Visibility-driven Mesh Analysis and Visualization through Graph Cuts

Kaichi Zhou, Eugene Zhang, Jiří Bittner, and Peter Wonka

Abstract— In this paper we present an algorithm that operates on a triangular mesh and classifies each face of a triangle as either inside or outside. We present three example applications of this core algorithm: normal orientation, inside removal, and layer-based visualization. The distinguishing feature of our algorithm is its robustness even if a difficult input model that includes holes, coplanar triangles, intersecting triangles, and lost connectivity is given. Our algorithm works with the original triangles of the input model and uses sampling to construct a visibility graph that is then segmented using graph cut.

Index Terms—Interior/Exterior Classification, Normal Orientation, Layer Classification, Inside Removal, Graph Cut.

1 Introduction

We address the following problem: given a model as a potentially unstructured set of triangles $t_i \in T$ where each triangle t_i consists of two faces $t_{i,1}$ and $t_{i,2}$. As output we want to compute a classification of all triangle faces $t_{i,j}$ into either inside or outside. Such an approach has several applications and we will demonstrate three in this paper: the inside removal of architectural and mechanical models for faster visualization, normal orientation and layer-based visualization of multiple layers of geometry using transparency.

We believe that previous work does not use visibility to its full potential so that classification errors are likely in more difficult models. There are two existing approaches to using visibility for inside outside classification. 1) Rays are sampled from an outside bounding volume to classify geometry as outside (e.g. [4, 16], see Fig. 1 left). This approach has difficulties with cracks and with the fact that parts of the outside surface might not be visible from an enclosing bounding volume. 2) Rays are sampled from the outside to stab the whole model. The inside-outside classification changes with each intersection (e.g. [16], see Fig. 1 middle). This approach has also some difficulties with cracks, double sided triangles, self intersecting triangles, and coplanar triangles. In contrast, we observe that any ray path can be used to propagate inside-outside classifications, see Fig. 1 right.

Our solution is to use visibility analysis to establish connections between entities that we call half-space nodes. A half-space node can correspond to a single half-space point or a (possibly infinite) set of half-space points. A half-space point is an oriented sample on a triangle and consists of a point p_i in R^3 and hemisphere centered at p_i. The orientation of the hemisphere is decided by the normal of a triangle face at p_i. Visibility relationships between half space points are established through sampling using ray casting. The details of our algorithm include a solution on how and where to create half space nodes and how to sample rays to establish visibility relations. The second part of our approach is a classification using iterative graph cut. The main contributions of our work are the following:

- We propose a model preprocessing algorithm that can classify triangle faces into inside or outside even if the input contains coplanar polygons, self-intersections, two-sided triangles and cracks.
- We are the first to propose semi-automatic extensions to inside outside classification to increase robustness in case the automatic method is unspecified or the errors in the model are above a user defined tolerance.

- *Kaichi Zhou graduated from Arizona State University and is now with NVIDIA, E-mail: kaichi.zhou@gmail.com.*
- *Eugene Zhang is with Oregon State University, E-mail: zhange@eecs.oregonstate.edu*
- *Jiří Bittner is with Czech Technical University in Prague, E-mail: bittner@fel.cvut.cz*
- *Peter Wonka is with Arizona State University: E-mail: pwonka@gmail.com*

Manuscript received 31 March 2008; accepted 1 August 2008; posted online 19 October 2008; mailed on 13 October 2008.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

1.1 Challenges

There are several geometric configurations that we want to consider in this paper as explained in the following:

One sided and two sided triangles: A one sided triangle has one side on the inside and one side on the outside while a two sided triangle has both faces on the outside (or inside). The main difficulty is to ensure that the algorithm can detect holes and does not classify triangle faces as outside that are only visible through an unwanted crack or hole in the model (see Fig. 2 for an example illustration).

Intersections and coplanar triangles: The algorithm should be robust in the presence of self intersections and coplanar triangles (see Fig. 3 left and right).

User input: It is unlikely that all models can be handled without some user input. In many cases, such as with terrains the question of what is inside and what is outside is actually a modeling problem that requires user input (see Fig. 3 right). We want to make use of minimal user input to clarify ambiguous configurations and to improve robustness in difficult cases.

Inside-outside definitions: There are two fundamentally different definitions of an inside-outside classification: *view-based* and *object-based*. The *view-based* definition considers all triangle faces as outside that are seen along a straight line from a bounding region, e.g. sphere, around the object. The *object-based* definition considers all triangle faces as outside that can be seen along a poly-line from the bounding region. The line or poly-line cannot intersect other triangles or pass through cracks in the model. Please note that only the *object-based* definition establishes an inherent property of the object, while the *view-based* definition produces different results for different viewing regions. In this paper we focus on the *object-based* definition. The *view-based* definition is a from-region visibility problem that could be addressed with our previous work [21, 20] as a starting point.

1.2 Related Work

Surface based model repair: Surface based model repair algorithms operate directly on the input mesh and fix model errors by local modifications to the mesh. A large class of methods fixes the model errors by either stitching the boundaries of the patches together or filling the holes by inserting additional geometry [3, 1, 9, 12]. Murali and Funkhouser [15] construct a BSP tree using the input triangles and then use linear programming in order to classify the cells of the BSP either inside or outside, remove cracks and fill holes. However due

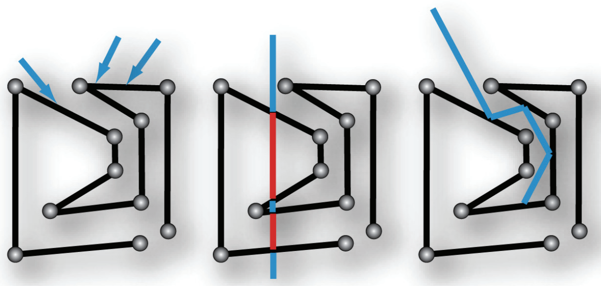


Fig. 1. Triangles (line segments) are shown in black, vertices as grey spheres, an outside classification is shown in blue and an inside classification is shown in red. Left: Classification of outside geometry by sampling visibility from a bounding sphere around the scene. Middle: stabbing the object along straight lines and alternating outside inside classification. Right: propagating inside and outside classification along an arbitrary path.

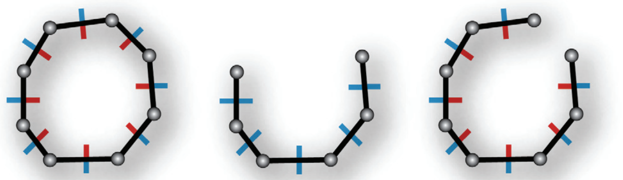


Fig. 2. Triangles (line segments) are shown in black, vertices as grey spheres, an outside classification is shown in blue and an inside classification is shown in red. Left: a typical example of a one sided object. Middle: a cup where each triangle has two outside faces. Right: this is a tricky example that could have multiple interpretations. We show the interpretation as one sided object with a hole.

to the BSP and linear programming the application of this approach to large models is costly and memory demanding. Our algorithm is closely related and partially inspired by the paper of Borodin et al. [4]. This paper also proposes to use visibility sampling for mesh analysis, but there are three issues that we want to improve upon: 1) The algorithm is not able to cope with coplanar polygons and intersections. 2) the algorithm computes neither a view-based nor an object-based inside-outside classification, but some mixed form. 3) The algorithm does not have a mechanism to correct sampling errors due to the ray tracer. This includes the omission of cosine-based sampling suggested by Zhang and Turk [21].

Volumetric model repair: In recent years a significant attention has been paid to techniques which repair a polygonal model by using an intermediate volumetric representation [16, 10, 2]. Nooruddin and Turk [16] convert the model into a volumetric grid. They perform inside outside classification of the cells of the grid by using ray casting. The grid is processed using morphological operations. Finally, the model is converted back to a polygonal representation. Ju [10] extends this approach by using an octree instead of a regular grid. This method uses a more efficient local approach for the inside outside classification of the cells of the octree. For the conversion to the polygonal model he suggests to use dual contouring which is able to better preserve sharp features of the model. Bischoff et al. [2] proposed a method which is able to robustly handle interpenetrating geometry or dangling triangles. This method also uses octree based voxelization, which is followed by hole filling using morphological operations and an extension of the dual contouring [11]. The common problem of volumetric methods is that they perform low pass filtering of the input geometry. Therefore they have problems with preserving sharp features and thin structures of the model.

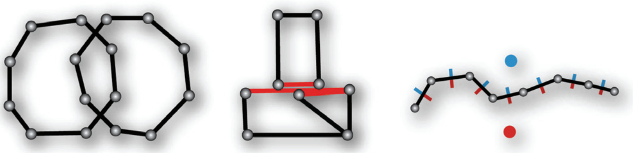


Fig. 3. Left: intersecting lines segments (triangles). Middle: a simple object showing three coplanar line segments (triangles). Right: A terrain has no inherent orientation. The user can specify the classification of two points in space to define inside and outside. Outside points are shown in blue and inside points in red. The algorithm suggested in this paper can correctly classify the triangles that are non-intersecting and non-coplanar. On the intersecting and coplanar triangles there would be a number of seed points (depending on the setting) that are classified as inside or outside.

1.3 Overview

The pipeline of our algorithm consists of four stages (see below):

$$\begin{tikzcd}\boxed{\text{Preprocessing}} \arrow[r] & \boxed{\text{Sampling}} \arrow[r] & \boxed{\text{Classification}} \arrow[r] & \boxed{\text{Applications}}\end{tikzcd}$$

Preprocessing: In the preprocessing step we load a model as triangle soup. In the basic version of the algorithm we mainly use three preprocessing steps. We build a kd-tree, we mark triangles that intersect other triangles, and we mark triangles that are coplanar with other triangles. Alternatively, we also use three more sophisticated preprocessing techniques that are optional. We can make use of connectivity and geometry information to construct clusters of triangles, we can compute exact triangle intersections, and we can remove coplanar triangles. The latter two computations are fairly difficult and not as robust as the other parts of the pipeline. For example, triangle intersections can often lead to many small additional triangles that are a disadvantage for many applications. See section 2 for details.

Sampling: In the sampling stage we create half-space nodes on triangle surfaces and shoot rays to sample visibility connections to other half space nodes. The methodology combines ideas from visibility, geometry, and global illumination to obtain a robust sampling strategy for the creation of half space nodes and the generation of rays. See section 3 for details.

Classification: The classification step analyzes the graph using a max-flow, min-cut algorithm to classify half space nodes as either inside or outside. Additionally, we provide a user input to set some global parameters of the classification, or to locally refine the computation in case of difficult geometric configurations. See section 4 for details.

Application and Results: The algorithm can be used for normal orientation, inside removal, or layer classification. We present a few more details on how to fine tune the pipeline for these applications in section 5 including a variety of results on selected example models.

2 Preprocessing

The input to our algorithm is a set of n_0 triangles t_i, with $1 \leq i \leq n_0$. In this section we explain six preprocessing steps: 1) kd-tree construction, 2) intersection testing, 3) coplanar testing, 4) intersection retriangulation, 5) coplanar triangle removal, and 6) patch clustering. The first three steps are required for all models and the second three are optional.

Our main philosophy is to establish a conservative and robust model processing algorithm. We found that the two optional preprocessing steps intersection retriangulation and coplanar triangle removal give undesirable results in many cases. The main focus of this paper is therefore to establish robustness despite geometric errors and inaccuracies rather then to fix these problems. A general problem with intersection retriangulation is the long implementation time for a correct algorithm and the many (often very thin) triangles that can be generated in the process (see Fig. 4 for an example intersection retriangulation). Coplanar triangle removal shares the same problems of intersection retriangulation. Additionally, it is unclear how coplanar triangles can be removed in textured models, because there are multiple textures or texture coordinates to choose from. As default we will assume that only the required steps are performed and will note additional optional preprocessing steps for each model.

Kd-tree construction: All triangles in the model are sorted into a kd-tree to accelerate ray casting. The kd-tree construction and the ray tracer are an improved version of multi-level ray tracing [17]. The kd-tree construction takes under two minutes for the largest model in the paper.

Intersection Test: The intersection test is performed for each triangle $t_i \in T$ and outputs a flag $inters_i \in \{true, false\}$ that is $true$ if the triangle intersects another triangle and is $false$ if the triangle does not intersect another triangle. The algorithm has to be conservative, i.e. two non-intersecting triangles can be incorrectly classify as intersecting but not the other way around. The reason for this conservative strategy is that the subsequent sampling and classification steps more carefully analyze intersecting triangles. Therefore, the first type of misclassification does not lead to any problems. We proceed as follows. We first use the kd-tree to select a list of intersection candidate triangles. We can find a set of candidate triangles $CSet_i$ by computing the bounding box B_i of triangle t_i and then taking all triangles that are stored with the leaf nodes of the kd-tree that intersect the bounding box B_i. For each triangle $t_j \in CSet_i$ we compute an intersection using the algorithm proposed by Moeller [13]. Alternate intersection routines are described in [14] chapter 13. We choose the first algorithm, because we found it easier to modify to make it conservative. The main idea of the conservative intersection tests is to introduce ε thresholds. We omit a very detailed description because it would be very lengthy due to many special cases and the ε intersection algorithm is not a contribution of our paper. Some of the special cases arise due to degenerate intersections and overcoming floating point limitations. Please note that we do not consider triangles sharing a vertex or an edge as intersecting. These cases are explicitly excluded in the implementation.

Coplanarity Test: The conservative coplanarity test for triangles is performed for each triangle t_i and outputs a flag $cpl_i \in \{true, false\}$ that is $true$ if the triangle is coplanar to another triangle and is $false$ otherwise. The test is conservative because we use ε thresholds as in the intersection computation. The algorithm proceeds as follows. We reuse the candidate set $CSet_i$ to test each triangle $t_j \in CSet_i$. We compute the angle α between the plane containing t_i and the plane containing t_j with a normal vector dot product. If α indicates parallel plane orientation we proceed to test the distance d_{ij} between the two closets points on t_i and t_j. If the distance is within an ε threshold we finally project t_i into the plane of t_j and use a two-dimensional version of the triangle intersection test [13]. If the triangles intersect they are both marked as coplanar.

Triangle Clustering: The idea of this stage is to cluster triangles that can be classified together. The clustering algorithm assumes that all connected triangles are part of orientable surface patches. The clustering algorithm is a simple greedy algorithm that needs two main parts: the clustering algorithm itself and a preprocessing algorithm that marks each triangle edge with a flag $\in true, false$ that indicates if triangles that share this edge can be clustered. In the following we first explain how to compute the edge flag and then give the algorithm outline. The edge flag is set to $true$ by default and then we set the flag to $false$ in the following cases: 1) the edge belongs to a triangle with $inters_i = true$ or 2) $cpl_i = true$; 3) the edge is non-manifold, i.e. it is shared by more then two triangles, and 4) the edge belongs to a triangle that is too close to another non-adjacent triangle.

The clustering algorithm marks all triangles of the model as not visited using a boolean flag per triangle. Iteratively, the first not visited triangle is selected to start a new cluster. The cluster is expanded by adding other triangles that share edges marked $true$ using breath first search. If there is only one triangle in a cluster we assign the cluster id $cluster_i = -1$ and otherwise we set the cluster id of all triangles in the cluster using a counter to ensure unique cluster ids.

Intersection Re-triangulation: The idea of intersection re-triangulation is to re-triangulate triangles so that no triangle is intersected by another one. While the output of this algorithm is a very helpful simplification of a general input, we believe that the actual use of the algorithm is controversial, because it can create many additional triangles and it is difficult to implement correctly. The triangle intersection algorithm for a triangle t_i proceeds similar to the intersection test and first computes intersection candidates and then triangle intersections. For each triangle intersection we store the intersection lines l_j in a set $LineS_i$. After all intersection lines are computed we retriangulate t_i using constrained Delauney triangulation [18]. More robust triangulations can be computed with the CGAL library [6]. See Fig. 4 for an examples.

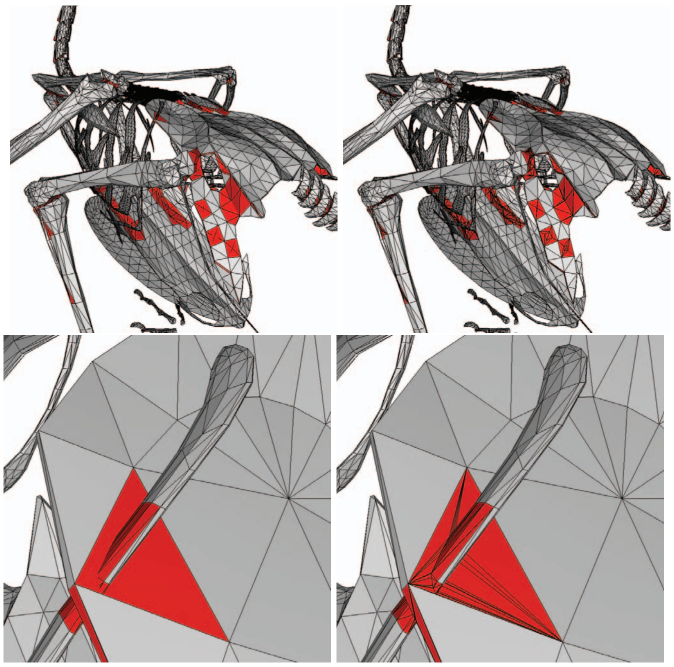


Fig. 4. The images on the left show the bird model before triangle splitting. The images on the right show the bird model after splitting. All triangles considered for splitting are shown in red. Please note the high number of thin polygons in the closeup view.

Coplanar Removal: The coplanar triangle removal uses the result of the coplanarity test. Coplanar triangles are projected into one plane and the resulting polygon is retriangulated.

Output of Preprocessing: At the end of preprocessing we obtain a set of n triangles t_i, with $1 \leq i \leq n$. If steps four and five are not performed $n = n_0$. Additionally, each triangle i has a flag $cpl_i \in \{true, false\}$ to denote if the triangle has other coplanar triangles and a flag $inters_i \in \{true, false\}$ to denote if the triangle is intersecting other triangles. If triangle clustering is used we store a cluster identifier $cluster_i$ with each triangle.

3 SAMPLING

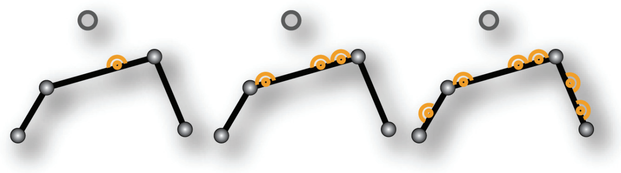


Fig. 5. Three cases of half-space nodes. Left: a single half-space point. Middle: the set of half-space points on one triangle face. Right: all half-space points on a cluster of triangle faces.

The goal of the sampling stage is to generate a visibility graph $G(V,E)$ consisting of *half-space nodes* $hsn_i \in V$, *virtual nodes* $\in V$, and edges $\in E$ that encode visibility connections between the nodes.

We divide the process into two steps: geometry sampling for node generation and visibility sampling. In principal the nodes in the graph correspond to geometry. We want to classify the nodes into inside and outside and then later transfer the node classification back to the geometry. These are then our two fundamental ideas on how to design such a graph:

1. The goal for the node design is to create nodes that correspond to geometry that can receive a consistent classification. This will result in a solution where more nodes are placed in difficult parts and fewer nodes are placed in easy parts of the model. For example, intersecting and coplanar triangles are difficult, because the visibility classification is expected to change within a triangle and we cannot assume that we can classify the faces of such a triangle consistently. On the other extreme, triangles within a triangle cluster will have two sets of faces so that the classification within each of the two sets is consistent.

2. The goal for edge design is to generate edge weights that measure how strong the visibility between two nodes is. Therefore, we propose to use a measure in ray space that corresponds to how many visible rays exist between two nodes.

3.1 Geometry Sampling

We define a *half-space point* as a point in R^3 and a set of viewing directions on a hemisphere Ω. Typically, the half-space point is a sample on a triangle and the hemisphere is defined by the normal vector of one of the two triangle faces.

A *half-space node* corresponds to either a single half-space point or a set of half-space points (see Fig. 5). We use different methods to generate half-space nodes for triangle clusters and the coplanar and intersected triangles:

Triangle Clusters: All the triangles belonging to a triangle cluster give rise to two sets of triangle faces fs_1 and fs_2 that will have the same inside-outside classification. Only two half space nodes are generated for a triangle cluster. One for all half space points on triangle faces in fs_1 and one for all half-space points on triangle faces in fs_2. Note that a single triangle is just a special case of triangle cluster.

Intersected and Coplanar Triangles: A predefined number of sample points are distributed over each intersected triangle. Each point contributes two half space nodes in the graph. We propose three sampling schemes to distribute the points: 1) Uniformly sample the triangle. 2) Uniformly sample the polygonal face, plus sample along the polygon's edges. 3) Compute all the intersection edges on the face (for coplanar faces, first project all the coplanars onto the interested face's plane), triangulate the face and uniformly sample the sub-faces.

The sampling quality increases from the first method to the third, but the time complexity also increases. In our experiments, we found that the second scheme generates good results for most of the models. To avoid potential numerical issues of ray casting, we need to detect whether the selected sample point is within an epsilon proximity of other triangles. For each of the other triangles intersected or coplanar with the current triangle, the distance from the point to its plane is computed and also the point is projected onto its plane to see if the point really falls inside the triangle. The half space point is offset along the normal of the furthest triangle that satisfies the proximity test above. Fig. 6 illustrates the adjustment.

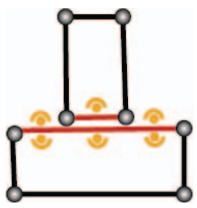


Fig. 6. Coplanar and intersected samples are adjusted to the surface of the furthest triangle that satisfies the proximity test.

Virtual Nodes: The graph always contains at least two virtual nodes, one of which represents the outside and the other represents the inside. Later, we will compute a cut in the graph to compute two disjunct sets of nodes. The set of nodes that contains the *outside node* (*inside node*) will be classified as outside (inside).

The sampling stage first creates the half-space nodes and then stochastically samples visibility between half-space nodes using ray casting. During this process a number of half-space points is generated for each half-space node. The details will be described in the next section.

3.2 Visibility Sampling

The constructed graph of halfspace nodes contains edges with weights. The weight of an edge connecting two halfspace nodes should reflect the amount of visibility between the corresponding patches. In the next section we derive the weight assignment and then we describe how the weights are established using ray casting.

3.2.1 Weight Assignment

We desire that the weight of connection of two patches in the scene reflects the amount of their mutual visibility. Mutual visibility of two patches has been studied intensively in the context of radiosity methods which aim to simulate illumination of diffuse environments. In radiosity methods the visibility between patches drives the amount of power exchanged between them.

Inspired by these methods, we define the weight of an edge connecting two patches (half-space nodes) as the average amount of power transferred between these patches. The power transfer from patch i to patch j is:

$$P_{ij} = A_i B_i F_{ij} \tag{1}$$

where A_i is the area of patch i and F_{ij} is the form factor between patches i and j. Assuming all patches have unit radiosity ($B_i = 1$) we get $P_{ij} = A_i F_{ij}$. The form factor F_{ij} is given by the following equations [7]:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} F(x,y) dA_j dA_i \tag{2}$$

where x and y are points on patches A_i and A_j respectively, dA_i and dA_j are differential surface areas, and $F(x,y)$ is the point-to-point form factor defined as:

$$F(x,y) = \frac{(\Theta_{xy} \odot N_x)(-\Theta_{xy} \odot N_y)}{\pi r_{xy}^2} V(x,y) \tag{3}$$

where Θ_{xy} is a unit direction vector from x to y, $N_x(N_y)$ is the patch normal vector at $x(y)$, r_{xy} is the distance between x and y, and $\odot$ is the inner product. $V(x,y)$ is the visibility function given as:

$$V(x,y) = \begin{cases} 1 & \text{if a ray from point } x \text{ hits point } y \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

From the definition of the form factor, we can see that $A_i F_{ij} = A_j F_{ji}$ and thus $P_{ij} = P_{ji}$.

The form factors can be computed using ray casting as follows: If we originate a ray from a uniformly chosen location on patch i and shoot into a cosine-biased direction (i.e. the probability of the direction depends on the cosine between the direction and the patch normal), the probability of such a ray lands on patch j is equal to the form factor F_{ij} [7]. Thus we can estimate for sufficiently large number of rays: $F_{ij} \approx \widetilde{F_{ij}} = n_{ij}/n_i$ where n_{ij} is the number of rays shot from patch i and landed on patch j and n_i is the total number of rays shot from patch i. From $\widetilde{F_{ij}}$ and $\widetilde{F_{ji}}$ we can compute estimates of power exchange $\widetilde{P_{ij}}$ and $\widetilde{P_{ji}}$. The weight of the link between patches i and j is then defined as the average of the two power exchange estimates:

$$w(i,j) = \frac{1}{2}(\widetilde{P_{ij}} + \widetilde{P_{ji}}) = \frac{1}{2}(A_i \frac{n_{ij}}{n_i} + A_j \frac{n_{ji}}{n_j}) \tag{5}$$

The weight $w(i,j)$ can also be interpreted as the measure of rays which connect the two patches. In order to support this intuition we provide the following substitution: Assuming uniformly distributed rays we know that $n_i \approx n\frac{A_i}{A}, n_j \approx n\frac{A_j}{A}$, where n is the total number of rays cast and A is the total area of patches. By substitution we can rewrite Eq.5 as:

$$w(i,j) \approx \frac{A}{2}\frac{n_{ij}+n_{ji}}{n} \tag{6}$$

Thus the defined weight corresponds to the ratio of the number of rays connecting the two patches with respect to all rays cast with a constant scale factor $\frac{A}{2}$.

Note that the area of a patch is the sum of the areas of all the triangles belonging to the patch. If a half space node represents a single half space point on a triangle the patch corresponds to a voronoi region on the triangle (the details will be discussed in section 3.2.2). Since the samples are mostly uniformly distributed on the triangles, it is reasonable to assume that each such voronoi region has the same area, which is the triangle's area divided by the number of samples on the triangle.

The rays hitting void from a half space node i contribute to the connection between node i and the outside node. The geometrical meaning of the outside node is the bounding sphere of the scene. The weight is defined as:

$$w_o(i) = A_i(1-\sum_j \frac{n_{ij}}{n_i}) \tag{7}$$

Note that this weight is not reciprocal, since we do not shoot rays from the bounding sphere. Based on geometrical probability, the form factor computation stays valid if we shoot rays from the bounding sphere and count the number of hits landed on each node. Nevertheless, it is inefficient and unnecessary. To embody the *inside node* in a geometric sense is problematic. One of the workarounds is that the user assigns a region on the surface or a volume in space as inside. Then we can shoot rays from such regions to build the connection between half space nodes and the inside node. Without users' guidance, we contribute one half space node's outside weight to its opposite half space node's inside weight: $w_i(opp(i)) = w_o(i)$ where opp() is an operator to find out node i's opposite half space node.

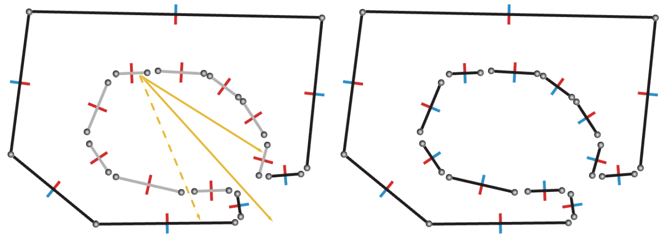


Fig. 7. A 2D illustration of visibility filtering for a cave model. Thicker line segments represent triangle faces in 3D. Thinner blue lines represent outside normals of the faces and red ones represent inside normals. Left: An undesirable classification occurs because rays pass through small cracks between faces. These rays are shown as dotted line. The visibility propagated through the small cracks outweighs the visibility propagated through the main entrance of the cave (shown as straight line). Right: The weights are filtered so that the algorithm gives less weight to smaller cracks resulting in the desired classification.

Due to the imperfection of the geometry, the sampling may incorporate some amount of errors (see Fig. 7). Thus we apply a cos filter to dampen the error fraction. For each node i,

$$w_{max}(i) = max\{w(i,j), w_o(i), w_i(i)\}, \forall j \tag{8}$$

Our directed filter function is defined as:

$$filter(w(i,j)) = \frac{1}{2}(1+\cos(\pi - \pi\frac{w(i,j)}{w_{max}(i)}))w(i,j) \tag{9}$$

3.2.2 Ray Casting

The start locations of the sampling rays are generated uniformly in the region each half space node possesses. Specifically, for a cluster half space node, one triangle of the cluster is selected according to its area and the location is uniformly distributed on the triangle. The directions are sampled on the corresponding hemisphere with a cosine-distributed probability. If a sampling ray shot by node i hits a triangle belonging to a cluster, then a connection is built between the corresponding cluster node and node i.

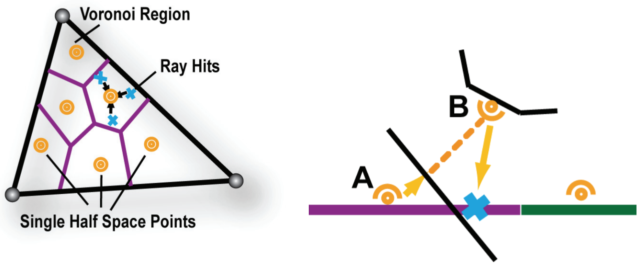


Fig. 8. Left: Voronoi regions formed by single half space nodes in a triangle. Right: A ray hit from Node B is snapped onto Node A, which is not visible from Node B.

If the hit triangle is either an intersected one or a coplanar one, we aim at estimating visibility in such a manner that a single half space point represents a voronoi region on the triangle instead of a single point (see Fig. 8). As we want to avoid complex computations we only try to compute a fast estimate that results in robust visibility classifications. We therefore snap the ray to a half space node j on the triangle closest to the hit point. Since we only originate rays from one location of such node, which is the center of its voronoi region, the form factor integral is biased. Due to the fact that $P_{ij} = A_iF_{ij} = A_jF_{ji} = P_{ji}$, we reduce the variance by using P_{ji} only as the weight between node i and node j instead of $\frac{P_{ij}+P_{ji}}{2}$, if node j is not a single half space point.

Another important special case with snapping occurs when visibility changes between the ray hit and the snapped half space point. In order to discover this case we test reverse visibility by shooting a reverse ray from the node snapped on to the sampling ray's origin. The right of Fig. 8 shows a case where visibility changes in a voronoi region. Node B's sampling ray hits a triangle and the hit is snapped onto Node A, but a reverse ray from Node A to Node B is blocked by other geometry in between. Therefore the ray hit from Node B to A is discarded. Please note that while visibility is symmetric in general, in this case visibility is not symmetric. This is because the starting point of the reverse ray is not the end point of the original ray.

4 CLASSIFICATION

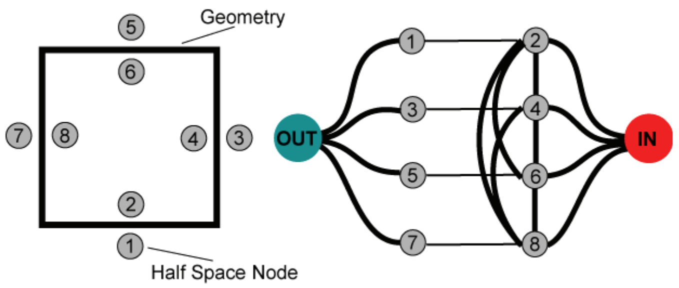


Fig. 9. The visibility graph built from the input geometry on the left.

This section describes how the visibility graph can be processed to obtain an inside and outside classification of each node in the graph. The algorithm is basically a graph clustering or segmentation algorithm. There are several older but powerful algorithms, such as kmeans, single-link, agglomerative, or Ward's algorithm that could be transferred to our problem. See Tan et al.'s book [19] for a review of these techniques. While these algorithms can be fast, they are not guaranteed to produce optimal results. In contrast, many new techniques rely on spectral matrix decompositions.

Our problem is given as follows. The sampling stage generates a visibility graph $G = \langle V, E \rangle$ (see Fig. 9). An inside-outside classification of N half space nodes can be denoted by a binary vector $X = (x_1, x_2, \ldots, x_N)$, where $x_i = 1$ if node i is outside, otherwise $x_i = 0$. We define an energy function for the classification proportional to the visibility connections that have to be cut:

$$E(X) = \sum_{(i,j)\in E} w_{ij}\delta_{ij} \tag{10}$$

where w_{ij} is the weight between node i and j, and

$$\delta_{ij} = \begin{cases} 1 & \text{if } x_i \neq x_j \\ 0 & \text{if } x_i = x_j \end{cases} \tag{11}$$

The minimization of this energy function defines an inside-outside classification with minimal visibility errors. The optimal solution can be computed by the algorithm of Boykov and Jolly [5]. This algorithm was proposed in the context of image processing and it was designed to work with graphs where each node has only a few incident edges. Our experiments show that the algorithm still performs well for graphs with denser (visibility) connections. This is partially due to the fact that we do not shoot a very large numbers of rays to estimate visibility. Fig. 10 shows a visualization of an example graph. In complex situations the algorithm does slow down and can take several minutes. This classification algorithm can be computed once or iteratively depending on the application. In the next section we will show several example applications that we implemented and explain how they setup the classification.

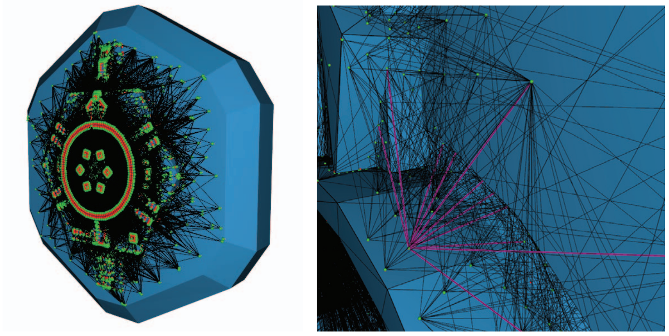


Fig. 10. Left: An example visibility graph. Nodes are shown in green and red and graph-edges in black. Right: All graph-edges connected to a selected node are highlighted in pink.

5 Applications

Inside Removal: Inside removal of models can be computed by one iteration of the sampling and classification step. All triangles that are associated with an outside half space node are considered to be in layer one. All other triangles can be removed. See Fig. 11 and 22 for an example.

Normal Orientation: Normal orientation of a model is computed by one iteration of the sampling and classification step. The user can decide the *front face vertex order*, i.e. if triangles should be oriented in clockwise or counterclockwise order. All triangle faces associated with exactly one outside half-space node, or triangles with multiple agreeing half space nodes are oriented according to the user setting. Triangles associated with multiple half space nodes that indicate a conflicting inside-outside classification are typically duplicated and two triangles with the two possible orientations are created. We found that to be the most intuitive output of normal orientation for general models (see Fig. 13). If the application requires one triangle face to be outside and one triangle be face to be inside, i.e. one normal per triangle, we can pick the orientation where the sum of edge weights to the outside is bigger.

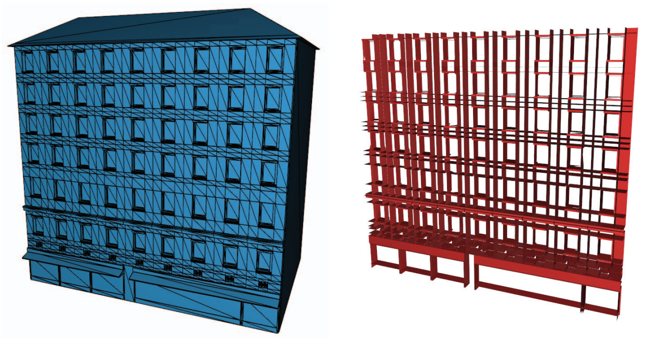


Fig. 11. Left: The outside of the *house* model. The model was created by stacking boxes, so that almost all triangle-edges are non-manifold, and most faces are coplanar. Right: The interior geometry that was removed.

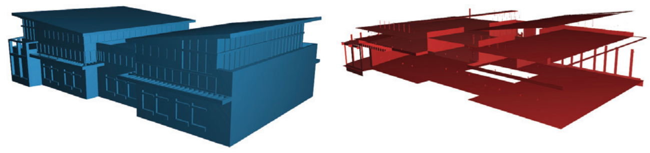


Fig. 12. Left: the outer layer of the *blgd2* model. Right: the inside polygons that were removed.

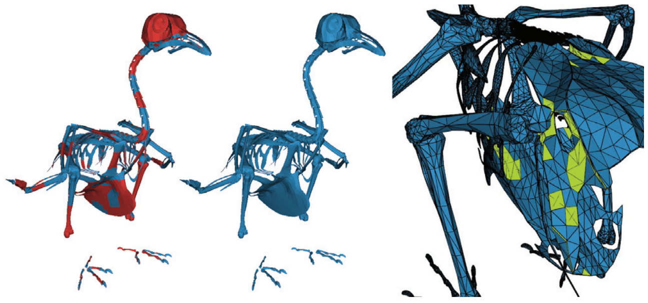


Fig. 13. Left: the *bird* model with front facing (outside) triangles in blue and back facing (inside) triangles in red. Middle: reoriented normals. Right: The model is difficult to process due to the thin (coplanar) structures shown in yellow.

Layer Computation and Visualization: Layer computation can help to gain insight into the internal structure of a model. Layers can be computed by multiple iterations of the classification step and an edge reweighing step. The first iteration computes the first layer. After the first iteration the edges connected to inside nodes associated with triangles in the first layer are connected to the source (the outside). Then the classification step is repeated. The layer computation can be used to assign different transparency values to different layers. See Fig. 14 and Fig. 16 for examples where we render transparency based on an Nvidia whitepaper [8]. While this application has been suggested by previous authors our contribution is the improved definition and computation of inside and outside layers. See Fig. 15 for the separate layers.

Interactive Editing and Mesh Analysis: The framework also includes the ability to interact with the visibility graph. A user can define additional nodes on the model or in free space and manually set weights of edges and classifications of nodes. We do not use interac-

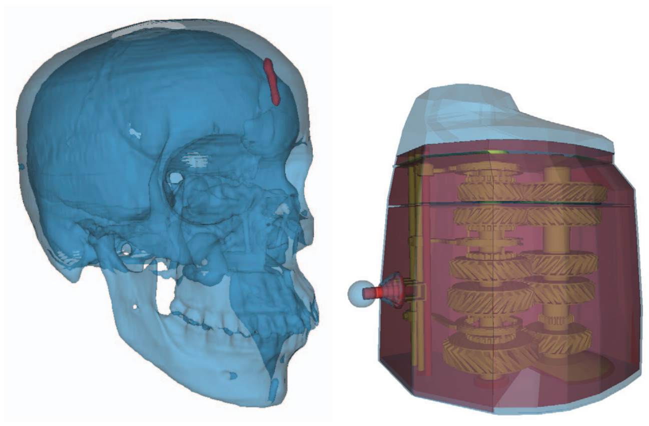


Fig. 14. Left: The *skull* model with the first layer in blue and the second layer in red. Right: layer visualization in the *motor* model.

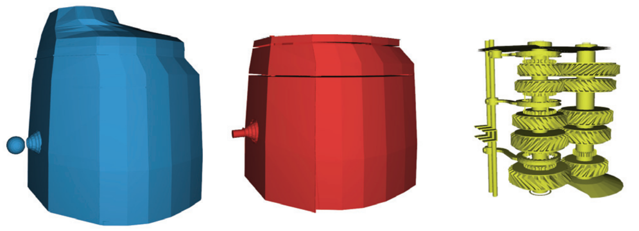


Fig. 15. Three layers of the top of the motor model.

tive editing for the results in this paper, because that would skew the comparison with other approaches.

6 Results

We present the running times of our algorithm on a small set of models. The models were selected to include some variety of how difficult the models are and how many sharp features they possess. For each model we run one inside-outside classification for inside removal. We list the number of polygons of the input model, the number of polygons of the output, as well as the complete running time (See Table 1). We also include an informal comparison with Ju's PolyMender software [10] available at his web page[1]. PolyMender needs to scan the input models into a hierarchical grid. One parameter *maxd* allows the user to control the maximal depth of the octree. In our experiments, we tune the parameters for PolyMender to capture the details of different models. Polymender generates an order of magnitude more triangles for the models house, building 2, and mechanic 1. A main reason is that coplanar faces seem to force a very detailed voxelization. Most of the models are well reconstructed with $maxd = 9$. The bird skeleton contains many detailed structures and requires $maxd = 10$. For simpler models such as the Turbine and Skull we set $maxd = 8$. We use the version *dc-clean* for all models except for the bird which uses *dc*. While our algorithm cannot compete with Polymender's hole filling functionality, we believe that the results underline that our proposed system can complement volumetric methods well. We also list a breakdown of our running times for the three major steps clustering, sampling, and graph cut in Table 2. In the same table we list statistics about the graph and memory consumption. The main parameters of our algorithm are the number of random samples, the number of border samples, the number of rays per node, the threshold for the intersection computation and the threshold for the coplanarity computation. For these tests we made a binary correct or not correct decision based on visual inspection. See Table 3 and Table 4 for the results. We measured the influence of clustering on three models (see Table 5). Even though we do not perform clustering in the second column, we still have to run the intersection and coplanarity tests (I[s]). Note that the skull is an extreme case that benefits a lot from clustering, as it has very large connected components; the others are just normal cases where both connected mesh and intersected triangles both exist. The third column simulates a triangle soup case by randomly messing up selected triangles. The clustering algorithm typically improves the overall running time because it reduces the number of nodes and edges in the visibility graph processed by the graph cut algorithm.

[1]http://www.cse.wustl.edu/ taoju/code/polymender.htm

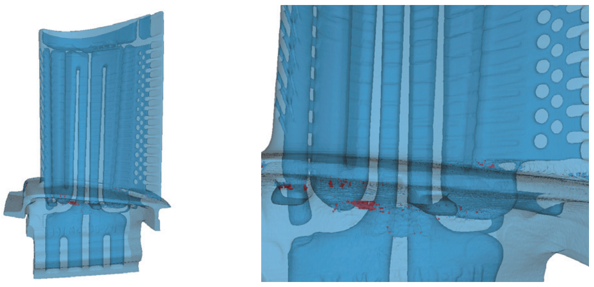


Fig. 16. Left: The *turb* model with the first layer in blue and second layer in red. This model is an illustration of the difference of the *view-based* and *object-based* inside-outside classification. Please note that the interior pipe structure is correctly classified as outside because it is reachable through several smaller pipes. Right: a closeup.

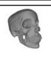
Model		Alg.	Tri.In#	Tri.Out#	Time [s]
house		VGC	24K	11K	89.1s
		JU	24K	1.9M	14.97s
bird		VGC	111K	92K	17.6s
		JU	111K	1.1M	8.37s
bldg1		VGC	93K	85K	43.9s
		JU	93K	791K	17.812s
bldg2		VGC	33K	22K	22.7s
		JU	33K	897K	6.687s
mech1		VGC	1.04K	1.03K	5.52s
		JU	1.04K	861K	3.062s
mech2		VGC	25K	23K	3.75s
		JU	25K	742K	2.34s
turb		VGC	1.76M	1.76M	66.5s
		JU	1.76M	843K	46.18s
skull		VGC	1.16M	1.16M	47.89s
		JU	1.16M	1.61M	44.31s
motor		VGS	140k	44k	74.14s
		JU	140k	1.5M	17.016s

Table 1. The table lists the number of input triangles (Tri.In) and output triangles (Tri.Out) and total computation time in seconds. We compare our results (VGC: Visibility Graph Cut) against Ju's PolyMender (version 1.7).

7 Discussion

In this section we want to compare to previous work, identify contributions and open problems that are of interest for future research.

Mesh Repair: Our algorithm performs several steps of a mesh repair framework, but we do not currently address hole filling, a major challenge that is implemented in some previous mesh repair systems, e.g. [2, 10]. However, the problem of hole filling is not solved by previous work and it remains an inherently difficult topic. Many cases require user input to be resolved. Our major avenue for future work is to determine how the visibility graph can be used to let a user specify hints for hole filling.

Inside Outside Classification: We believe that our algorithm significantly improves the state of the art, because we make better use of visibility information. Previous work, especially Borodin et al. [4]

Model	C[s]	S[s]	G[s]	Rays	Node	Edge	Mem
house	2.1	73	14	20M	1.2M	8.7M	316M
bird	7.3	9.1	1.2	3.5M	224K	1.2M	46M
bldg1	6.4	36	1.5	11M	691K	5.3M	183M
bldg2	4.8	16.7	1.2	6.9M	432K	1.9M	74M
mech1	0.1	5.3	0.1	2.4M	77K	825K	28M
mech2	0.7	2.1	0.95	1.2M	80K	156K	7M
turb	66	0.1	1ms	9.4K	590	739	40K
skull	46	1.4	0.06	55K	3.4K	1.5K	338K
motor	5.1	58	11	26M	1.6M	12.4M	443M

Table 2. The break-down of computation times for each model and running times in seconds for clustering (C) , sampling (S), and graph cut (G). Further we list the number of rays used in the sampling stage (Rays), the number of nodes (Node) and edges (Edge) of the visibility graph, and the memory consumption (Mem).

RS = 5, BS = 30, RperN =	8	16	32	48	64
house, time=	49s	89s	206s	327s	382s
RS = 5, RperN = 16, BS =	6	15	24	30	36
house, time=	23s	44s	68s	89s	121s
BS = 30, RperN = 16, RS =	0	5	10	15	20
house, time=	75s	89s	102s	121s	143s
RS = 10, BS = 0, RperN =	8	16	32	48	64
bird, time=	7s	17s	22s	32s	42s
RS = 10, RperN = 16, BS =	0	6	15	24	30
bird, time=	17s	35s	59s	108s	146s
BS = 0, RperN = 16, RS =	5	10	15	20	25
bird, time=	8s	17s	33s	52s	79s

Table 3. We evaluate the parameters number of random samples (RS), number of border samples (BS), and the number of rays per node (RperN) on two selected models. We always keep two parameters the same and vary the other one. As result we report the running time. The lowest running time that produces a correct result is highlighted in red.

and Murali et al. [15] make many assumptions about the model and are therefore not robust to intersecting and coplanar triangles. On the other hand volumetric methods [2, 10] can deal with a larger number of inputs, but they are not able to classify the original geometry. As a result a significant increase in triangles is likely for all models that do not have a nice uniform triangulation. Furthermore, our experience with the Polymender software [10] shows that there are some robustness issues that would have to be resolved. We therefore argue that our algorithm is the best available choice for inside-outside classification and related applications.

8 Conclusion

We presented a robust visibility based geometry analysis algorithm that takes a triangular model as input and computes an inside and outside classification of oriented triangle faces. We show how to use this classification for three example applications: inside removal, normal orientation, layer-based visualization. The core idea of out framework is to propagate visibility using ray casting and to compute a classification using graph cut. We believe that our algorithm is a significant technical improvement over previous techniques.

Model	x-size	y-size	z-size	I-thresh	C-thresh
house	204	216	157	$10^{-6} - 10^{-2}$	$10^{-5} - 10^{-2}$
mech1	26	25	30	$10^{-6} - 10^{-3}$	$10^{-5} - 10^{-2}$
motor	5	21	20	$10^{-6} - 10^{-2}$	$10^{-5} - 10^{-2}$

Table 4. We list the setting for three models for the threshold parameters that produce correct results: threshold for the intersection detection (I-thresh) and threshold for the coplanar detection (C-thresh). The parameter range was determined by running a large number of tests with different thresholds and subsequent visual inspection of the results.

	Clustering (C[s]/S[s]/G[s])	No Clustering (I[s]/S[s]/G[s])	Triangle Soup (S[s]/G[s])
bird	18 (7.3/9.1/1.2)	25 (3.5/18.7/3)	11 (10.1/1.1)
skull	47 (46/1.4/0.06)	194 (21.4/145/28)	161 (135.2/25.9)
motor	25 (3.5/20/1.5)	48 (2.3/43.2/2.5)	11 (10.1/0.5)

Table 5. Running times for clustering(C), sampling(S), intersection(I), and graph cut(G)

Acknowledgements

The authors acknowledge the contribution of all the reviewers and financial support from NSF IIS-0612269, NSF CCF-0643822, NSF CCF-0546881, and grant LC-06008 from the Czech Ministry of Education, Youth and Sports. We thank Will Schroeder, Ken Martin, Bill Lorensen, Bruce Teeter, Terry Yoo, Mark Levoy and the Stanford Graphics Group for the 3D models in this paper.

References

[1] G. Barequet and M. Sharir. Filling gaps in the boundary of a polyhedron. *Computer Aided Geometric Design*, 12(2):207–229, 1995.

[2] S. Bischoff, D. Pavic, and L. Kobbelt. Automatic restoration of polygon models. *ACM Transactions on Graphics*, 24(4):1332–1352, Oct. 2005.

[3] J. H. Bøhn and M. J. Wozny. A topology-based approach for sell-closure. In *Geometric Modeling for Product Realization*, volume B-8, pages 297–319, 1992.

[4] P. Borodin, G. Zachmann, and R. Klein. Consistent normal orientation for polygonal meshes. In *Computer Graphics International*, pages 18–25. IEEE Computer Society, 2004.

[5] Y. Boykov and M.-P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images. In *ICCV*, pages 105–112, 2001.

[6] CGAL. Cgal: Computational geometry algorithms library. www.cgal.org, 2008.

[7] P. Dutre, K. Bala, and P. Bekaert. *Advanced Global Illumination*. AK Peters, 2006.

[8] C. Everitt. Interactive order-independent transparency, 2001.

[9] A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Trans. Vis. Comput. Graph*, 7(2):136–151, 2001.

[10] T. Ju. Robust repair of polygonal models. *ACM Transactions on Graphics*, 23(3):888–895, Aug. 2004.

[11] T. Ju, F. Losasso, S. Schaefer, and J. D. Warren. Dual contouring of hermite data. *ACM Trans. Graph*, 21(3):339–346, 2002.

[12] P. Liepa. Filling holes in meshes. In *Symposium on Geometry Processing*, volume 43, pages 200–206, 2003.

[13] T. Möler. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[14] T. Möller and E. Haines. *Real-Time Rendering, Second Edition*. A. K. Peters Limited, 2002. ISBN 1568811829.

[15] T. M. Murali and T. A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *1997 Symposium on Interactive 3D Graphics*, pages 155–162. ACM SIGGRAPH, 1997.

[16] F. S. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, Apr./June 2003.

[17] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. on Graphics*, 24(3):1176–1185, 2005.

[18] Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, WACG*, 1996.

[19] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addiso Wesley, 2006.

[20] P. Wonka, M. Wimmer, K. Zhou, S. Maierhofer, G. Hesina, and A. Reshetov. Guided visibility sampling. *ACM Trans. Graph.*, 25(3):494–502, 2006.

[21] E. Zhang and G. Turk. Visibility-guided simplification. In *IEEE Visualization*, 2002.