EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 2

High-Resolution Volumetric Computation of Offset Surfaces with Feature Preservation

Darko Pavić and Leif Kobbelt

Computer Graphics Group, RWTH Aachen University, Germany

Abstract

We present a new algorithm for the efficient and reliable generation of offset surfaces for polygonal meshes. The algorithm is robust with respect to degenerate configurations and computes (self-)intersection free offsets that do not miss small and thin components. The results are correct within a prescribed ε-tolerance. This is achieved by using a volumetric approach where the offset surface is defined as the union of a set of spheres, cylinders, and prisms instead of surface-based approaches that generally construct an offset surface by shifting the input mesh in normal direction. Since we are using the unsigned distance field, we can handle any type of topological inconsistencies including non-manifold configurations and degenerate triangles. A simple but effective mesh operation allows us to detect and include sharp features (shocks) into the output mesh and to preserve them during post-processing (decimation and smoothing). We discretize the distance function by an efficient multi-level scheme on an adaptive octree data structure. The problem of limited voxel resolutions inherent to every volumetric approach is avoided by breaking the bounding volume into smaller tiles and processing them independently. This allows for almost arbitrarily high voxel resolutions on a commodity PC while keeping the output mesh complexity low. The quality and performance of our algorithm is demonstrated for a number of challenging examples.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Offset surfaces play a very important role in geometry processing and especially in various CAD/CAM applications. They can be used for tolerance analysis in machine processing and collision detection. In the context of tool path generation for numerically controlled (NC) milling machines, offset surfaces are used to define the domain where the machine tool positions are constrained to lie and they provide the input for collision-free path planning. Furthermore offset surfaces are used in finite element modeling, electrical circuit design, for generating hollowed or shelled versions of surface models, filleting and rounding of 3D models, as well as for morphological operations on geometric models.

Polygonal meshes are the most popular representation for 3D models because they provide the simplest way to approximate any possible shape. Since all commercial CAD systems are able to handle polygonal meshes or at least provide import and export routines for them, polygonal meshes can be seen as the universal geometry representation for interchange. The most common standard data format is the STL format (STereoLithography), where meshes are represented as *triangle soups*, i.e. as sets of triangles without any additional connectivity information. Such a format is on the one hand easy to generate, but on the other hand when transferring STL-files between systems, various types of inconsistencies can occur like thin gaps, holes, and flipped orientation. In this paper we propose a novel method for computing offset surfaces for polygonal meshes which is able to handle any kind of such inconsistencies.

An offset surface of a solid is the set of points having the same distance δ (*offset distance*) from the original geometry. The offsetting operation can be understood as a special case of the Minkowski sum which is a well explored operation in mathematical morphology [Ser83]. The Minkowski sum of two sets M and S in Euclidian space is defined as $M \oplus S = \{m+s | m \in M, s \in S\}$. If we take M to be an arbitrary input mesh and S a sphere of the given radius δ centered at the origin then an offset surface is defined as the boundary of their Minkowski sum. Notice that this definition is

based on an unsigned distance function. Hence for closed objects, an offset surface usually falls into at least two connected components (inner and outer).

For a polygonal mesh, the Minkowski sum can be decomposed into a set of spheres, cylinders, and prisms corresponding to vertices, edges, and faces of the mesh. A constructive solid geometry approach to offset surface computation is based on computing the union of all these elements, i.e., computing the minimum of the superposition of all the unsigned distance fields associated with these elements. In our algorithm we follow a volumetric approach to identify the cells in a voxel grid that are intersected by the offset surface and extract a polygonal representation from them. The most important properties of our algorithm are:

- **hierarchical**: By using an octree data structure, high voxel resolution is only generated in regions that are actually affected by the offset surface. Refining an octree cell is done only if the minimum distance of the cell to the input geometry is below the offset distance and the maximum is above the offset distance.
- **features**: Sharp features on the offset surface, which are caused by concave regions of the input geometry, are properly detected and reconstructed. Hence the visual quality and accuracy of the output is not affected by the fact that the distance function is discretely sampled on an adaptive octree grid.
- **correctness**: The algorithm is guaranteed to produce the geometrically correct output within a prescribed ε-tolerance even for input meshes with topological inconsistencies. No thin parts are lost. The topological resolution, i.e., the minimum distance between separate sheets is bounded by the minimum voxel size or, equivalently, by the maximum refinement level of the octree.
- **scalability**: Since each sub-region of the input can be processed independently from the others, it is straightforward to confine the offset computation to a sub-cell of the embedding space. This allows for virtually unlimited voxel resolutions since the bounding volume can be split into (overlapping) tiles, which can be processed sequentially to save memory or in parallel to save computation time.

There are two major observations that have inspired our approach to offset computation and which distinguish our algorithm from previous approaches.

- **transpose computation:** For a proper evaluation of the distance function within a voxel cell, we would have to compute the minimum distance for each point within the cell to all triangles and then take the minimum and maximum of these distances across the cell. Instead we compute for each triangle the minium and maximum distances within a cell by a closed formula and then take the minimum over all triangles. While this computation is correct for the minimum distance, it provides only a conservative estimate of the true maximum distance (max of min $\leq$ min of max). Hence, we base the actual offset computation only on the minum values and use the maximum distance estimates as an efficient refinement criterion for the adaptive octree.
- **offsets vs. (zero) iso-contours:** The computation of offset surfaces is a very special instance of the more general problem of iso-contour extraction. The major difference is that for offsets the part of the input geometry that affects a certain region of the output surface has the distance δ. Hence in a sufficiently refined voxel grid (or from a certain octree level on) the input geometry always lies *outside* the cell for which the distance function has to be computed. This strongly reduces the number of special configurations to be considered and efficient classification schemes known from polygon clipping can be exploited.

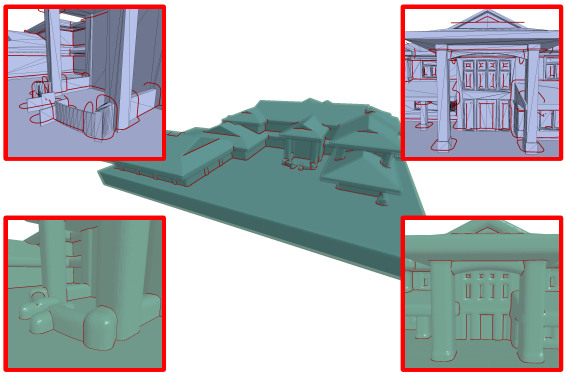

Figure 1: *Here we show the offsetting result for an architectural model having all kinds of inconsistencies like, holes, gaps, overlaps, double walls and self-intersections. The zoomed views show complex regions of the model with sharp feature lines in red.*

Fig.1 shows an example of an offset generated with our algorithm. Notice the quality of the extracted features. Since our approach is volumetric, self-intersections are elimintated automatically.

2. Related Work

The mathematical basis for offsetting of solids is described in earlier work by Rossignac et al. [RR85]. There the offseting operation is introduced as a new solid-to-solid transformation and associated with methods like filleting and rounding of solids. A number of methods for computing offset surfaces have been suggested since then.

An offset surface can be generated by creating solid primitives (for each vertex a sphere, for each edge a cylinder and for each face another parallel face) and combining those by trimming to the final offset surface [RR85, For95]. This is a computationally rather involved process and trimming at tangential intersections is numerically very unstable. Our algorithm is also based on computing the union of a set of primitives. However, our computations are stable since we work on a volumetric representation and hence intersections are computed by min/max operations applied to distance

functions. Since our algorithm is a hierarchical approach where the offset surface is intermediately represented by an adaptively refined octree, it can be understood as a kind of an adaptively sampled distance field [FPRJ00]. In order to be able to extract the offset of a given surface we are computing not only the minimum but also the maximum distance in each cell. Adaptive subdivision is an approach often used, e.g., in the context of isosurface extraction [VKSM04], where usually sampling of a volumetric function is done at cell corners. In contrary we are estimating minimum and maximum distance for the cells as a whole.

Other surface-based approaches for generating offset surfaces simply shift the original vertices in offset direction [QS03]. This is problematic when it comes to handle self-intersections which can either occur locally in areas of high curvature or globally when different parts of the input mesh meet. If the input mesh is convex or decomposed into convex pieces then this approach is simple and effective [VKKM03]. In [CVM*96] simplification envelopes are introduced for global error-control in mesh simplification. Their offset surface generation method requires manifold meshes, which do not contain any degenerated configurations. In contrary our algorithm can process all kinds of mesh inconsistencies since we treat every triangle independently.

Offsetting is a very important operation in layered manufacturing and in this context, approaches were introduced where 3D offsetting is reduced to computing 2D offsets of the 2D contours generated by slicing the input geometry [MS00]. These methods, however, are not applicable for more general scenarios. Since offsetting can be understood as a morphological operation it is an intuitive approach to extend the 2D pixel-based erosion and dilation operations [GW01] to 3D resulting in a very simple volumetric offsetting approach, where the 26-neighborhood in a voxel-grid is used to propagate distance information [GZ95]. Obviously, the accumulation of errors with increasing offset distances is the main problem of this method.

More advanced volumetric methods were presented based on distance volumes and the fast marching method [BMW98, BM99]. While these methods work on regular voxel grids, we use an adaptive octree datastructure instead, which allows for much higher voxel resolutions for a given memory budget. Moreover, the approximation properties of fast marching [Set99, OF02] do not allow for high accuracy. In contrast, our method uses accurate distance computations for the offset surface extraction.

Varadhan et al. [VM04] have proposed a method to approximate the Minkowski sum of polyhedral models, which covers also the computation of offset surfaces as a special case. Since our approach is especially designed for offset surface computation it runs much faster (see Section 8). The acceleration is mostly due to our cell-to-X (X $\in$ { vertex, edge, triangle } distance computation described in Section 4 which is more specialized than the max-norm distance computation in [VM04]. Furthermore our method is immune to all kinds of degeneracies in the input model whereas their method requires closed manifolds which are free from artifacts like self-intersections.

Recently, a point-based offsetting approach was introduced [CWRR05b, CWRR05a]. Here point samples are first generated on the input surface and then moved in normal direction. The Minkowski sum volume is rasterized on a regular voxel grid in order to remove self-intersections. In a regular grid, the computational and memory complexity is growing cubically with the voxel resolution. In our algorithm, voxel cells are generated on demand by traversing an octree in breadth first order. This guarantees that only those cells are generated which are actually needed for the offset surface extraction. This implies that the complexity only grows quadratically with the resolution. In [HLC*01, HC02], another offsetting approach is presented which is mostly aiming at visualizing the offset via surface splats. In their case the classification whether a cell intersects the offset surface is based on conservative estimates for both the minimum and maximum distance. While this is sufficient for visualization purposes, it would be non-trivial to extract a proper manifold offset surface from it. This is why we use an estimate only for the maximum distance and compute the exact minimum distance for each cell. To the resulting adaptive voxel grid we apply the surface extraction scheme of [BPK05] to generate a guaranteed manifold output mesh.

Feature preservation for offset surfaces is addressed in [QZS*04] where the spatial cells are adjusted to align with gradient discontinuities. We use a standard adaptive octree and recover features from normal information [KBSS01].

3. Algorithm Description

The input to our algorithm is an arbitrary, maybe non-manifold or otherwise degenerated polygonal mesh $M = (V, E, F)$ consisting of a set of vertices V, a set of edges E and a set of faces F. Moreover the user specifies an offset distance δ and a maximum octree level $\mathcal{L}$. The maximum octree level obviously limits the topological resolution ε of the offset surface since in each cell only one sheet of the surface can be extracted. Hence sheets of the offset surface which are closer than the size of a voxel are implicitly merged. This limitation is acceptable for most practical applications since the input STL file is usually only an approximation of some unknown object surface anyway. Each of the *elements* (vertex, edge, or face) of the mesh defines an unsigned distance function in space (represented by a sphere, cylinder or prism). The offset surface of M is computed by taking the minimum over all these distance functions in space.

3.1. Rasterization phase

The rasterization of the offset surface is done by traversing an octree in breadth first order and splitting each cell which is potentially intersected by the offset surface, i.e. for which the minimum distance to M is less than δ and (a conservative estimate of) the maximum distance is larger than δ.

In our transposed computation the minimum distance for a cell can be found by simply taking the minimum of the distances with respect to all vertex, edge, and face primitives (see Section 4). If the cell does not intersect the input surface, the minimum distance is always found somewhere on the boundary of a cell (otherwise it is zero). For the maximum distance within a cell we would have to find that point, which has the maximum minimum distance to any primitive in the input. This information, however, is not available since we are processing the individual primitives independently. Hence we have to settle with a conservative estimate of the maximum distance, the most simple one being the minimum distance plus the diagonal of the cell. In Section 4 we will describe a tighter estimate. The remaining false positives, i.e. cells where the true maximum distance is below δ while our estimate is above δ will be detected and discarded later in the mesh extraction phase (see Section 3.2).

Our main data structure is a (linearized) octree where the children of a cell are grouped in blocks of 8 cells. The cells are defined as:

```
1  struct OctreeCellData {
2    int first_child;
3    int location[3];
4    float minDist, maxDist;
5    float minPoint[3], minNormal[3];
6    Data* primitives;
7  };
8  struct Data {
9    Vertex_List V;
10   Edge_List E;
11   Face_List F;
12 };
```

Due to the linear layout, we only need to store the index of the first child node, the others follow in the next seven entries. The integer location of the cell is stored for efficiency reasons. During the computation, **minDist** and **maxDist** hold the currently best (lowest) estimate for the respective distances. In addition we store the position **minPoint** where the current minimum distance on the cell boundary is taken on and the normal vector **minNormal** pointing to the corresponding base point on the input surface M. This information is used in the surface extraction phase (see Section 3.2) to compute an offset surface sample within the cell. The pointer **primitives** points to a set of lists that store those mesh primitives which can have an effect on this cell, i.e., for which the offset distance lies within the interval between minimum and maximum distance. While maintaining these lists per cell causes some memory overhead, it effectively avoids many redundant computations on refined octree levels.

The pseudo-code of our rasterization method is:

```
1  root.minDist := 0;
2  root.maxDist := FLT_MAX;
3  root.primitive→V := {all vertices}
4  root.primitive→E := {all edge}
5  root.primitive→F := {all faces}
6  for level = 1 to ℒ
7    for all cells C from level−1
8      if (C.minDist ≤ δ ≤ C.maxDist)
9      {
10       SPLIT(C);
11       for D = CHILD(C,0) ... CHILD(C,7)
12       {
13         for all v_i ∈ C.primitive→V
14           SPHERE_MINMAX(v_i, D);
15         for all e_j ∈ C.primitive→E
16           CYLINDER_MINMAX(e_j, D);
17         for all f_k ∈ C.primitive→F
18           PRISM_MINMAX(f_k, D);
19       }
20     }
```

We initialize the root cell of the octree and push all mesh elements into the list of relevant primitives. In the main loop we traverse the octree level by level. Each surface cell from the previous level is split and for each of its children the minimum and maximum distance is computed. By this approach we refine the octree only when and where it is needed. The template for the distance computation is:

```
1  <ELEMENT>_MINMAX(ELEMENT X, CELL C)
2  {
3    D_min = min_{c_i∈C} { min_{x_j∈X} |||c_i − x_j||| };
4    D_max = max_{c_i∈C} { min_{x_j∈X} |||c_i − x_j||| };
5
6    C.minDist = min(C.minDist, D_min);
7    C.maxDist = min(C.maxDist, D_max);
8
9    if (D_min ≤ δ ≤ D_max)
10     C.primitives->Push(X);
11 }
```

The different versions of the minmax procedures for spheres, cylinders, and prisms only differ in the way how the values $D_{\min}$ and $D_{\max}$ are calculated. A detailed description is given in Section 4. The minimum operation in line 6 guarantees that the global minimum is computed correctly. The maximum, obtained by taking the minimum of the maxima (see Line 7), is in general only a conservative estimate of the true maximum distance. This can cause some slight computational overhead due to false positive cells[†] being split but it does not affect the overall correctness of the algorithm since these cells are discarded in the mesh extraction phase. Figure 2 depicts such a situation. Notice that false positives can only occur near the medial axis of the input geometry (in cells where the maximum distance is not taken on at one of the corners) and only in the interior of the offset volume (since the minimum distance is always computed correctly).

[†] i.e. cells that are wrongly assumed to intersect the offset surface.

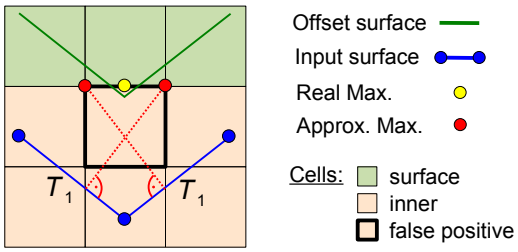

Figure 2: *Example for a false positive octree cell. Although the maximum distance is computed correctly for each individual triangle T_i, the combination may lead to a conservative estimate, when the cell is intersected by the medial axis. Eventually such* false positive *cells are never used for mesh extraction since they are not adjacent to an outer cell.*

3.2. Offset Surface Extraction

After the rasterization we have a number of *surface cells* on the finest resolution level $\mathcal{L}$ that are intersected by the offset surface, i.e., for which $minDist \leq \delta \leq maxDist$. The false positives among these surface cells are easily detected as those which do *not* have a neighbor cell with $\delta \leq minDist$ (*outer cell*). See Figure 2 for details. All false positives are discarded from mesh extraction. To the remaining surface cells we apply the variant [BPK05] of the Dual Contouring Algorithm [JLSW02]. This variant is guaranteed to produce manifold meshes and it extracts a mesh which is topologically equivalent to the boundary between surface cells and outer cells without requiring additional inside/outside information.

What remains to be done is the computation of a surface sample in each cell. For this we use the minimum distance information $d = minDist$, $\mathbf{p} = minPoint$, and $\mathbf{n} = minNormal$ and set up the plane equation

$$\mathbf{n}^T\mathbf{x} = \mathbf{n}^T\mathbf{p} + \delta - d$$

which represents the tangent plane of M shifted by δ. This is the best planar approximation to the offset surface within the current cell. We define a surface sample by projecting the center $\mathbf{c}$ of the cell to that plane (see Fig.3), i.e.,

$$\mathbf{c}' = \mathbf{c} + \mathbf{n}\left(\mathbf{n}^T(\mathbf{p}-\mathbf{c}) + \delta - d\right)$$

If the assumption that the offset surface is locally smooth (flat) does not hold, the sample $\mathbf{c}'$ might actually not lie on the offset surface (Fig.3). Furthermore the sample $\mathbf{c}'$ might even lie outside the cell (Fig.3). The samples computed outside the cell are simply clamped. These problems will be taken care of later in the smoothing step (Section 6).

3.3. Volume Tiling

So far we implicitly assumed that the octree root cell is initialized as the bounding box of the offset surface, i.e. the bounding box of the input mesh dilated by the offset distance δ. However, for very large voxel resolutions, the size of the data structure can quickly grow above the available memory capacity – even if we apply adaptive refinement.

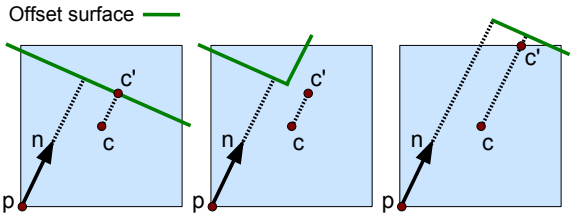

Figure 3: *2D illustration for the computation of cell representatives.* $\mathbf{p}$ *is the minimum distance point,* $\mathbf{n}$ *is the tangent plane normal vector pointing to the corresponding base point,* $\mathbf{c}$ *is the cell-midpoint, and* $\mathbf{c}'$ *is the resulting representative (left). If the local smoothness assumption does not hold,* $\mathbf{c}'$ *is not guaranteed to actually lie on the offset surface (middle). If the representative lies outside the cell (right), we simply clamp the projection to the cell hull. A smoothing procedure will resolve these problems later (Section 6).*

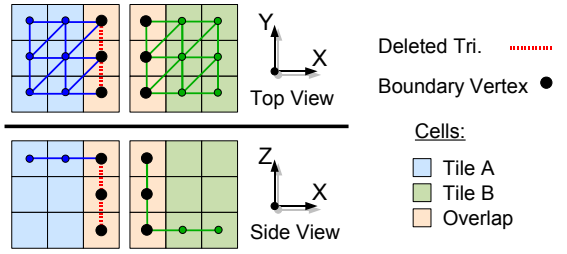

Figure 4: *Removing the overlap between neighboring tiles. The triangles lying in the overlapping area of the tile A are removed while those of tile B remain. The boundary vertices computed on both sides are simply snapped.*

In order to significantly reduce the amount of information that has to be stored simultaneously, we split the bounding box into smaller tiles that can be processed independently (sequentially or in parallel) by applying the algorithm of Section 3.1 and 3.2 to each tile. The critical part then is to guarantee that these independently generated offset meshes can be merged into a proper manifold surface. This can be achieved by defining the tiles such that they overlap by one layer of voxels with their neighbor tiles and then discard those output triangles that have been generated twice (s. Fig.4). This overlap requirement implies that we cannot use simply the cells of an intermediate octree level as tiles since they do not overlap.

If we bound the maximum refinement level for the octree data structure within each tile to $\mathcal{L}$ and use a grid of n^3 tiles, we can effectively work on a $((2^{\mathcal{L}}-1)n+1)^3$ voxel grid. For the identification of the double triangles we do not really need to compare triangles. For each tile we simply discard all

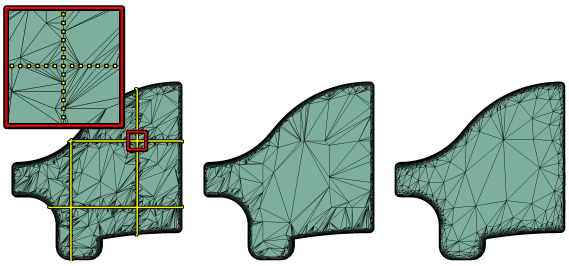

Figure 5: *Volume tiling example. Left: The offset of the fan model after volume tiling with 3^3 tiles (zoomed view of the boundary corner where 4 tiles meet). Middle: after snapping boundaries and decimation. Right: After final smoothing.*

triangles whose three vertices all lie in the right, front, or top layer of voxels. This asymmetric definition guarantees that exactly one copy of each double triangle is removed. One tiling example is shown in Fig.5.

4. Distance computation

The last missing functionality in the rasterization procedure, is the actual distance computation. For each distance primitive, i.e., sphere, cylinder, and prism, we have to compute the minimum and maximum distance with respect to a given cell. Here we can take advantage of several nice properties of distance fields. Moreover, from a certain octree level on, the primitives are guaranteed to be located *outside* the cells through which the offset surface passes. This allows us to apply efficient classification techniques known from polygon clipping in order to determine whether the minimum distance is taken on at a face, edge, or corner of the cell.

For the distance field of a (weakly) convex object, all iso-contours are (weakly) convex, too. Hence, if we restrict the spatial distance field to a planar polygon, the maximum distance value is always obtained at one of the corner vertices. If we apply this observation to the six sides of a cubical cell, it follows that the maximum distance within a cell is always obtained at one of the corners. Since we are processing just spheres, cylinders, and prisms, which are all weakly convex, this observation applies in our case.

For the minimum distance within a cell it is in general not sufficient to check the distance at the corners since the minimum can be obtained in the interior of one of the edges or faces, too. However, since the iso-contours of the distance field to a polygonal mesh can be decomposed into planar, cylindrical, and spherical regions there is usually just a finite number of different relative constellations that needs to be checked.

In the following let C be the cell to be checked. It has eight corners $\mathbf{c}_1,\dots\mathbf{c}_8$, twelve edges $\mathbf{e}_1,\dots\mathbf{e}_{12}$, and six sides $\mathbf{s}_1,\dots\mathbf{s}_6$. Notice that the edges and sides are parallel to the coordinate axes, which makes distance computations significantly easier since some of the vector entries vanish.

4.1. SPHERE distance function

For a given vertex position V we want to estimate minimum and maximum distance of this vertex from the cell C.

For the minimum distance we first have to determine whether this minimum is obtained at a corner, edge, or side of the cell. Let $S_1,\dots S_6$ be the supporting planes of the cell sides $\mathbf{s}_1,\dots\mathbf{s}_6$, oriented such that the cell lies in the intersection of the negative half-spaces. By checking the vertex V with respect to the six supporting planes, we generate a six digit binary number from which we can conclude directly on which corner, edge, or side the minimum distance is obtained. If V happens to lie in the interior of the cell (binary code 000000), the minimum distance is set to zero.

The same binary code can be used to determine, which corners are candidates for the maximum distance. If the minimum distance is obtained at a side of the cell, we have to check the four corners of the opposite side. If the minimum lies on an edge, the candidates for the maximum distance are the corners of the opposite edge. Finally in case the minimum distance occurs at a corner the maximum is obtained at the opposite corner. The binary code 000000 implies that all eight corners have to be checked.

We store the binary code as a vertex attribute and re-use it when the incident edges and faces are processed.

4.2. CYLINDER distance function

An edge E with endpoints $\mathbf{a}$ and $\mathbf{b}$ defines a cylindrical distance field which is valid in the region between the two planes with normal vector $\mathbf{e} = \mathbf{a} - \mathbf{b}$ passing through $\mathbf{a}$ and $\mathbf{b}$ respectively. Hence we first check if the cell intersects this region. Otherwise no further computation is needed since minimum and maximum distance have already been determined correctly based on the edge's endpoints.

In case the cell intersects the cylinder region, we next check if the edge E is intersecting the cell by using the 3D version of the Liang-Barsky line clipping algorithm [FvDFH90]. If this is the case, the minimum distance is set to zero. Otherwise, the binary codes computed and stored for the two endpoints $\mathbf{a}$ and $\mathbf{b}$ can be used to quickly determine the relative constellation of E and C. We multiply the binary codes of $\mathbf{a}$ and $\mathbf{b}$ digit by digit (logical "and"). If the resulting binary code does not vanish we can again restrict the distance computation to one side (including its four boundary edges and corners), one edge (including its endpoints), or one vertex of the cell. From a certain octree level on (i.e. when the cell size is less than the offset distance δ), the edge E is most likely to lie outside of the cell and hence the probability for a constellation, which allows for simplified calculations, is very high.

In order to compute the minimum and maximum distances for the cell, we have to compute distances between E and either some of the sides, edges or corners of the cell.

The distance between E and a corner $\mathbf{c}_i$ can be computed

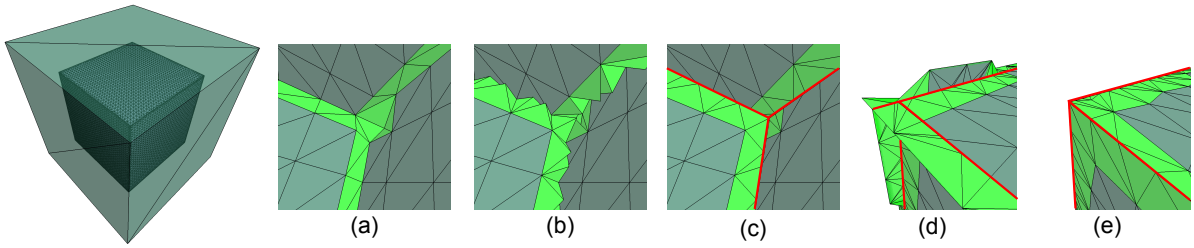

Figure 6: *Feature reconstruction on the inner offset of a cube. In (a) we see one of the corners of the extracted offset surface from inside (feature faces are green). (b) shows the same part after the insertion of feature vertices and (c) after the edge flipping (features now shown in red). As explained in Section 3.2 (see also Fig.3), the cell representatives do not necessarily lie on the offset surface in the vicinity of a non-smooth feature. (d) shows the offset surface from outside. By mesh smoothing, these outliers can be pulled back (see Section 6) to the correct offset surface (e).*

by solving a quadratic equation. If the nearest point on the supporting line of E to $\mathbf{c}_i$ does not lie in the interior of E, we compute the distances between $\mathbf{c}_i$ and the endpoints of E instead [SE03].

While edge-to-corner distances are needed for the minimum and the maximum, edge-to-edge and edge-to-side distances are only necessary to compute the minimum distance.

The distance between E and an edge $\mathbf{e}_j$ is computed by solving a 2×2 system. Again the minimum distance between the respective supporting lines is only valid if the corresponding nearest points are actually lying on E and $\mathbf{e}_j$ respectively. Otherwise, we fall back to the edge-to-vertex or even vertex-to-vertex distance computation.

Finally, the distance between E and a side $\mathbf{s}_k$ is obtained by the minimum of the two endpoint's distances to the supporting plane of $\mathbf{s}_k$. However these distances are only valid if the orthogonal projection of the endpoints lies in the interior of $\mathbf{s}_k$. If one of the vertex-to-plane distances is invalid, we don't have to do any further computations since we can substitute it by the corresponding edge-to-edge distance (computed previously).

Notice that for the sake of simplicity and efficiency we are computing the distances between E and the complete cell C, not only the part of C that falls into the valid cylinder region. Even if this causes redundant computations, it turns out that these computations are less expensive than clipping of the cell C at the two bounding planes and computing the distances to the clipping polygon.

4.3. PRISM distance function

A triangle F and its normal vector span an infinite triangle prism (ITP). We start by checking if the cell C intersects this prism. If this is not the case we can skip any further computation since the distance estimates have already been computed in the spherical or cylindrical distance function. Next we check if F intersects the cell in which case the minimum distance is set to zero.

Again, we use the binary codes stored at the vertices and multiply them digit by digit to obtain a six digit binary code for F. This code determines the relative spatial constellation and allows us to identify the sides, edges, and corners to which the distance function has to be evaluated.

The distance between a corner and the triangle F is computed by the standard procedure described in [SE03]. Within the ITP, the distance field is just the linear distance field to a plane. Hence for polygons the extremal distances are always obtained on the boundary and for edges the extremal distances are obtained at the endpoints. We exploit this observation by computing the intersections of the relevant cell edges $\mathbf{e}_j$ with the sides of the prism and the intersections of the prism edges with the relevant sides of the cell. Minimum and maximum distances are then computed as the minimum and maximum among the point-to-plane distances between the supporting plane of F and the set of points (intersection points and cell corner points).

5. Feature Reconstruction

In this section we explain how our initial offset surface can be improved by adding feature information (see also Fig.6). In an offsetting operation sharp features (shocks) are caused by concave regions when the offset distance is larger than the concave radius of curvature. We reconstruct these features in three steps.

Detecting feature faces The normal in each vertex $\mathbf{v}$ of the reconstructed offset surface (these are the cell representatives computed in Section 3.2), is the corresponding **minNormal** computed in Section 3.1 We define a face to be a feature face if the maximum angle between two of these vertex normals is above a prescribed threshold ϕ. The choice of this threshold is not critical since setting it too low will only cause false positive feature detections which do not compromise the quality of the resulting surface.

Subdivision Each vertex of a feature triangle defines with its normal vector a tangent plane. A good sample point on the sharp feature can therefore be computed by intersecting these tangent planes [KBSS01]. The linear system

$$\begin{pmatrix} n_{0,x} & n_{0,y} & n_{0,z} \\ n_{1,x} & n_{1,y} & n_{1,z} \\ n_{2,x} & n_{2,y} & n_{2,z} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \end{pmatrix}$$

characterizing the intersection point has to be solved by singular value decomposition since along smooth feature curves, the matrix can become singular. In this case the SVD pseudo inverse will compute the least norm solution. This is why we have to shift the voxel center to the origin before we set up the plane equations.

Flipping and alignment The feature samples lie on the feature, but the mesh connectivity does not yet properly represent the feature curve as a polygon of mesh edges. In order to achieve this, we have to flip edges that cross the feature curve [KBSS01]. Hence we make a pass over all mesh edges and flip them if (1) after the flip, this edge connects two feature vertices and (2) the angle between the normal vectors at the two endpoints before the flip is above the threshold ϕ for the feature detection (see Fig.6).

6. Smoothing

The last step in our offset surface computation is a smoothing operation, which is necessary because the computed cell representatives are not guaranteed to lie exactly on the offset surface as shown in Fig.3 and Fig.6(d). Our smoothing operator uses two forces: The relaxation force moves the vertex towards the center of gravity of its 1-ring neighbors and the offset force pulls the feature vertex to the offset surface.

Let $\mathbf{v}$ be a vertex and $\mathbf{v}_1, \dots \mathbf{v}_n$ its one-ring neighborhood. In order to efficiently find the base point $\mathbf{b}$ on M having the minimum distance to $\mathbf{v}$, we have to build a spatial search data structure. Here, we cannot use the octree from the rasterization phase since it represents only a single volume tile while smoothing is applied to the entire object as a final step (see Section 7). Hence, for simplicity and efficiency we build a kd-tree for the complete input mesh M. Then the relaxation force is pulling $\mathbf{v}$ towards $\mathbf{v}'$ while the offset force is pulling $\mathbf{v}$ towards $\mathbf{v}''$:

$$\mathbf{v}' = \frac{1}{n}\sum_i \mathbf{v}_i \qquad \mathbf{v}'' = \mathbf{b} + \delta\frac{\mathbf{v}-\mathbf{b}}{\|\mathbf{v}-\mathbf{b}\|}.$$

The smoothing operator moves each vertex to a weighted average of the two target points, i.e., $\mathbf{v} \leftarrow (1-\alpha)\mathbf{v}' + \alpha\mathbf{v}''$. In this smoothing operator the relaxation force is effectively avoiding triangle flips. Fig.6 and Fig. 7 show the effect of the smoothing operation. In our current implementation we are using $\alpha = 0.5$.

For feature vertices, we only consider 1-ring neighbors which are feature vertices themselves to compute the relaxation force. A feature vertex with just one or more than two feature vertex neighbors is considered a corner vertex and is not smoothed at all.

7. Scalability

In order to be able to use very high voxel resolutions, we already introduced the volume tiling technique in Section 3.3. However, not only the growing size of the octree data structure is critical but also the growing complexity of the output mesh. Hence we need to apply mesh decimation [GGK02]

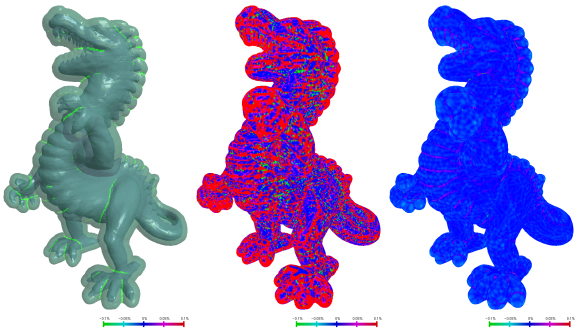

Figure 7: *Left: input model blended with the offset surface. Middle: error visualization on the initially extracted offset surface. Right: error visualization after feature extraction and smoothing. The errors are color-coded on the scale* $[-0.5\%\delta..0..0.5\%\delta]$ *with colors [green..blue..red] and* $\delta = 2\%$ *(of the bounding box diagonal).*

to avoid redundant over-tesselation in flat regions of the offset surface. Obviously we have to guarantee consistency between the surface patches corresponding to neighboring tiles. Hence we block the boundary vertices of each submesh from being decimated. The same applies to features: they are excluded from the decimation to make sure that features are preserved across tiles.

The overall offset generation procedure then is as follows (see Fig.5): The bounding box is split into overlapping tiles. For each tile we perform the hierarchical distance field computation and mesh extraction (Section 3). Then we discard the triangles from the right, front, and top layers (section 3.3). Next we perform the feature *detection* (Section 5) and label all vertices belonging to a feature face as well as all boundary vertices as passive. To the remaining active vertices we apply an incremental mesh decimation scheme like [Hop96].

After we have processed all tiles, we merge the submeshes into one single mesh, which is trivial since we did not change their boundaries. Then we apply the feature reconstruction (subdivision and flipping, see Section 5) and finally apply a concluding decimation step which now also removes former boundary vertices. Features can easily be preserved by enabling edge collapses only between two feature vertices or two non-feature vertices but never between a feature and a non-feature vertex. As a final step we apply our smoothing procedure described in Section 6.

8. Results & Discussion

All our experiments are performed on a commodity PC (AMD64 2.2GHz with 4GB RAM). In Fig.9 we show a selection of offset surfaces for some challenging input meshes created by our method. In the lighthouse example we see that even the finest detail like the antenna and the handrail are preserved when computing the offset surface. With the Part model, which contains many degenerate triangles and incon-

input model	voxel-res.	tiling	offset	total time	offset time	dec. time	max. tile time	#faces	memory peak
Cogwheel(2K△)	60^3	$1^3(64^3)$	2%	1.9s	1.7s	-	-	12K	2MB
Cogwheel(2K△)	121^3	$1^3(128^3)$	2%	3.6s	3s	-	-	66K	4MB
Buddah(1M△)	478^3	$4^3(128^3)$	2%	3100s	2400s	50s	168s	210K	43MB
Fan(13K△)	864^3	$4^3(256^3)$	1.6%	450s	124s	265s	41s	154K	59MB
Fan(13K△)	904^3	$4^3(256^3)$	3.2%	458s	140s	258s	38s	152K	55MB
Lighthouse(3K△)	322^3	$4^3(128^3)$	1.1%	63s	19s	32s	10s	121K	17MB
Lighthouse(3K△)	332^3	$4^3(128^3)$	2.2%	62s	20s	32s	9s	100K	16MB
Part(9K△)	507^3	$4^3(128^3)$	1.3%	109s	56s	39s	15s	150K	17MB
Part(9K△)	2008^3	$4^3(512^3)$	1.3%	1100s	280s	680s	166s	574K	203MB

Table 1: *Timings and memory requirements for the examples shown in Fig.8 and 9. The columns show the effective voxel resultion, the tiling pattern and the offset distance δ. Besides the total computation time we also give the timings for the actual offset computation and the decimation (both included in the total time). The number of faces shows the output mesh complexity for both, inner plus outer offset after merging, feature reconstruction, decimation, and smoothing.*

offset (in %)	1	**2**	4	8	12	16	20
time (in s)	37	**44**	51	61	67	82	96
mem. (in MB)	47	**50**	60	74	90	108	129
input (in K)	5	10	**20**	40	60	80	100
time (in s)	17	28	**44**	74	102	132	167
mem. (in MB)	42	45	**50**	58	66	74	82

Table 2: *Statistics for some experiments with the Dragon model at various offset distances and input mesh resolutions. Fig. 7 shows the results for the test in bold (2%, 20K△). Time and memory, grow proportionally to the input mesh complexity (lower part). For varying offset distances, time and memory depend on the offset surface area (upper part).*

sistencies, we demonstrate the robustness of our method. It can deal with all kinds of artifacts and still produces consistent and water-tight output. We depict inner offsets as well to show how reliable the feature reconstruction works.

Some statistics for the examples from Fig.9 and Fig.8 can be found in Table 1. In order to provide an accurate evaluation, the voxel resolution describes the size of the tight bounding box of the resulting offset surface. The voxel-resolution of the individual tiles is given in the brackets of the "*tiling*"-column. The total running time depends on the input complexity and the voxel resolution. The timings for the two most time consuming steps, namely octree generation and offset mesh decimation, are given in separate columns. The memory column in the table shows the peak of the memory footprint during the computation. Since our volume tiling technique allows for parallel processing of tiles, when using a PC cluster with sufficiently many machines the total running time reduces to the one of the most complex tile. These timings are given in the "*max. tile time*"-column.

In Fig.8 we present an example of a morphological opening operation. For this we compute an inner offset to the original Bunny model (erosion) and then an outer offset to the inner offset (dilation). This computation took 90s and the final mesh has 90K faces. The complexity of the Cogwheel model is comparable to those models used for offsetting by Varadhan et al. [VM04]. The timings in Table 1 show that our method is about one order of magnitude faster. With

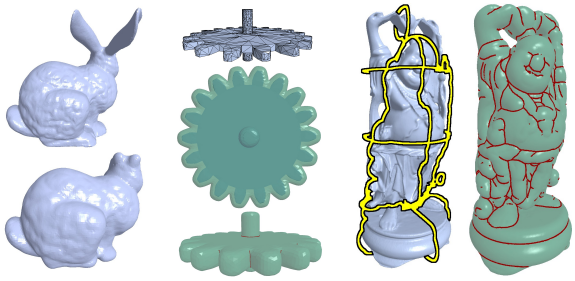

Figure 8: *Left to right: Bunny model (70K) after a morphological opening operation with $\delta = 3\%$ (thin parts like the ears are removed), the Cogwheel with two offset-views ($\delta = 2\%$) and the Buddah with its offset ($\delta = 2\%$).*

the Buddah model (1M faces) we show that our method performs well also for large input meshes.

In Table 2 we show how the offsetting distance and the input complexity affect the memory and running time of our method performed for the Dragon model from Fig.7.

9. Limitations & Future Work

Although our method performs well in general, there are some problems left to solve. The feature extraction is still problematic in strongly concave regions, i.e. if there are too strong (close to π) normal deviations between the corresponding input faces of a feature line. Our feature extraction method could be applied in the context of other methods which are based on volumetric representations and therefore suffer from typical resampling artifacts. Examples are general iso-surface extraction, mesh repair, etc. For many applications only an approximation of the offset surface could be sufficient. Hence, we would also like to find a way to simplify our computations and so enable interactive offsetting at the expense of accuracy.

References

[BM99] BREEN D. E., MAUCH S.: Generating shaded offset surfaces with distance, closest-point and color voumes. In *Proceedings of the International Workshop on Volume Graphics* (1999), pp. 307–320.

[BMW98] BREEN D. E., MAUCH S., WHITAKER R. T.: 3d scan conversion of csg models into distance volumes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization* (New York, NY, USA, 1998), ACM Press, pp. 7–14.

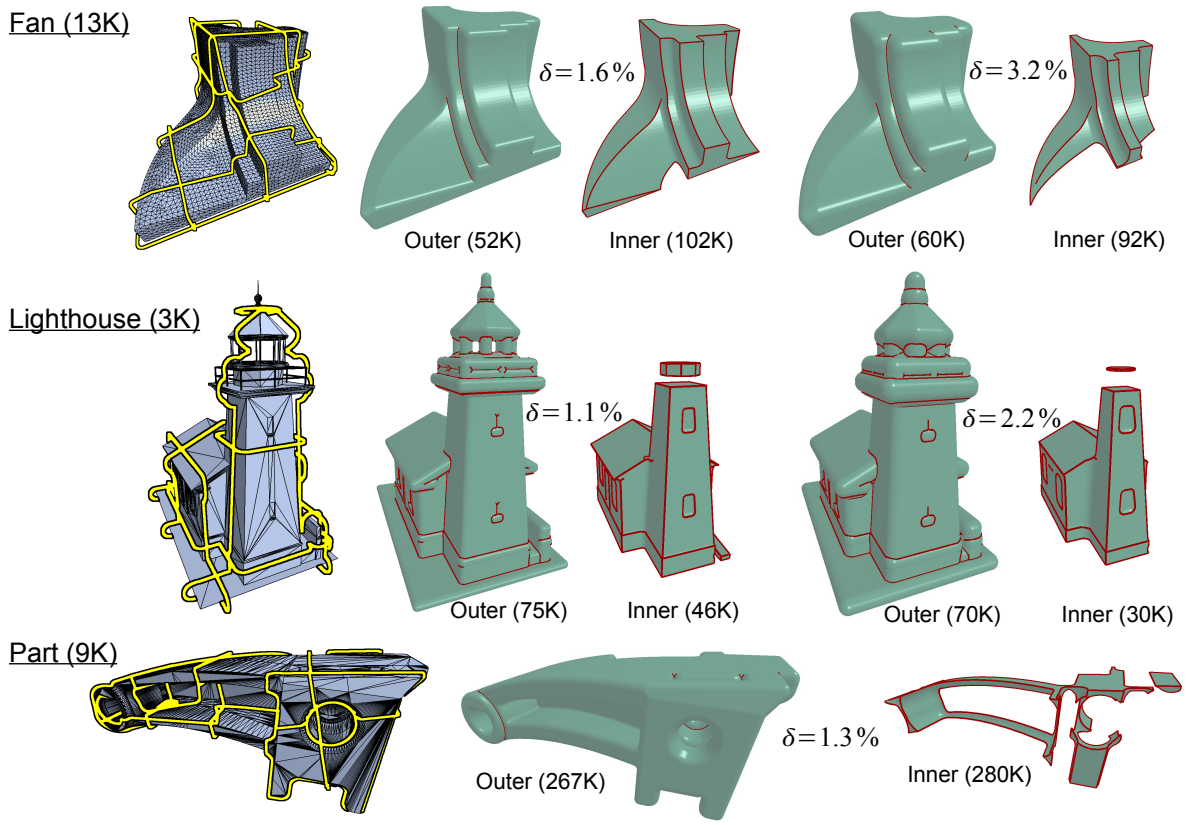

Figure 9: *Offset surfaces generated by our algorithm for a number of challenging technical examples. Since we use the unsigned distance function, both the inner and outer offsets are extracted. Timings and other quantities are given in Table 1 for both parts together. The offset δ is given in percent of the corresponding bounding box diagonal. On the left the tiling boundaries are visualized on the input model.*

[BPK05] BISCHOFF S., PAVIC D., KOBBELT L.: Automatic restoration of polygon models. *ACM Trans. Graph. 24*, 4 (2005), 1332–1352.

[CVM*96] COHEN J., VARSHNEY A., MANOCHA D., TURK G., WEBER H., AGARWAL P., BROOKS F., WRIGHT W.: Simplification envelopes. In *Proc. ACM SIGGRAPH* (1996), pp. 119–128.

[CWRR05a] CHEN Y., WANG H., ROSEN D. W., ROSSIGNAC J.: Filleting and rounding using a point-based method. In *DETC'05 Proceedings* (2005).

[CWRR05b] CHEN Y., WANG H., ROSEN D. W., ROSSIGNAC J.: *A Point-Based Offsetting Method of Polygonal Meshes*. Tech. rep., 2005.

[For95] FORSYTH M.: Shelling and offsetting bodies. In *Proc. of the ACM Symp. on Solid Modeling and Applications* (1995).

[FPRJ00] FRISKEN S. F., PERRY R. N., ROCKWOOD A. P., JONES T. R.: Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proc. of ACM SIGGRAPH* (2000), pp. 249–254.

[FvDFH90] FOLEY J., VAN DAM A., FEINER S., HUGHES J.: *Computer Graphics Principles and Practice*. Addison-Wesley, 1990.

[GGK02] GOTSMAN C., GUMHOLD S., KOBBELT L.: Simplification and compression of 3d-meshes. In *Tutorials on Multiresolution in Geometric Modelling*. Springer, 2002.

[GW01] GONZALEZ R. C., WOODS R. E.: *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[GZ95] GURBUZ A. Z., ZEID I.: Offsetting operations via closed ball approximation. *CAD 27*, 11 (1995), 805–810.

[HC02] HUANG J., CRAWFIS R.: Adaptively represented complete distance fields. *Geometric Modeling for Scientific Visualization* (2002).

[HLC*01] HUANG J., LI Y., CRAWFIS R., LU S. C., LIOU S. Y.: A complete distance field representation. In *VIS '01: Proceedings of the conference on Visualization '01* (2001), pp. 247–254.

[Hop96] HOPPE H.: Progressive meshes. In *ACM SIGGRAPH* (New York, NY, USA, 1996), ACM Press, pp. 99–108.

[JLSW02] JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual contouring of hermite data. In *Proc. of ACM SIGGRAPH* (2002), pp. 339–346.

[KBSS01] KOBBELT L. P., BOTSCH M., SCHWANECKE U., SEIDEL H.-P.: Feature sensitive surface extraction from volume data. In *SIGGRAPH* (2001), pp. 57–66.

[MS00] MCMAINS S., SMITH J.: Layered manufacturing of thin-walled parts. In *ASME Design Engineering Technical Conference, Baltimore, Maryland.* (2000).

[OF02] OSHER S. J., FEDKIW R. P.: *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

[QS03] QU X., STUCKER B.: A 3d surface offset method for stl-format models. *Rapid Prototyping Journal 9*, 3 (2003), 133–141.

[QZS*04] QU H., ZHANG N., SHAO R., KAUFMAN A., MUELLER K.: Feature preserving distance fields. In *VV '04: Symposium on Volume Visualization and Graphics* (2004), pp. 39–46.

[RR85] ROSSIGNAC J. R., REQUICHA A. A. G.: Offsetting operations in solid modelling. *Comput. Aided Geom. Des. 3*, 2 (1985), 129–148.

[SE03] SCHNEIDER P. J., EBERLY D. H.: *Geometric Tools for Computer Graphics*. Morgan Kaufmann Publishers, 2003.

[Ser83] SERRA J.: *Image Analysis and Mathematical Morphology*. Academic Press, Inc., 1983.

[Set99] SETHIAN J. A.: *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.

[VKKM03] VARADHAN G., KRISHNAN S., KIM Y. J., MANOCHA D.: Feature-sensitive subdivision and isosurface reconstruction. In *Proc. of the IEEE Vis.* (2003).

[VKSM04] VARADHAN G., KRISHNAN S., SRIRAM T., MANOCHA D.: Topology preserving surface extraction using adaptive subdivision. In *Proc. of the Eurographics Symp. on Geometry Processing* (2004), pp. 235–244.

[VM04] VARADHAN G., MANOCHA D.: Accurate minkowski sum approximation of polyhedral models. In *Proc. of Pacific Graphics Conf.* (2004), pp. 392–401.