Automatic Restoration of Polygon Models

STEPHAN BISCHOFF, DARKO PAVIC and LEIF KOBBELT
RWTH Aachen University, Computer Graphics Group

We present a fully automatic technique which converts an inconsistent input mesh into an output mesh that is *guaranteed* to be a clean and consistent mesh representing the closed manifold surface of a solid object. The algorithm removes all typical mesh artifacts such as degenerate triangles, incompatible face orientation, non-manifold vertices and edges, overlapping and penetrating polygons, internal redundant geometry as well as gaps and holes up to a user-defined maximum size ρ. Moreover, the output mesh always stays within a prescribed tolerance ε to the input mesh. Due to the effective use of a hierarchical octree data structure, the algorithm achieves high voxel resolution (up to 4096^3 on a 2GB PC) and processing times of just a few minutes for moderately complex objects. We demonstrate our technique on various architectural CAD models to show its robustness and reliability.

Categories and Subject Descriptors: I.3.5 [**Computational Geometry and Object Modeling**]: Geometric algorithms, languages, and systems

General Terms: Algorithms

Additional Key Words and Phrases: mesh repair, polygon meshes, surface extraction, voxelization

1. INTRODUCTION

In computer-aided industrial development and design processes the concurrency and dependency of the various project stages is increasing the requirements for flexible exchange of geometry data. Sophisticated sub-tasks like numerical fluid-, structure-, or crash-simulation, scientific visualization, interactive shape design, and rapid prototyping all should ideally operate on a common 3D model during the process in order to enable a proper product data management. This is why polygonal mesh models (e.g., in the form of STL-files) have established as a universal language to communicate geometric information between different software systems in many computer graphics application areas ranging from mechatronics and architecture to automotive-, ship-, and airplane design. Moreover, polygon meshes are quite easy to generate since all we need is a set of sample points on the object's surface that have to be connected in order to define a piecewise linear approximation of the underlying surface.

On the other hand, since different applications impose quite different requirements on the consistency and quality of the geometry data, many compatibility problems occur in practice. For example, the tessellation backends of most CAD systems produce polygon meshes that are sufficient for mere visualization but often the resulting surfaces are not suitable for further processing since they may have small holes, overlapping faces, degenerate triangles, or topological inconsistencies.

This leads to application scenarios where more time is spent to convert and repair geometry data between the different phases of the processing pipeline than to perform the actual computations on it. Even worse, due to the various kinds of errors and inconsistencies, it is considered very difficult to repair polygon meshes in a fully automatic manner. Often the user has to support the procedure interactively which significantly adds to the project costs.

In this paper we present a fully automatic technique for the restoration of polygon meshes. Our input consists of a possibly inconsistent triangulated polygon mesh, a tolerance value ε, and a threshold ρ for the maximum size of holes to be closed. Our technique generates a new triangle mesh that approximates the original mesh as good as possible but at least up to the prescribed tolerance ε and that has all inconsistencies smaller than ε fixed. Moreover, if the original data has gaps or holes that are larger than ε but smaller than ρ, we fix these automatically by locally filling in surface patches. The output is guaranteed to be a clean and consistent mesh that represents the closed manifold surface of a solid, i.e., at every edge exactly two triangles meet and the fan of triangles around each vertex is topologically equivalent to a disk. Such models can be used directly for down-stream applications like numerical simulation or rapid prototyping and standard mesh processing algorithms like remeshing and mesh decimation can be performed on such meshes without problems.

Our automatic technique is able to remove all typical mesh inconsistencies such as degenerate triangles, incompatible face orientation, non-manifold vertices and edges, overlapping and penetrating geometry, narrow gaps, small holes and internal redundant geometry (like double walls). At the same time, even though the algorithm resamples the original model, all important geometric features like sharp corners and edges are well preserved. Our technique is sufficiently fast to process even complex input models with high precision in just a few minutes on a standard PC.

Although we do not make any specific assumptions about the type of input geometry, we are focusing on technical CAD datasets. In principle, densely sampled meshes as they are generated by 3D scanners could also be processed by our algorithm. However, for this kind of input data the precision requirements, especially for sharp feature preservation, are usually not as high since the input data is noisy anyway. Also face orientation and redundant internal geometry are minor problems when dealing with densely scanned data.

2. PROBLEM DEFINITION

In this section we give an intuitive and abstract explanation of the basic ideas and concepts of our algorithm. These concepts are best described in a continuous setting which doesn't rely on implementation details. However, this continuous setting cannot be implemented robustly by a discrete data structure. Hence, in the subsequent sections we show how to approximate and adapt this continuous setting by a discrete voxel grid. Of course, we will then have to adapt our algorithm accordingly, but its basic concepts stay the same.

Let $\{T_i\}$ be a set of triangles that inconsistently describes the geometric shape of a solid object S. By *inconsistent* we mean that the orientation of each triangular face can be arbitrary and that the edges of neighboring triangles do not have to match perfectly, i.e., we can have gaps and holes in the surface as well as self-penetrating geometry. Moreover, complex vertices and edges can occur where the surface locally fails to be homeomorphic to a (half-) disk. We are interested in constructing a clean and consistent triangle mesh which approximates the boundary of the solid S. Obviously, due to the inconsistencies of the mesh $\{T_i\}$ the solid S might not be uniquely defined. Hence our goal is to reconstruct at least a solid S' which tightly fits to the input triangles within the prescribed tolerance ε. This means that for every point on any relevant triangle T_i the surface of the solid S' is not further away than ε. Here, the relevant triangles are those that contribute to the outside surface of S.

The complete surface restoration problem falls into two sub-tasks. One is to determine the *topology* of the resulting surface and the other is to properly sample the surface in order to faithfully reconstruct its *geometric shape*.

Let $\{E_i\}$ be the set of boundary edges in the triangle soup $\{T_i\}$, i.e., those edges that belong to just one triangle. Similarly let $\{V_i\}$ be the set of boundary vertices whose adjacent triangles do not form at least one cyclic fan around it. Notice that according to our definition, complex edges with more than two adjacent triangles and complex vertices with more than one cyclic fan are not considered as belonging to the boundary.

Now assume we replace each triangle with a flat triangular prism of infinitesimal height ε, then the boundary of the resulting solid S_ε trivially is a manifold surface (possibly consisting of several components).

If we further replace each boundary edge E_i by a cylinder of radius ρ and each boundary vertex V_i by a sphere of radius ρ we still have a well-defined solid $S_{\varepsilon,\rho}$ with manifold boundary[1]. As ρ increases the different components of the solid S_ε are progressively merging. Since the input triangles are supposed to provide a reasonable approximation of the surface of some solid object S, there is a value ρ for which the solid $S_{\varepsilon,\rho}$ divides the embedding space into exactly one outside component C and one or several inside components. The interface Q between this solid $S_{\varepsilon,\rho}$ and the outside component C defines the *topology* of the restored surface R that we want to construct. This definition implies that we automatically remove any internal geometry (see Fig. 1). Note that we cannot derive ρ from the input geometry, as we do not know the underlying solid S. Hence, in practice, ρ has to be specified by the user and the algorithm will then close all gaps of diameter $\leq 2\rho$.

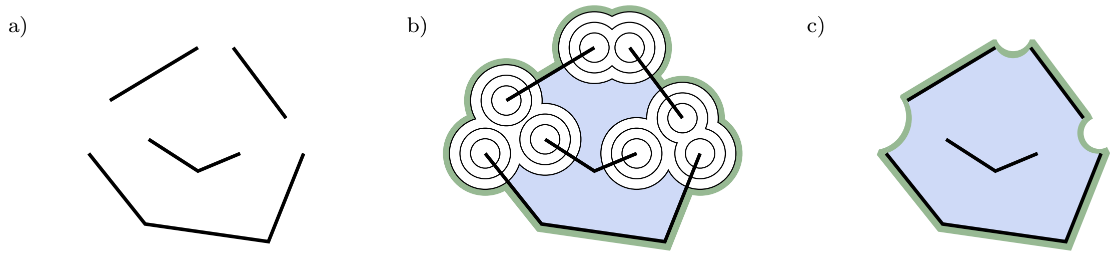


Fig. 1. Starting from an inconsistent polygon model (a) we let cylinders with radius ρ grow at all boundary edges. At some point, the resulting solid separates the outside component from the interior components and we find the interface (green) that defines the restored surface's *topology* (b). The *geometry* of the restored surface finally matches the original data wherever it is available. The holes in the original data are eventually closed by smooth patches (c).

The *geometric shape* of R is reconstructed by re-sampling the input triangles $\{T_i\}$. To avoid sampling artifacts we have to make sure that sharp features defined by adjacent triangles are properly sampled. For those regions of R which correspond to the cylindrical boundary parts of Q there is no underlying input geometry available. Hence, we fill in smooth membrane patches that interpolate the original boundary edges.

3. OUR APPROACH

We use a combination of a volumetric geometry representation and the original triangle data in order to exploit the advantages of both. The collection of input triangles $\{T_i\}$ provides the best available *geometric* information and is necessary for the reliable detection and faithful reconstruction of sharp features. The volumetric representation is conceptually based on a voxel grid (in fact, we use an adaptive octree data structure) which makes it easy and safe to detect and resolve *topological* inconsistencies since we can rely on the simple voxel set topology. In fact, the theory of digital topology [Kong and Rosenfeld 1989] states that we can consistently represent a manifold surface by a set of voxels if we consider "full" voxels as adjacent if they are face-neighbors (6-neighborhood) and "empty" voxels as adjacent if they are vertex-neighbors (26-neighborhood).

Hence our approach is to determine the mesh connectivity (= topology) of the resulting surface exclusively based on the voxel topology and to define the vertex positions of the output mesh (= geometry) exclusively

[1]If the boundary happens to be non-manifold because of a configuration with two cylinders or spheres exactly touching each other, we can always infinitesimally adjust the radius ρ to obtain a manifold configuration. Since our implementation is based on a finite voxelization, we will apply well-known results from digital topology to avoid such non-manifold configurations.

based on the input triangles. This extremely simplifies the handling of topological degeneracies in the input and it guarantees that the output faithfully approximates the input.

Our specific contributions in this paper are:

—A fast and reliable sparse adaptive voxelization technique which comes with the guarantee that the collection of “full” cells exactly represents the topology of the input solid S_ε.

—A set of space and time efficient voxel operations that exploit the hierarchical structure of the octree. These operations include triangle voxelization, volumetric seed filling as well as morphological dilation and erosion.

—A feature sensitive surface sampling technique that computes optimal sample positions in the *interior* of those voxels that are intersected by the surface.

—An extension of the dual contouring algorithm [Ju et al. 2002] that, unlike the original, produces guaranteed manifold meshes.

Putting all these ingredients together, we present an algorithm that restores meshes completely automatically. The technique is 100% robust against any type of inconsistency in the input triangle soup since it does not rely on inherently error-prone computations such as ray intersection test with an inconsistent mesh. Through the effective use of hierarchical data structures, we are able to run voxel resolutions as high as 4096^3 for moderately complex models on a PC with 2 GB main memory.

4. RELATED WORK

In the computer graphics literature one can distinguish between two fundamentally different approaches to surface restoration. One approach is surface based and makes the implicit assumption that artifacts and inconsistencies affect only a small fraction of the object. One tries to identify consistent sub-meshes which are then merged together by snapping corresponding boundary segments or by stitching small patches into the remaining gaps and holes [Bøhn and Wozny 1992; Dolenc and Mäkelä 1991; Borodin et al. 2002; Barequet and Sharir 1995; Barequet et al. 1998; Barequet and Kumar 1997; Guéziec et al. 2001; Turk and Levoy 1994; Liepa 2003]. Other local inconsistencies can be removed by first cutting the mesh and then stitching, e.g., complex vertices and edges [Guéziec et al. 2001] or small handles and tunnels [Guskov and Wood 2001].

The major difficulties with this approach arise from numerical robustness issues and from the fact that spatial proximity not always coincides with geodesic proximity [Weihe and Willhalm 1998]. As a consequence, artifacts like overlapping geometry and “double walls” are difficult to handle. Moreover, some surface artifacts might not even be detectable in the connectivity of the input mesh, e.g., two topologically consistent meshes spatially penetrating each other.

The other approach is volume oriented. Here the idea is to convert the given mesh data into a volumetric representation, e.g., a signed distance function [Frisken et al. 2000], and then generate a consistent mesh when converting back to a surface [Lorensen and Cline 1987; Gibson 1998]. In the volumetric setting, various filter operations can be applied to fix some of the topological artifacts and holes [Davis et al. 2002] but this also removes sharp features from the input data. Hence, most volumetric approaches have been suggested in the context of topology-modifying surface simplification [Andújar et al. 2002; Nooruddin and Turk 2003]. In contrast, we are not aiming at simplification but rather at preserving as much geometric detail as possible. Nevertheless we could use our algorithm for simplification by choosing a large tolerance threshold ε.

In the volumetric approach the conversion from surfaces to signed distance functions can be tricky in the presence of penetrating geometry and cracks [Nooruddin and Turk 2003]. Moreover, when converting back from volumes to surfaces, alias errors can compromise the output quality unless feature sensitive sampling techniques are employed [Kobbelt et al. 2001; Ju et al. 2002].

Recently [Ju 2004] presented a hybrid approach that resamples the input geometry on the edges of a regular grid and uses approximate discrete minimal surfaces to fill the holes. However, as the hole boundaries are explicitly traced, interpenetrating non-manifold geometry, like dangling triangles, can lead to unexpected and wrong fillings. Furthermore, the approach does not adapt to the surface shape and hence produces overly fine tessellated output meshes even in flat regions of the surface.

The technique presented in this paper combines the surface and the volume approach in a way that exploits the advantages of both. We use the volumetric approach to restore a proper surface topology but instead of trying to compute the characteristic function of the unknown solid S inconsistently defined by the input data $\{T_i\}$ [Andújar et al. 2002; Nooruddin and Turk 2003], we compute the characteristic function of the known solid S_ε (see Sect. 2) which is much more reliable. Similar to [Brunet and Navazo 1990; Greß and Klein 2003] we use a hierarchical representation which associates polygonal geometry with every octree cell.

5. DETAILS OF THE ALGORITHM

Our algorithm proceeds in six steps. First an adaptive octree is generated where each cell stores references to the triangles that intersect with it. In the second step, this octree representation is adaptively refined further to increase the resolution in regions of high geometric complexity and in the vicinity of boundary edges. The third step applies a sequence of morphological operations to the cells of the octree to determine the topology of the restored surface. This topological information allows us to compute the connectivity of a triangle mesh in the fourth step by using an extension of the dual contouring algorithm [Ju et al. 2002] which guarantees that the restored surface has a proper manifold topology. Next, in step five, we compute the vertex positions for the output mesh. This is done by feature sensitive sampling of the input geometry. The final step six performs some simple and local mesh optimization operations on the output mesh to determine the shape of the patches that cover the holes in the input mesh.

The input to our algorithm consists of an unstructured set of triangles $\{T_i\}$, a tolerance value ε and a value ρ which controls the maximum size of holes in the surface that should be fixed.

5.1 Voxelization

We assume that the input model is centered at the origin and that

$$M \;:=\; \max\{|x_i|, |y_i|, |z_i|\}$$

is its maximum absolute coordinate value. We initialize the root cell of an octree [Samet and Webber 1988] with a bounding cube that has its corners at $(M + s\,\varepsilon)[\pm 1, \pm 1, \pm 1]^T$. By setting the maximum refinement level of the octree to k with

$$k - 1 \;<\; \log_2(\,M/\varepsilon + s\,) + 1 \;\leq k$$

we guarantee that the size of the smallest leaf cells (*voxels*) is below the prescribed tolerance ε and that we have at least s layers of empty voxels along the faces of the root cell. This simplifies the implementation of voxel operations since no special cases where a relevant part of the geometry intersects boundary cells of the voxel grid, have to be considered.

For every input triangle T_i we recursively traverse the octree and store a reference index i in each leaf cell that is intersected by T_i. A cell is split as soon as two non-coplanar triangles are registered where we define the supporting planes of two triangles to be numerically coplanar if they deviate by less than ε within the current cell.

Let $E_1 = [\mathbf{n}_1, d_1]$ and $E_2 = [\mathbf{n}_2, d_2]$ be the normalized equations of the supporting planes of two triangles T_1 and T_2 that intersect the same octree cell with edge length $2\,h$ centered at the origin. If $\mathbf{n}_1$ and $\mathbf{n}_2$ are parallel, the deviation of E_1 and E_2 is simply $|d_1 - d_2|$. If $\mathbf{n}_1$ and $\mathbf{n}_2$ are not parallel, we conservatively estimate the deviation of the two planes E_1 and E_2 within a sphere S of radius $r = \sqrt{3}\,h$ centered at the cell center $\mathbf{c}$. Elementary geometry tells us that the maximum deviation occurs in the plane E^* spanned by $\mathbf{n}_1$ and $\mathbf{n}_2$ and which passes through the point $\mathbf{p}$ that is the point on the intersection line of E_1 and E_2 lying closest to the cell center. The point $\mathbf{p}$ is computed by solving the underdetermined system

$$\begin{pmatrix} \mathbf{n}_1^T \\ \mathbf{n}_2^T \end{pmatrix} \mathbf{p} = \begin{pmatrix} -d_1 \\ -d_2 \end{pmatrix}$$

in the least norm sense. Now let

$$\mathbf{b}_1 = \frac{\mathbf{n}_1 + \mathbf{n}_2}{||\mathbf{n}_1 + \mathbf{n}_2||}, \quad \mathbf{b}_2 = \frac{\mathbf{b}_1 \times (\mathbf{n}_1 \times \mathbf{n}_2)}{||\mathbf{b}_1 \times (\mathbf{n}_1 \times \mathbf{n}_2)||}$$

be the two angle bisectors of E_1 and E_2 and let

$$\begin{aligned} \{\mathbf{a}_1, \mathbf{a}_2\} &= E_1 \cap E^* \cap S \\ \{\mathbf{a}_3, \mathbf{a}_4\} &= E_2 \cap E^* \cap S \end{aligned}$$

which can be computed by simple ray-sphere intersection tests. We then estimate the deviation δ of E_1 and E_2 by projecting the extremal points $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4$ onto the bisectors $\mathbf{b}_1, \mathbf{b}_2$, i.e.

$$\delta = \min(\delta_1, \delta_2)$$

where

$$\delta_i = \max_{j=1,2,3,4} \mathbf{b}_i^T \mathbf{a}_j - \min_{j=1,2,3,4} \mathbf{b}_i^T \mathbf{a}_j$$

When $\delta > \varepsilon$ we split the current cell and assign the registered triangles to the corresponding sub-cells.

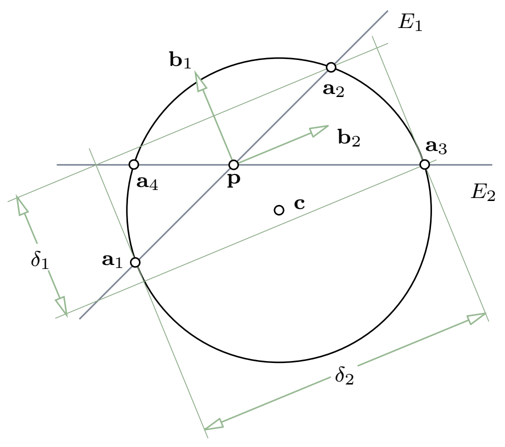


Fig. 2. Subdivision criterion in the plane E^* to compute the maximum deviation of two planes E_1 and E_2.

For each triangle T_i the octree traversal is controlled by a quick triangle-cell intersection test which uses the separating axis theorem (SAT) for convex polytopes [Gottschalk et al. 1996]. The idea is to project both objects onto an axis and check if the resulting intervals are disjoint. SAT says that we only have to check a finite number of axes. Let D be the set of all edges from both objects then we have to check every axis that is defined by the cross product of any two vectors from D. In the triangle-cell case this means we have to check at most 13 different directions.

The traversal stops if the finest level k has been reached. After the first phase we have an adaptively refined octree with each leaf cell referring to a set of triangles intersecting with it. Our cell splitting criterion guarantees that all triangles in a cell that is not from the finest level k, are coplanar up to the prescribed error tolerance ε.

5.2 Octree disambiguation

Since we want to use the digital voxel topology, i.e., face-adjacency or 6-neighborhood [Bischoff and Kobbelt 2002] to determine the surface topology of the output mesh, we have to make sure that both are equivalent where the input data is (locally) manifold. This implies that we have to refine our octree representation such that the given surface actually passes through all faces between adjacent "full" cells (see Fig. 3). By this we automatically guarantee that non-adjacent geometric features of the input mesh are sufficiently separated by at least one layer of empty cells. In order to achieve this we make another pass over the octree, this time splitting each cell that has another non-coplanar triangle in any of its 26-neighbors (i.e., face-, edge-, or vertex-adjacency). Moreover, if the triangles in two face-neighboring cells are numerically coplanar, we still have to check for each triangle whether its supporting plane actually intersects the common face between the two cells in order to guarantee that the resulting voxel topology correctly reflects the surface topology of the input mesh. This test can be done very fast by just checking if the four corners of this common face are lying on the same side of the supporting plane.

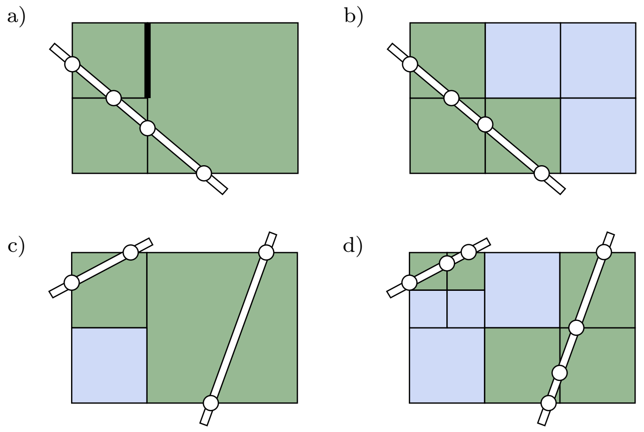


Fig. 3. In order to have the input surface topology correctly represented by the collection of full octree cells, we have to refine the octree in a number of ambiguous configurations. In (a) the surface does not pass through the octree facet between two full cells (fat facet). Further refinement in (b) established this property. In (c) two components of the input geometry are not sufficiently separated. Again, the configuration can be resolved by further octree refinement (d).

In the same pass we also refine every cell that contains a boundary edge such that eventually all boundary edges of the input data are represented on the finest level k. We tag the corresponding voxels as "boundary". This status information will be used in the next step to close the gaps and holes in the input surface.

The robust detection of boundary edges requires some effort since we do not assume any reliable connectivity information in the input data. We solve this problem by exploiting the graphics hardware. Let us assume that we have a cell from octree level $k' \leq k$ and that all triangles in this cell are numerically coplanar. Our idea is to render these triangles and check for pixels in the frame buffer that remain in background color.

According to the definition of the root cell in the octree, we find that the size of the level k' cell is not larger than $2^{(k-k')}\,\varepsilon$. Hence, we use a frame buffer with $2^{(k-k'+1)} \times 2^{(k-k'+1)}$ pixel resolution. We set the viewing frustum to the cell geometry (i.e., parallel projection) and we define the viewing direction by the maximum component of the normal vector to the supporting plane of the triangles.

Into the cleared frame buffer we first render the supporting plane in green with z-buffer and frustum clipping enabled. Then we render all the triangles in blue with a z-offset of ε. We know that the cell contains a boundary edge if the resulting frame buffer has at least one green pixel left (see Fig. 4).

As our rendering test only evaluates the cell geometry at a finite number of pixels we may fail to classify a cell as "boundary" even if it contains boundary edges. However, this can only happen if the boundary edges are closer than $\varepsilon \leq \rho$. Because of the volumetric representation, such a cell will nonetheless be tagged as "full" and the gaps will be "invisible" to the dual contouring algorithm which we use to reconstruct the geometry.

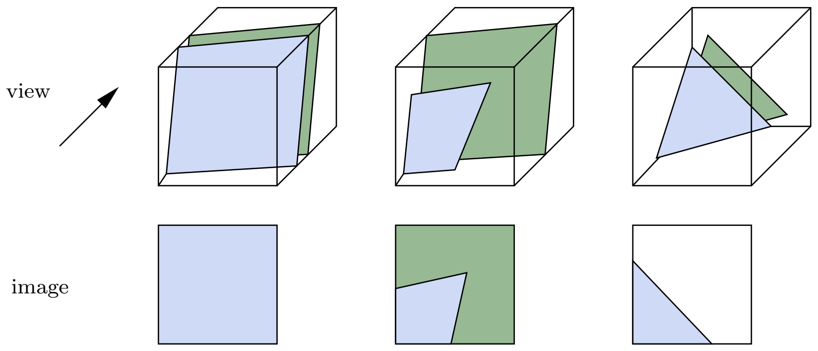


Fig. 4. Boundary detection by two-pass rendering of all triangles in a cell.

5.3 Topological hole fixing by morphological operations

After the second phase we have an adaptively refined octree with full and empty cells from all levels. The "full" voxel's face neighborhood relation properly represents the topology of the input mesh. All geometric features are separated by empty cells and some of the finest level cells are tagged as "boundary".

In order to close the gaps and holes and to determine the restored surface topology, we apply a sequence of morphological operators. The goal is to implement a discrete version of the cylinder growing process described in Sect. 2. Since the user-defined maximum radius of these cylinders is ρ, we simulate the cylinder growing by performing $s = \lceil \rho/\varepsilon \rceil$ elementary dilation steps on all "boundary" voxels. Some more voxel operations are then necessary to actually determine the outside component and to shrink back the "dilated" voxels to the original "full" voxels.

Morphological operations with a small and symmetric template can be understood as procedures that exchange information between adjacent voxels [Gonzalez and Woods 1992]. For example, an elementary dilation operation sets all voxels to active that have an active neighbor. Conversely, we can think of each active voxel passing its active status on to neighboring voxels. Hence we can decompose the elementary dilation into sub-operations, i.e., first pass the status information to the left neighbor, then to the right, top, bottom, front, back and so forth.

Each of these sub-elementary operations can be implemented very efficiently by a simple recursive traversal of the octree. The idea is to call a procedure for each pair of source and target voxel that are supposed to exchange information. As an example, we present the `pass_back_to_front()` procedures (cf. Fig. 5 for the indexing scheme). Passing status information to the other directions is implemented analogously.

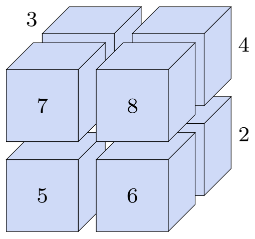


Fig. 5. Indexing scheme for the children of an octree cell.

```
pass_back_to_front_1(cell c) {
  for each back sub-cells c[1] ... c[4]
     pass_back_to_front_1(c[i])
  for each neighbor pair (c[1],c[5]), ... (c[4],c[8])
     pass_back_to_front_2(c[i],c[i+4])
  for each front sub-cells c[5] ... c[8]
     pass_back_to_front_1(c[i])
}

pass_back_to_front_2(cell c1, cell c2) {
  if both cells are leaves
     c2.next_status = c1.current_status
  else if c1 is a leaf cell
     for each back sub-cell c2[1] ... c2[4]
        pass_back_to_front_2(c1,c2[i])
  else if c2 is a leaf cell
     for each front sub-cell c1[5] ... c1[8]
        pass_back_to_front_2(c1[i],c2)
  else for each neighbor pair (c1[5],c2[1])...(c1[8],c2[4])
     pass_back_to_front_2(c1[i+4],c2[i])
}
```

Once the status information has been passed into all directions, we call the procedure

```
commit(cell c) {
   if c is a leaf
      c.current_status = c.next_status
   else
      for each sub-cell c[1] ... c[8]
         commit(c[i])
}
```

to finalize the elementary dilation operation. Notice that this operation can be applied to an adaptively refined octree and the status information is passed from each active leaf cell to its neighbors no matter from which refinement level they are.

If we want the dilation operation to grow the active region with *uniform* speed into all directions we cannot pass information between cells from different refinement levels, e.g., when we want the growing cylinders to stay approximately circular. Hence, in situations where the dilation speed matters, we restrict the status transfer to the finest level voxels (*restricted dilation*). As a consequence we have to refine each cell that is adjacent to an active voxel, down to the finest level before it is activated. This is done by replacing the `pass_back_to_front_2()` procedure with

```
pass_back_to_front_2'(cell c1, cell c2) {
  if both cells are finest level voxels
     c2.next_status = c1.current_status
  else if c1 is an active finest level voxel
     if c2 is a leaf cell
        refine c2
     for each back sub-cell c2[1] ... c2[4]
        pass_back_to_front_2'(c1,c2[i])
  else if c1 is a leaf cell
     return
  else if c2 is a leaf cell
     for each front sub-cell c1[5] ... c1[8]
        pass_back_to_front_2'(c1[i],c2)
  else for each neighbor pair (c1[5],c2[1])...(c1[8],c2[4])
     pass_back_to_front_2'(c1[i+4],c2[i])
}
```

Obviously, depending on the number of restricted dilation steps, this will increase the complexity of the octree representation significantly. However, since we apply this kind of restricted dilation operators only to the "boundary" voxels, just a few regions of the volume are actually affected.

The elementary dilation operators can now be combined to implement our topology reconstruction algorithm, see Fig. 6. Each individual step performs a dilation *from status-x cells to status-y cells*. By this we mean that *status-x* is considered the active status for this operation and only cells with *status-y* are affected by the operation. All other cells' status remain unchanged.

Step (a): Initially all cells in the octree are "full" or "empty" and some voxels are tagged "boundary". We start by performing $s = \lceil \rho/\varepsilon \rceil$ *restricted* dilation steps from all "boundary" voxels to the "empty" cells. Since we define the topology of the output surface via the 6-neighborhood of "full" voxels, the restricted dilation propagates information only in the six major directions. All voxels that receive a status update during these steps are tagged "dilated". If the input data has gaps and holes that are smaller than ρ then these are filled with "dilated" voxels by this operation, see Fig. 6 a.

Step (b): We set all "empty" cells that are touching the outer faces of the root cell to "outside". This is correct due to our definition of the root cell in Sect. 5.1. Then we perform several unrestricted dilation steps from the "outside" cells to the "empty" cells until no more cells change their status. By this we in fact implement a seed filling algorithm that finds the complete outside component enclosing a solid which consists of "full", "dilated" and "empty" cells, see Fig. 6 b. The refinement in the disambiguation step (Sect. 5.2) guarantees that the seed filling actually reaches all parts of the outside component. Since we use the 6-neighborhood between "full" and "dilated" voxels, we use the 26-neighborhood for the complement, i.e., the "outside" voxels. This guarantees a compatible digital topology [Bischoff and Kobbelt 2002].

Step (c): Now we perform s dilation operations from the "outside" cells to the "dilated" cells. This shrinks back the boundary of the solid enclosed by the "outside" cells while not changing the status of any "full" cell, see Fig. 6 c. Since all "dilated" cells into which the "outside" cells propagate their status, have been generated in step (a), they already are from the finest level. Hence the propagation speed is uniform even if we do not apply restricted dilation operations.

After this last step, we have the surface topology of the output mesh implicitly defined by all 6-connected "full", "dilated" or "empty" cells that are 26-adjacent to an "outside" cell, see Fig. 6 d. If the input data happens to contain gaps and holes that are larger than the user-defined threshold ρ then these are not closed in this phase. However, the following phases of our model restoration algorithm are not handicapped by this. The algorithm will still produce a proper manifold surface which covers both the inside and the outside of the input surface. *Geometrically* the two sheets will coincide but *topologically* they are distinct. Notice that there is no way of automatically solving the general hole filling problem for large holes. If significant parts of the input geometry are missing then the *semantics* of the object cannot be recovered.

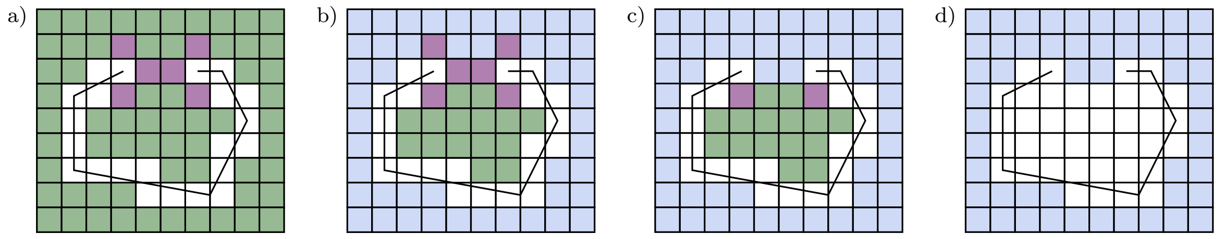


Fig. 6. Hole-fixing by morphological operations. (a) First the boundary cells are dilated (magenta) into the empty cells (green). (b) Next we determine the outside component (blue). (c) The outside component is dilated back into the already dilated cells (magenta). (d) The result is a clean separation of outside (blue) and inside cells (white) from which we deduce the surface topology of the output mesh.

5.4 Surface topology extraction

We turn the implicit voxel topology information into explicit mesh connectivity information by applying an extension of the dual contouring algorithm [Ju et al. 2002]. Our extension guarantees that the output mesh is a clean manifold by carefully splitting complex vertices and edges that are produced by the original algorithm.

We use a face based mesh representation where each face stores pointers to its neighboring faces and to its adjacent vertices. Note that we do not need an explicit representation for the edges of the mesh. In fact, using just the face neighborhood relation implies that the mesh data structure cannot represent complex edges at all.

The mesh is then build up in 3 steps: First we create the faces, second we set the pointers between neighboring faces and finally we create the vertices and set the corresponding vertex pointers.

Step 1: A grid-vertex in the octree is considered "outside" if it has at least one adjacent "outside" cell. A grid-edge is considered "inside" if all the surrounding cells are "full" or "dilated". We enumerate all pairs $(\mathbf{v}, e = (\mathbf{v}, \mathbf{w}))$ of "outside" grid-vertices $\mathbf{v}$ and incident "inside" grid-edges $e = (\mathbf{v}, \mathbf{w})$ where all adjacent cells are leaf-nodes. This is done by a recursive octree traversal procedure similar to [Ju et al. 2002]. For each such pair we create a *dual* polygonal face f, namely a triangle or quadrangle depending on whether e has three or four adjacent leaf cells. By duality, each edge of such a polygonal face f is associated with an octree grid-facet F. Finally we assign a *piercing point* and a *piercing normal*

$$\begin{aligned} \mathbf{p}_f &= \frac{2}{3}\mathbf{v} + \frac{1}{3}\mathbf{w} \\ \mathbf{n}_f &= \mathbf{v} - \mathbf{w} \end{aligned}$$

to f. The piercing point lies on e and provides a preliminary geometric embedding for the face f. This piercing point is necessary to distinguish between the two faces that might be associated with the edge e if both end points are "outside". Notice that the piercing point position is *not* used for reconstructing the geometric shape of the final surface. The piercing normal is chosen such as to always point to the "outside".

Step 2: We recursively visit all octree grid-facets F and collect the polygonal faces $f_1, \ldots, f_n$ associated with F (if any). Note that by construction, n is even. Note also that by duality, all f_i conceptually share an edge that is dual to the octree grid-facet F. Let $\mathbf{c}$ be the center of F and $\mathbf{p}_1, \ldots, \mathbf{p}_n$ the piercing points assigned to the faces $f_1, \ldots, f_n$. The vectors $\mathbf{p}_i - \mathbf{c}$ all lie in the supporting plane of F and hence induce a canonical counterclockwise ordering $f_{\pi(1)}, \ldots, f_{\pi(n)}$ of the faces f_i where we can assume without loss of generality that $\mathbf{n}_{f_{\pi(1)}}$ points in counter-clockwise direction. To establish the face connectivity, we double link the faces $f_{\pi(1)}, \ldots, f_{\pi(n)}$ in that cyclic order (see Fig. 7). Note that since we do not represent mesh edges explicitly, there is no need to split the complex edge common to all f_i.

Step 3: Finally, we create a new vertex for every cyclic triangle/quad fan in the polygonal mesh and set the pointers of the corresponding faces to this vertex. This is implemented by iterating over all faces and following the face-neighbor links until each n-sided face has all its n vertices. This procedure automatically splits complex vertices since a new copy is generated for each fan adjacent to it. Hence the result is guaranteed to be a closed and manifold polygonal mesh. Notice that in contrast to the original dual contouring algorithm, more than one vertex can be generated for a cell. The geometric position for the vertices is determined in the next step.

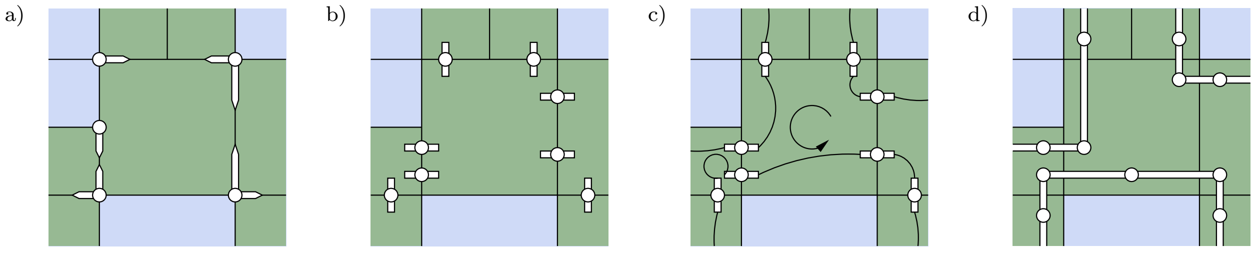


Fig. 7. The figure shows the supporting plane of an octree grid-facet. Octree facets that are adjacent to two full cells are marked green, facets that are adjacent to one or two empty cells are blue. For each pair of an "outside" vertex and an "inside" edge (a) we create a polygonal face f and its associated piercing point $\mathbf{p}_f$ symbolized by bars and dots in (b). The faces are then linked in counterclockwise order (c). Finally, we create a vertex for each cyclic triangle/quad fan (d). The actual geometric positions of these vertices are determined in the geometry reconstruction phase.

5.5 Surface geometry reconstruction

Once the mesh connectivity has been reconstructed, we have to compute the vertex positions. Each vertex is associated with one cell of the octree. If this cell is "full" we compute a feature sensitive sample point based on the set of triangles that are registered in this cell. If the cell is "dilated", i.e., it has been activated during the topological hole filling then there is no local geometric information available and we set the vertex position to "don't care". The actual position of these vertices is determined in the concluding post-processing phase.

After the hierarchical voxelization phase, all the input triangles are registered in the corresponding leaf cells. The refinement criterion guarantees that if a leaf cell is not from the finest level k then all the triangles that intersect this cell are numerically coplanar, i.e., they deviate by less than ε. Hence, we can be sure that the sample points computed for each voxel are satisfying the prescribed tolerance ε.

On the other hand, a common problem with CAD datasets are "double walls" and other internal redundant geometry. These are geometrically irrelevant because they do not contribute to the outside surface of the solid S but they still might affect the sample point computation if the internal geometry reaches into a cell where the restored surface passes through, i.e., a cell which is adjacent to an "outside" cell. Although the resulting geometric artifacts are not violating the error tolerance, they might nevertheless cause some disturbing deviation of the normal vectors (see Fig. 8).

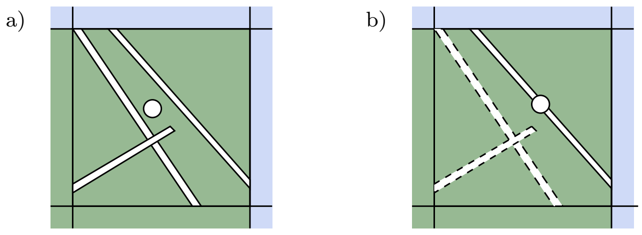


Fig. 8. Redundant internal geometry like double walls can affect the sample position (a). Although this does not violate the prescribed tolerance ε, we can usually further improve the quality of the output mesh by a local visibility test from the blue outside region. This test detects the redundant triangles and excludes them from the sample point computation (b).

Hence we can visually improve the quality of the output surface by locally determining for each cell which of the registered triangles actually contribute to the outside surface. The feature sensitive sample computation is then based only on these triangles. If the manifold extraction procedure generates more than one surface sheet within a cell, this local test has to be done for each copy of the splitted vertex.

To avoid complex visibility tests, we implemented the culling procedure for irrelevant geometry, again, by exploiting the graphics hardware. For each surface sheet we select one of the "outside" corners of the octree cell as the viewing position and define the three adjacent grid-facets as clipping planes. Then we render the registered triangles each with a different color and finally read out the frame buffer to check which triangles are visible at least in one pixel. After this local visibility test, we use only the remaining relevant triangles to compute the actual sample point.

It is tempting to try to make this visibility test globally, i.e., to determine which triangles are contributing to the outside surface in a pre-process and then to start the mesh restoration algorithm only with this subset. However, this pre-computation turns out to be as difficult as the mesh restoration itself since we cannot use a simple from-viewpoint-visibility test and some triangles might be partially relevant and partially irrelevant.

For computing a sample point in a "full" cell, we distinguish between the cases when all triangles lie in the same supporting plane, when they belong to two intersecting supporting planes, or when they belong to three or more planes. In order to minimize the number of potential fold-overs in the resulting mesh, we aim at finding sample points that lie in the interior of the corresponding cells. Hence among the candidates for a good sample point position we want to pick the one that is closest to the cell center in the $\|\cdot\|_\infty$-norm since this one could lie inside of the cell even in situations when the optimal sample according to the Euclidean norm lies outside [Varadhan et al. 2003]. For the cases of one or two supporting planes the $\|\cdot\|_\infty$-optimal point can be calculated easily, for three and more planes we fall back to the standard approach of the quadric error metric [Garland and Heckbert 1997] for feature sensitive sampling [Kobbelt et al. 2001].

One supporting plane: The optimal sample point that is closest to the cell center with respect to the $\|\cdot\|_\infty$-norm can be found by placing an infinitesimally small cube around the center and letting it grow until it touches the supporting plane of the triangle. Since a "full" voxel does not contain a boundary edge it is obvious that this growing cube touches the triangle with a corner first. In special axis-aligned configurations the growing cube might touch the triangle simultaneously with an edge or a face. In these situations we can pick any of the involved corners. The cells classified as “boundary” can be treated as “dilated” with no valid geometry information which implies a “don’t care” vertex position, or they can be handled as if they had two supporting planes intersecting along the boundary edge.

We compute the closest corner point by shooting rays from the cell center (which we take to be at the origin in what follows) into all space diagonal directions $[\pm 1, \pm 1, \pm 1]^T$. Let $E = [\mathbf{n}, d]$ be the normalized plane equation then the respective distances to the intersection points are

$$\lambda_{\pm,\pm,\pm} \;=\; \frac{-d}{\pm n_x \pm n_y \pm n_z}$$

and the minimum distance is

$$\lambda_{\min} \;=\; \frac{|d|}{|n_x| + |n_y| + |n_z|}.$$

The position of the corresponding sample $\mathbf{p}$ is then given by

$$\mathbf{p} \;=\; \lambda_{\min} \operatorname{sign}(d) \begin{pmatrix} \operatorname{sign}(n_x) \\ \operatorname{sign}(n_y) \\ \operatorname{sign}(n_z) \end{pmatrix}.$$

Two supporting planes: In this case we want to place the sample on the intersection line L between the two planes in order to preserve the potentially sharp edge. We use the same analogon of the growing cube and find that this time the cube will touch the (unbounded) intersection line L with one of its edges first. Again special axis-aligned cases can occur but we can still select one of the involved edges.

In order to find the nearest point on $L : \mathbf{q} + \nu\,\mathbf{r}$ to the cell center (= origin) in the $\|\cdot\|_\infty$-norm we have to compute intersections of L with the planes spanned by the cell center and the edges of the cell. We simplify the calculations by projecting the local configuration into each coordinate plane. For example in the xy-plane we find that the projection of L has the implicit form

$$[r_y, -r_x] \begin{pmatrix} x \\ y \end{pmatrix} \;=\; d_{xy} \;=\; q_x\, r_y - q_y\, r_x.$$

Shooting rays into all diagonal directions $[\pm 1, \pm 1]^T$ in the xy-plane gives the minimum distance

$$\lambda_{xy,\min} \;=\; \frac{|d_{xy}|}{|r_x| + |r_y|}.$$

Analogously we compute the minimum distances $\lambda_{yz,\min}$ and $\lambda_{zx,\min}$ in the other coordinate planes. Finally, the minimum spatial distance $\lambda_{\min}$ is the *maximum* of the three values $\lambda_{xy,\min}$, $\lambda_{yz,\min}$, and $\lambda_{zx,\min}$. Let $\lambda_{\min} = \lambda_{\xi\eta,\min}$ then the coordinates of the sample point $\mathbf{p}$ are eventually obtained by

$$\begin{pmatrix} p_\xi \\ p_\eta \end{pmatrix} \;=\; \lambda_{\min} \operatorname{sign}(d_{\xi\eta}) \begin{pmatrix} \operatorname{sign}(r_\eta) \\ -\operatorname{sign}(r_\xi) \end{pmatrix}$$

and the third coordinate can be read off the parametric formulation of L. If the sample position obtained by this construction happens to lie outside the cell, we simply discard it and set the corresponding vertex to “don’t care”. This situation occurs when two planes pass through a cell but their intersection line lies completely outside. We do not lose significant feature information by discarding this feature sensitive sample since the corresponding edge will be properly sampled in the neighboring voxel which is guaranteed to be full as well due to our voxelization strategy (see Fig. 9).

Three or more supporting planes For three supporting planes the optimal sample point is uniquely defined by the intersection of the three planes. If even more numerically non-coplanar planes have to be

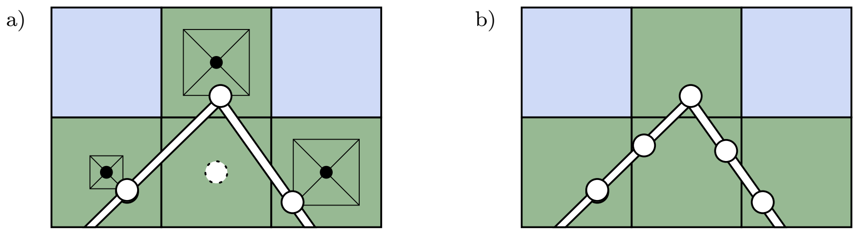


Fig. 9. (a) Sample points are computed by growing cubes from the cell centers. If only one plane intersects a voxel, the sample point is found along one of the voxel diagonals (bottom left and right). In case of two planes, the sample point is found along the edge of the cube (top cell). If a cell is intersected by three or more planes and the sample point lies outside of the cell, we set the sample point to "don't care" (bottom middle). (b) The corresponding vertex will be split in two by the manifold extraction procedure (Sect. 5.4) and the sample positions will be computed in the post-processing step (Sect. 5.6).

considered, the most reasonable way to define the optimal sample point is by using error quadrics and choosing that point which has the minimum squared distance to all involved planes [Garland and Heckbert 1997]. Again we discard the sample position and tag the vertex as "don't care" if the sample lies outside the current octree cell.

5.6 Post-Processing

5.6.1 *Computing the "don't care" positions.* In the last phase of the algorithm we already have a polygon mesh consisting of quads and triangles with most vertices having their coordinates assigned. What remains is the instantiation of the "don't care" vertices and the final triangulation which should avoid flipped triangles.

For the "don't care" vertices we could not compute a position in the earlier phases since they correspond to voxels that have been activated by the dilation operation to close holes and hence they cannot be associated with any input geometry. In the resulting mesh, the "don't care" vertices form small patches that span the holes in the input data and that are surrounded by vertices from "full" voxels. Hence the easiest way to define their vertex positions is to apply an iterative smoothing filter [Taubin 1995]. Since the vertices which correspond to the "full" voxels do not change their positions, the patches of the "don't care" vertices converge to smooth membrane patches that fill in the holes in the input data [Kobbelt et al. 1998]. Moreover, since the vertex positions obtained by feature sensitive sampling in the geometry reconstruction phase are not affected by the smoothing operation all sharp features of the input model are well preserved. Note, however, that we currently do not restrict the movement of the vertices during smoothing. Hence, in rare cases it may happen that the membrane surface intersects another part of the reconstruction.

To obtain a triangulated output mesh we have to split the quads generated with the extended dual contouring method by inserting diagonals. Here we have to choose the diagonals carefully in order to avoid flipped triangles. If the quad contains two opposite feature vertices we select the diagonal that connects these vertices, otherwise we simply pick that diagonal which leads to the smaller angle between the normals of the adjacent triangles.

A problem may arise if an edge is inserted multiple times, which may happen e.g.when we triangulate two opposing quads by the same diagonal or if more than two quads share a common edge. Then the resulting configuration can still be represented by any halfedge or face-based data structure but not by an indexed face set, which is unfortunately the structure used for many file formats. Hence, before saving our reconstruction to such a file format we proceed as follows: We detect double edges by enumerating for each vertex all incident edges. We disambiguate such edges by doing a 2-to-4 split (i.e., insert the midpoint of both copies of this "double" edge and connect it to the two opposite vertices of the adjacent triangles).

5.6.2 *Reducing the output complexity.* Even a polygon model with just a few triangles can have local configurations like tiny handles or tunnels that could be relevant or irrelevant depending on the design intent. We leave this decision to the user by letting him set the precision tolerance ε as an input parameter to our algorithm. Since this tolerance implies that the algorithm has to resolve all topological features up to this resolution, we have to refine the octree data structure up to the corresponding level if necessary. However, our algorithm automatically adapts the refinement depth to the local feature size and hence provides a maximally sparse representation. In particular the complexity (=number of octree nodes) of our data structure scales like $O(n)$ compared to $O(n^3)$ for a naive, uniform grid and to $O(n^2)$ for a restricted octree as is used in [Ju 2004]. Accordingly, the output complexity of our algorithm is an order of magnitude lower than that of other approaches which generally have to apply a marching cubes or dual contouring algorithm on the highest resolution. For example, the reconstruction at a resolution of 1000^3 of the architectural model at the top left of Table I produces approx. 1 million triangles with our algorithm compared to 6 million triangles for [Ju 2004] with the implementation provided by the author.

The output complexity of our algorithm could either be reduced by incorporating an octree pruning step *before* the surface extraction, as was done in [Ju et al. 2002]. This, however, would make the scheme less intuitive and simple and would have to rely on additional assumptions on the input geometry. Hence we have opted to apply a mesh decimation scheme *after* the extraction taking into account somewhat larger intermediate meshes. For our purposes we use a standard QEM decimation algorithm based on half-edge collapses that was modified to provide feature preservation by taking into account the normal variation before and after the collapse. Since efficient out-of-core decimation techniques are available [Wu and Kobbelt 2003], this is not considered a serious bottleneck.

6. RESULTS

We have run our algorithm on a large variety of technical and architectural CAD models (see Fig. 10, 11, 12, 13). The algorithm never failed to produce a closed and consistent manifold mesh. Table I gives an overview over some measurements. The timings for "voxelization" include steps 5.1 to 5.3 while "geometry" includes 5.4 to 5.6 (without post-decimation).

The running time of our algorithm depends on the prescribed tolerance ε as well as on the amount of detail in the object. In flat regions the refinement stops on coarse levels while in the vicinity of sharp corners and boundaries the refinement goes down to the finest level.

A problem with some architectural models is the presence of double walls that are extremely close together. This can trigger excessive refinement in apparently flat regions of the input model because the local refinement criterion in the first phase does not have access to the inside/outside information computed in the third step.

The resampling of the original data preserves all geometric features but it also increases the number of faces significantly. This is because our input models are usually generated by hand and hence faces are placed intelligently. In contrast our automatic resampling does not know the model "semantics" and decides locally where to put the sample points. However, we can reduce the output complexity down to a size that is comparable to that of the input mesh. In the examples shown in Figure 11 we have applied a standard QEM decimation algorithm that has been extended by a normal cone criterion. This criterion prevents edge-collapses that would result in a strong normal deviation of the adjacent triangles and leads to a better preservation of the sharp features. If the output becomes very large we can simplify it out-of-core using a streaming decimation algorithm [Wu and Kobbelt 2003].

Figure 12 demonstrates the automatic hole detection and hole closing capabilities of our algorithm. Note that even if we skip the hole closing stage, the output of our algorithm will still be manifold, although some parts of the surface might be triangulated from both sides.

Finally, Figure 13 shows how our algorithm nicely removes unwanted interior geometry that is due to the architect arbitrarily sticking together the pieces of the model.

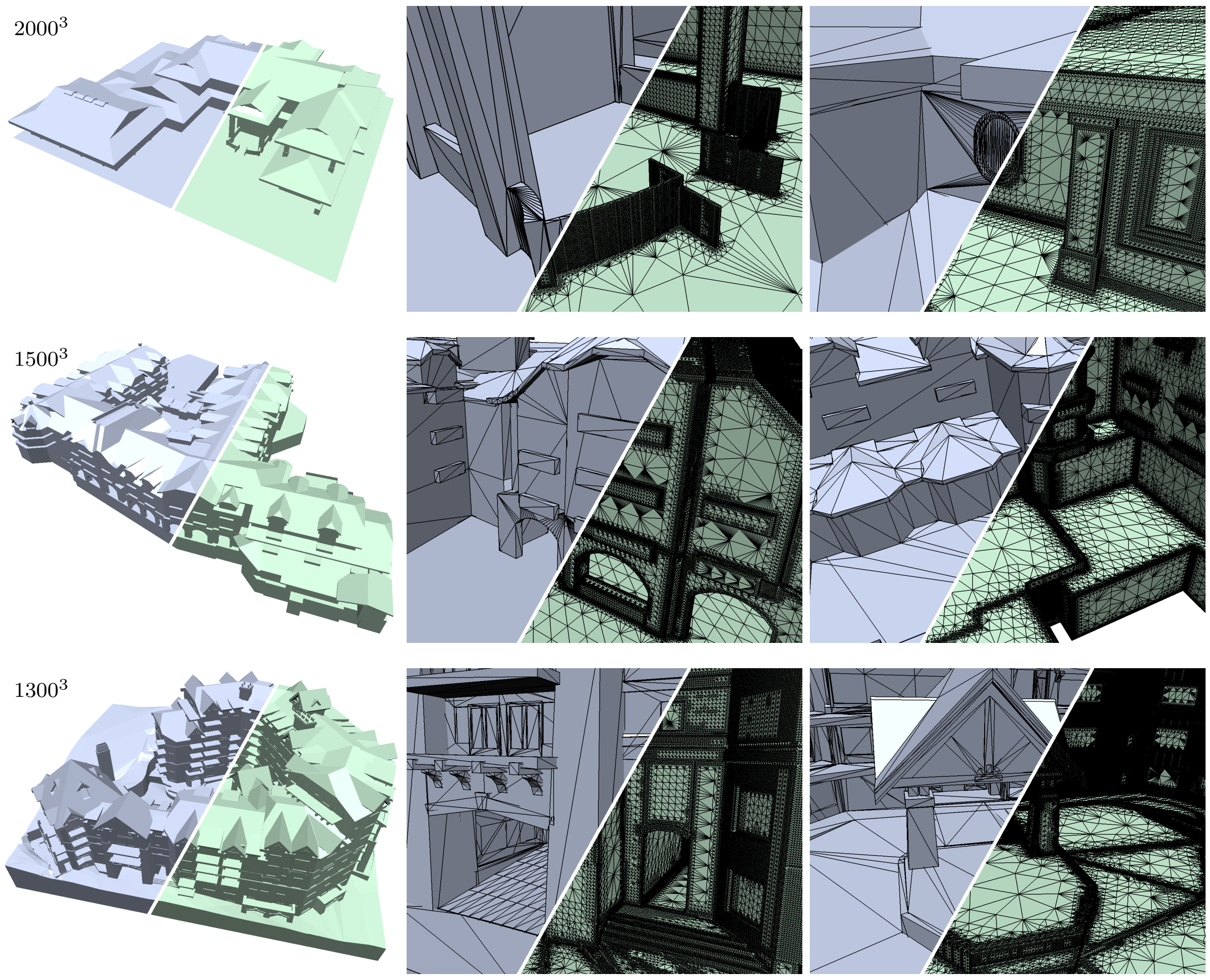


Fig. 10. The figures above show the reconstructions of various architectural models. The original models shown in blue have a large number of artifacts since most features are modeled as individual objects that are inconsistently stuck together. Note how the size of the triangles of the reconstructed models (green) adapts to the local feature size in the original models.

7. CONCLUSION AND FUTURE WORK

With the algorithm that we present in this paper the reconstruction of a clean and consistent triangle mesh from an inconsistent input mesh can be done fully automatically. The algorithm just requires two parameters, one is the error tolerance ε and the other is the maximum size ρ up to which gaps and holes in the surface should be fixed. The algorithm is guaranteed to produce a correct result since topology reconstruction is done based on the digital topology of an (adaptive) voxel grid. For the mesh restoration only the geometric location of the input triangles is used and no consistent normal orientation or connectivity information is required.

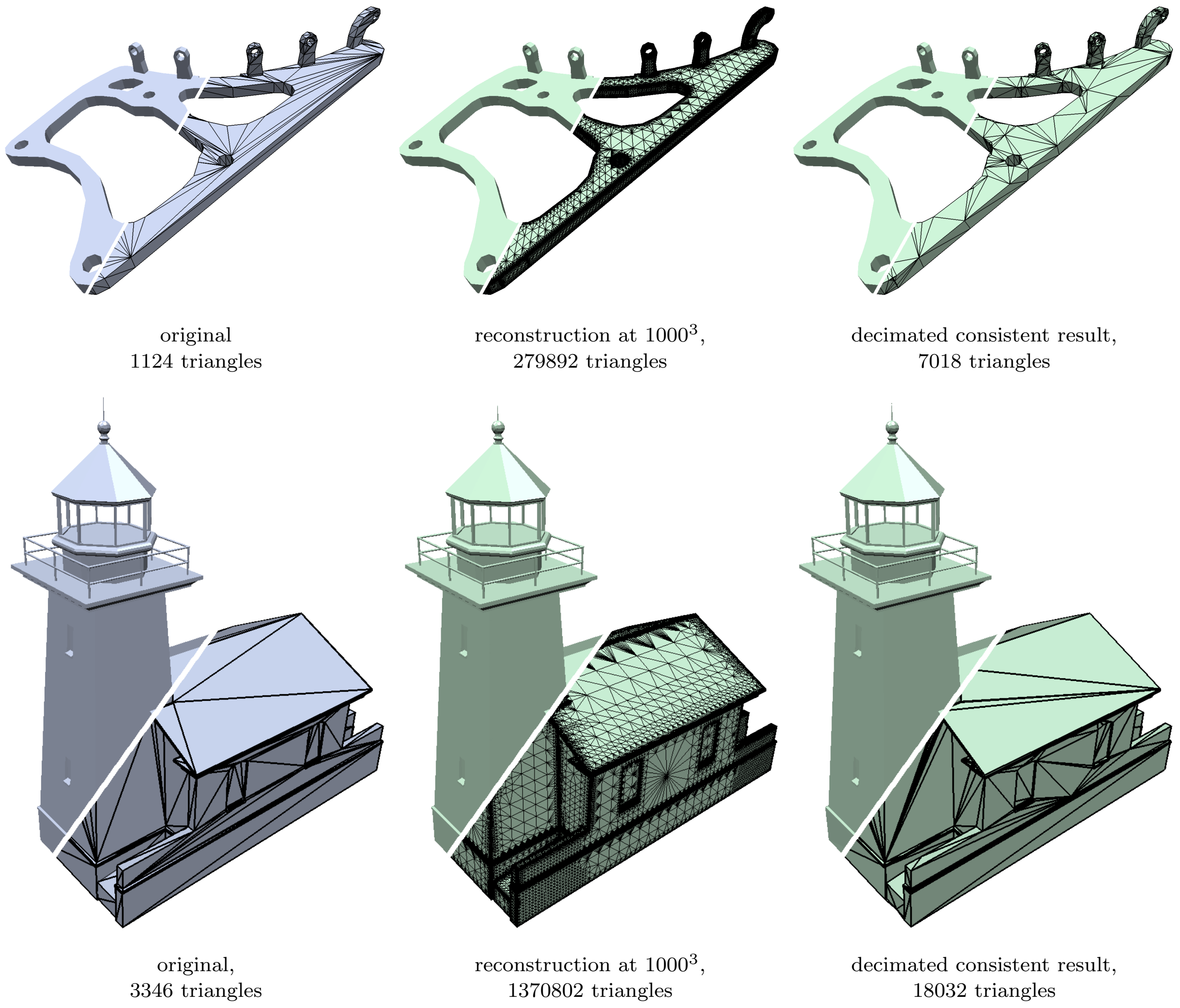

original
1124 triangles
reconstruction at 1000^3,
279892 triangles
decimated consistent result,
7018 triangles
original,
3346 triangles
reconstruction at 1000^3,
1370802 triangles
decimated consistent result,
18032 triangles

Fig. 11. Our algorithm automatically adapts the octree refinement depth to the local feature size of the input models. However, to further reduce the number of triangles in the output mesh, we can apply a standard mesh decimation algorithm to the reconstruction. Here we used an algorithm that is based on quadric error metrics and that incorporates a normal cone constraint for better feature preservation.

In our experiments we demonstrate the restoration of complex architectural CAD models that have many inconsistencies and geometric features from a large range of magnitudes. Due to the effective use of a hierarchical octree data structure we could run voxel resolutions as high as 4096^3 and the complete restoration process usually took only a few minutes. Only on very detailed input models (Table I, top right) the reconstruction might take considerably longer, which we attribute to memory swapping effects on our 2GB machine.

In rare cases our algorithm may accidentally fill in features like notches that are located in the immediate vicinity of boundary edges. Hence, we plan to design and evaluate heuristics to detect and preserve gaps

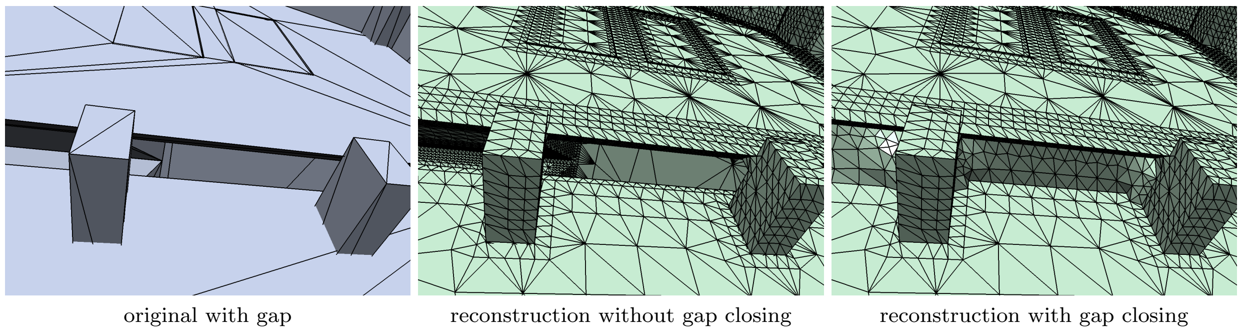

original with gap
reconstruction without gap closing
reconstruction with gap closing

Fig. 12. Our algorithm automatically fills holes up to a user prescribed threshold with smooth membrane surfaces (right). Even if the hole-filling step is skipped, the reconstruction still has a topologically valid connectivity (middle) but every wall is triangulated from both sides.

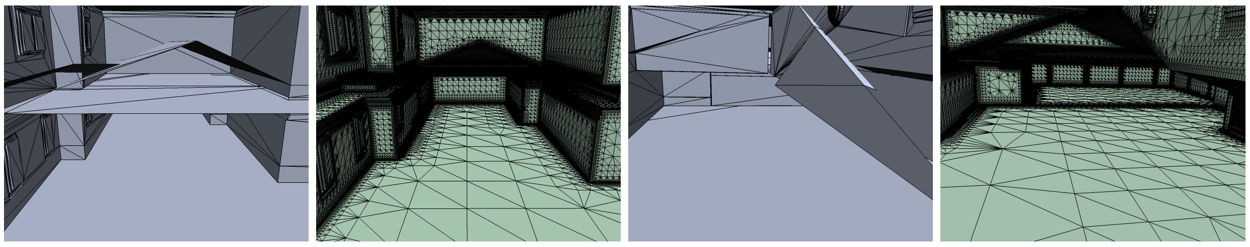


Fig. 13. The figure above shows the interior of an architectural model before (blue) and after (green) reconstruction. Note that the algorithm has successfully removed all interior dangling triangles.

that are actually intended by the designer.

Our next step towards an improvement of the algorithm is to enable virtually infinite voxel resolutions (and hence arbitrary precision) by decomposing the global octree into smaller bricks that are processed one at a time in main memory. The dependencies between neighboring bricks can be taken care of by providing sufficient overlap. The thin wall problem is still another open research question which has not been addressed in the literature.

REFERENCES

Andújar, C., Brunet, P., and Ayala, D. 2002. Topology-reducing surface simplification using a discrete solid representation. *ACM Trans. Graph. 21,* 2, 88–105.

Barequet, G., Duncan, C., and Kumar, S. 1998. Rsvp: A geometric toolkit for controlled repair of solid models. *IEEE Trans. on Visualization and Computer Graphics 4,* 2, 162–177.

Barequet, G. and Kumar, S. 1997. Repairing cad models. In *Proc. IEEE Visualization*. 363–370.

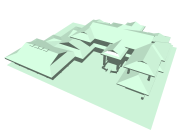
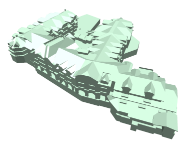
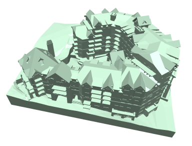
Barequet, G. and Sharir, M. 1995. Filling gaps in the boundary of a polyhedron. *Computer-Aided Geometric Design 12,* 2, 207–229.

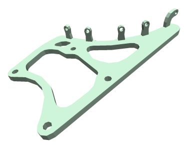
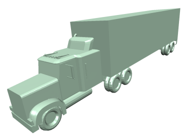
Bischoff, S. and Kobbelt, L. 2002. Isosurface reconstruction with topology control. In *Proc. Pacific Graphics 2002*. 246–255.

Bøhn, J. and Wozny, M. 1992. Automatic cad-model repair: Shell-closure. In *Proc. Symp. on Solid Freeform Fabrication*. 86–94.

Borodin, P., Novotni, M., and Klein, R. 2002. Progressive gap closing for mesh repairing. In *Advances in Modelling, Animation and Rendering*, J. Vince and R. Earnshaw, Eds. Springer Verlag, 201–213.

Brunet, P. and Navazo, I. 1990. Solid representation and operation using extended octrees. *ACM Trans. Graph. 9,* 2, 170–197.

model									
input triangles	8540			11904			50056		
resolution	500^3	1000^3	1500^3	500^3	1000^3	1500^3	500^3	1000^3	1300^3
#cells (×1000)	661	1702	2920	1042	4019	7610	4120	17411	29518
#triangles (×1000)	406	973	1681	1187	3882	6890	1459	4780	7421
total memory (MB)	71	155	296	155	495	947	331	1387	1751
voxelization (s)	9	33	141	18	255	311	110	1545	4852
geometry (s)	8	17	31	25	91	186	43	220	331
total time (s)	17	50	172	43	346	497	153	1765	5183

model									
input triangles	3346			1124			18340		
resolution	500^3	1000^3	1500^3	1000^3	2000^3	4000^3	500^3	1000^3	1500^3
#cells (×1000)	414	1142	2090	388	881	1916	329	1210	2544
#triangles (×1000)	529	1370	2391	279	646	1370	259	897	1712
total memory (MB)	68	166	293	44	97	198	37	137	271
voxelization (s)	6	39	34	10	32	72	5	17	43
geometry (s)	12	28	81	4	14	27	6	23	34
total time (s)	18	67	115	14	46	99	11	40	77

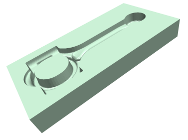
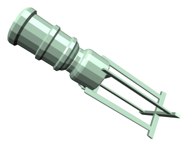
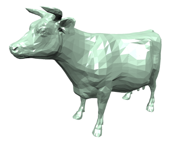
model									
input triangles	6442			2133			5804		
resolution	500^3	1000^3	2000^3	500^3	1000^3	1500^3	500^3	1000^3	1500^3
#cells (×1000)	238	721	2119	423	1242	2261	801	2536	4621
#triangles (×1000)	188	550	1582	272	774	1410	592	1829	3285
total memory (MB)	29	94	280	45	135	265	95	297	577
voxelization (s)	4	16	52	5	16	41	12	42	94
geometry (s)	3	14	53	5	14	26	13	41	71
total time (s)	7	30	105	10	30	67	25	83	165

Table I. Experimental results for the voxelization and extraction of the geometry measured on a Pentium 4, 2 GHz, 2 GB system for various models and resolutions ($\varepsilon = 1/resolution$). The top left model contained some holes which were automatically filled ($\rho = \varepsilon$). Note that the table shows the output sizes *without* post-decimation.

Davis, J., Marschner, S., Garr, M., and Levoy, M. 2002. Filling holes in complex surfaces using volumetric diffusion. In *Proc. International Symposium on 3D Data Processing, Visualization, Transmission*. 428–438.

Dolenc, A. and Mäkelä, I. 1991. Optimized triangulation of parametric surfaces. In *Technical Report TKO-B74, Helsinki University of Technology.*

Frisken, S. F., Perry, R. N., Rockwood, A. P., and Jones, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proc. SIGGRAPH 00*. 249–254.

Garland, M. and Heckbert, P. S. 1997. Surface simplification using quadric error metrics. In *Proc. SIGGRAPH 97*. 209–216.

Gibson, S. F. F. 1998. Using distance maps for accurate surface representation in sampled volumes. In *Proc. IEEE Symposium on Volume Visualization.* 23–30.

Gonzalez, R. and Woods, R. 1992. *Digital Image Processing.* Addison-Wesley.

Gottschalk, S., Lin, M. C., and Manocha, D. 1996. Obbtree: A hierarchical structure for rapid interference detection. In *Proc. SIGGRAPH 96.* 171–180.

Gress, A. and Klein, R. 2003. Efficient representation and extraction of 2-manifold isosurfaces using kd-trees. In *Proc. 11th Pacific Conference on Computer Graphics and Applications (PG 2003).* 364–376.

Guéziec, A., Taubin, G., Lazarus, F., and Horn, B. 2001. Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics 7,* 2, 136–151.

Guskov, I. and Wood, Z. J. 2001. Topological noise removal. In *Proc. Graphics Interface 2001.* 19–26.

Ju, T. 2004. Robust repair of polygonal models. *ACM Trans. Graph. 23,* 3, 888–895.

Ju, T., Losasso, F., Schaefer, S., and Warren, J. 2002. Dual contouring of hermite data. In *Proc. SIGGRAPH 02.* 339–346.

Kobbelt, L., Campagna, S., Vorsatz, J., and Seidel, H.-P. 1998. Interactive multi-resolution modeling on arbitrary meshes. In *Proc. SIGGRAPH 98.* 105–114.

Kobbelt, L. P., Botsch, M., Schwanecke, U., and Seidel, H.-P. 2001. Feature sensitive surface extraction from volume data. In *Proc. SIGGRAPH 01.* 57–66.

Kong, T. and Rosenfeld, A. 1989. Digital topology: Introduction and survey. *Computer Vision, Graphics and Image Processing 48*, 357–397.

Liepa, P. 2003. Filling holes in meshes. In *Proc. Symposium on Geometry Processing 03.* 200–205.

Lorensen, W. E. and Cline, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proc. SIGGRAPH 87.* 163–169.

Nooruddin, F. and Turk, G. 2003. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics 9,* 2, 191–205.

Samet, H. and Webber, R. E. 1988. Hierarchical data structures and algorithms for computer graphics. part i. *IEEE Comput. Graph. Appl. 8,* 3, 48–68.

Taubin, G. 1995. A signal processing approach to fair surface design. In *Proc. SIGGRAPH 95.* 351–358.

Turk, G. and Levoy, M. 1994. Zippered polygon meshes from range images. In *Proc. SIGGRAPH 94.* 311–318.

Varadhan, G., Krishnan, S., Kim, Y., Diggavi, S., and Manocha, D. 2003. Efficient max-norm distance computation and reliable voxelization. In *Proc. Symposium on Geometry Processing.* 116–126.

Weihe, K. and Willhalm, T. 1998. Why cad data repair requires discrete algorithmic techniques. In *Proc. 2nd Workshop on Algorithm Engineering.* 1–12.

Wu, J. and Kobbelt, L. 2003. A stream algorithm for the decimation of massive meshes. In *Proc. Graphics Interface 2003.* 185–192.

Received Month Year; revised Month Year; accepted Month Year