangulation of simple polygons (Chazelle, 1991). However, his method does not seem to be applicable in practice. Instead, usually one of several existing $O(k \log k)$ (where k is the number of input vertices) methods for polygon triangulation is applied. Here, we use a two-step method which consists of partitioning the polygon into y-monotone pieces in $O(k \log k)$ and afterwards triangulating each of the pieces in $O(k)$ time. The method to partition a polygon into monotone pieces is due to Lee and Preparata (Lee and Preparata, 1977) and the linear time algorithm for triangulating a monotone polygon has been proposed by Garey et al. (Garey et al., 1978). The complete algorithm is summarized in (de Berg et al., 2008).

For the monotone partitioning of the input polygon in the first step, vertices are classified into various types and sources of non-monotonicity are removed by adding additional diagonals, splitting the polygon into monotone pieces. The algorithm performs this partitioning operation using a plane sweep which has a runtime complexity of $O(k \log k)$, where k is the number of input vertices. Afterwards, each monotone polygon resulting from the previous step can be triangulated in $O(k)$ time by iterating over the vertices in decreasing y-direction and connecting the left and the right chain. The overall worst-case runtime of the triangulation step is thus in $O(k \log k)$. Again, the number k of input vertices equals the number n of contour edges, so that the runtime is in $O(n \log n)$.

Since the orientation of the original edges is retained during triangulation, the cap mesh is consistently orientated with the clipped mesh. If the triangulation is not consistently oriented in itself, this can easily be corrected using an $O(n)$ scan over the list of triangles. Moreover, a simple runtime optimization of the triangulation can be added by testing each polygon for convexity and applying a very simple $O(n)$ triangulation for convex polygons instead of the aforementioned algorithm. However, this does not change the upper bound of the runtime complexity.

For the set of triangles resulting from the triangulation, the original $3D$ positions of the vertices (as computed during clipping) are used instead of the vertex positions transformed into the xy-plane. If the vertices contain normal vectors, the normal vector attribute for all of the vertices in the cap mesh is set to the inverted normal of the clipping plane, which corresponds to the normal vector of the cut surface. After finishing the triangulation step for all polygons, the cap mesh is complete. The output geometry is then created by combining the cap mesh with the clipped input mesh (e.g., by uploading the cap mesh to the GPU). Since each contour edge of the cap mesh is shared between a triangle of the clipped input mesh and a triangle of the cap mesh, and the shared edges induce opposite direction in the meshes due to the output of the clipping step, the output mesh is a consistently oriented closed two-manifold triangle mesh. Since the runtime of the clipping step (if not performed in parallel) is $O(N)$ and each subsequent step of the algorithm is in $O(n \log n)$ where n is the number of triangles intersecting the plane, the algorithm has an overall worst case runtime of $O(N + n \log n)$.

5 CONCLUSION AND FUTURE WORK

We have proposed an efficient method for geometrically clipping and capping a closed two-manifold triangle mesh in $O(N + n \log n)$. Our method performs the clipping on the GPU and transfers the contour edges to the CPU, where the cap mesh is computed to close the clipped input mesh. One of the drawbacks of our algorithm is the numerical stability, which might be problematic particularly in the loop construction. However, this can be mitigated by using a small epsilon as an allowed distance between the end vertex of an edge and the start vertex of its successor. Another problem of our method may be the quality of the triangulation in the last step. In future work, we will try to improve this step by applying constrained Delaunay triangulation or use of additional Steiner vertices to increase the quality of the cap mesh.

REFERENCES

Bajaj, C. L. and Dey, T. K. (1990). Polygon nesting and robustness. *Inf. Proc. Lett.*, 35(1):23–32.

Burns, M. and Finkelstein, A. (2008). Adaptive cutaways for comprehensible rendering of polygonal scenes. *ACM Transactions on Graphics*, 27(5):154:1–154:7.

Chazelle, B. (1991). Triangulating a simple polygon in linear time. *Discrete & Comput. Geom.*, 6(5):485–524.

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition.

Erleben, K. and Henriksen, K. (2006). A simple plane patcher algorithm. Technical Report DIKU-TR-06/09, Department of Computer Science, University of Copenhagen.

Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1996). *Computer Graphics: Principles and Practice, 2nd ed. in C*. Addison-Wesley.

Garey, M. R., Johnson, D. S., Preparata, F. P., and Tarjan, R. E. (1978). Triangulating a simple polygon. *Inf. Proc. Let.*, 7(4):175–179.

Huang, J., Yagel, R., Filippov, V., and Kurzion, Y. (1998). An accurate method for voxelizing polygon meshes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 119–126. ACM.

Lee, D. T. and Preparata, F. P. (1977). Location of a point in a planar subdivision and its applications. *SIAM J. on Computing*, 6(3):594–606.

Lewiner, T., Lopes, H., Vieira, A. W., and Tavares, G. (2003). Efficient implementation of marching cubes' cases with topological guarantees. *J. of Graphics Tools*, 8:2003.

McGuire, M. (2011). Efficient triangle and quadrilateral clipping within shaders. *J. of Graphics, GPU, and Game Tools*, 15(4):216–224.

McReynolds, T. and Blythe, D. (2005). *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann.

Schwarz, M. and Seidel, H.-P. (2010). Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics*, 29(6):179:1–179:10.

Sutherland, I. E. and Hodgman, G. W. (1974). Reentrant polygon clipping. *Comm. of the ACM*, 17(1):32–42.

Trapp, M. and Döllner, J. (2013). 2.5d clip-surfaces for technical visualization. *J. of WSCG*, 21(1):89–96.

Weiskopf, D., Engel, K., and Ertl, T. (2003). Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312.