Implicit surfaces appear in many applications, including medical imaging, molecular modeling, computer aided design, computer graphics, and finite element analysis. A field function \( f(x, y, z) \) defines an implicit surface and assigns a scalar value to each point in space. The implicit surface consists of those points where the field function takes a specific value \( c \). In other words, the surface is \( \{ (x, y, z) | f(x, y, z) = c \} \).

In molecular modeling, for example, the electron density takes a specific value at the Van der Waals surface of the molecule.\(^1\) This surface estimates the structure of an unknown molecule whose electron density has been sampled (see Figure 1). In medical imaging, for example, a computed tomography (CT) scan produces a 3D array of sampled density values from a patient. Anatomical features are distinguished by their different densities and can be represented with implicit surfaces. In computer-aided design, for example, an offset surface can provide a smooth blend between two components meeting at a sharp angle.

Despite their many advantages, implicit surfaces are difficult to render efficiently. Today’s real-time graphics systems are heavily optimized for rendering triangles, so an implicit surface should be converted to a mesh of triangles before rendering.

Our algorithm polygonalizes an implicit surface. The algorithm generates a mesh of close-to-equilateral triangles with sizes dependent on the local surface curvature. We assume that the implicit surface is connected and \( G^1 \) is smooth (that is, the tangent plane varies continuously over the surface). The algorithm requires an evaluator for the implicit function defined at all points in space, an evaluator for the function gradient defined at points near the surface, and a bounding box around the surface.

The output of the algorithm is good for applications requiring a “well-behaved” triangulation, such as rendering systems and finite element partial differential equation (PDE) solvers. For rendering systems, curvature-dependent triangle sizing results in an accurate surface and silhouette (see the middle of the peanut in Figures 2a and 2b), while at the same time minimizing the number of triangles, resulting in faster rendering. For finite element methods, the near-equilateral triangles reduce the instability of the solution, and the curvature dependence keeps discretization errors low while minimizing the number of triangle elements, leading to a faster solution.

The algorithm

The algorithm operates in two phases. In the growing phase, a seed triangle (which forms the initial polygonization) is computed. The polygonization incrementally grows triangles from its edges and extends. Each new triangle is sized according to the local curvature, and a triangle isn’t added if it comes too close to an already existing triangle. At the end of this phase, the polygonization connects a region with long, narrow gaps between its branches.

In the filling phase, the gap is subdivided into small pieces by finding bridges that cross the gap. These bridges are good edges in the final triangulation. They separate the gap into smaller, more manageable pieces. Each smaller piece is triangulated with a set of heuristics.

Preliminaries

Before discussing the algorithm in detail, we offer the following concepts:

- Triangles are sized according to a user-defined parameter \( p \), which is the desired ratio of triangle edge length to local radius of curvature.
- Two triangles are said to overlap if part of one, when projected onto the plane of the other, lies inside the other. For nonoverlapping triangles, the distance between them is the minimum distance between a vertex of one and an edge of the other.
The radius of curvature at a point \( x \) is estimated by computing the radius of curvature of several geodesics that pass through \( x \) and taking the minimum. Geodesics are assumed to locally lie in a plane that passes through \( x \) and the surface normal at \( x \), denoted \( n_x \). To compute one such radius of curvature, the normal \( n_y \) at a point \( y \) close to \( x \) is first computed. Let \( q \) be the angle between \( n_x \) and \( n_y \), and let \( d \) be the distance between \( x \) and \( y \). Then the radius of curvature is estimated as

\[
R(x) = \frac{d}{2 \sin \left( \frac{\theta}{2} \right)}
\]

Growing phase

We first sample the bounding volume until we find one point inside the surface and one point outside the surface. We then find a surface point through repeated bisections. This point locates a seed triangle whose vertices lie on the implicit surface, and whose edge lengths are a fraction \( \rho \) of the local surface curvature.

The seed triangle’s edges are placed into an active edge list. Growing proceeds by iterating over this list and applying isosceles triangle growing and ear cutting operations to each edge (as detailed in the following sections). The growing phase terminates when no operation applies to any edge in the list. Figure 3 shows sample triangulations at the end of the growing phase.
Related Work

The new algorithm falls into the class of continuation methods, which incrementally extend a polygonization across the implicit surface. Following Bloomenthal, these methods are divided into piecewise linear and predictor-corrector classes.

Piecewise linear continuation methods divide space into discrete cells (typically cubes or tetrahedra) and polygonize each cell individually. A cell is considered for polygonization only if it’s adjacent to another cell that already contains part of the surface. A cell’s polygonization is determined from a lookup table indexed by signs of the implicit function at the cell’s vertices. Cells may be adaptively sized according to local surface features.

These methods are fast, but may result in polygonizations that contain high-aspect-ratio triangles and tiny polygons, such as those shown in Figure 2b. A number of post-processing methods eliminate undesirable features.

Unlike these methods, the new algorithm doesn’t require a separate postprocessing step to produce a high-quality triangulation. When each triangle is created, it’s sized according to the local surface curvature and is made close-to-equilateral, as shown in Figure 2a.

Adaptive continuation methods attempt to join polygons that lie in adjacent cells of different resolutions. The new algorithm avoids this difficulty by allowing the triangle sizes to vary continuously as the triangulation is extended.

The new algorithm, however, is slower than the piecewise linear methods. It spends considerable time computing local surface curvature in order to generate triangles of the appropriate size. (The curvature calculation requires many calls to the implicit function evaluator, since we assume that the second derivative isn’t directly available.) This time isn’t spent with the piecewise linear methods, most of which only evaluate the implicit function at cell vertices.

Predictor-corrector continuation methods extend the polygonization by generating new vertices on the current polygonization’s border. These vertices initially lie in the tangent plane at the border (the predicted position) and are subsequently settled onto the implicit surface (the corrected position). New polygons are added to join the vertex to the current polygonization. Alternatively, a disk centered on a boundary point may be created, projected onto the surface, and merged with the current triangulation.

The new algorithm is a predictor-corrector method that successfully addresses the difficulties encountered with 2D surfaces. In particular, the new algorithm avoids overlapping triangles, which can occur if two separate polygonizing branches converge as they are extending. The new algorithm also fills in the narrow gap between adjacent branches with well-shaped triangles sized according to the local surface curvature.

Another approach creates triangles of roughly uniform shape and size, which are locally Delaunay. (A triangle is locally Delaunay if its smallest circumscribing sphere doesn’t contain another point of the triangulation that has the same surface orientation.) This approach handles open two-manifolds, while the new algorithm assumes that the surface is closed. However, this approach doesn’t size its triangles according to the local curvature, which may lead to rendering artifacts, particularly on the silhouette.

The new algorithm resembles an algorithm of Hartmann, which iteratively performs one of two operations. Either the current mesh boundary adds triangles (like our growing phase) or the closest pair of nonadjacent vertices on the mesh boundary are joined (like our bridging operation). Both algorithms produce close-to-equilateral triangles, but Hartmann’s algorithm doesn’t appear to make the triangle size dependent on surface curvature. The new algorithm uses a large set of heuristics (see the “Filling phase” section) which help to produce good triangles in those areas where the triangle size changes rapidly.

Full-lattice methods assume that a full lattice of measured function values is available. These methods correctly treat surfaces of multiple connected components, while the new algorithm (and other continuation methods) require a seed polygon in each connected component.

Particle-based methods settle oriented particles to the implicit surface and maintain them on the surface during editing operations. Particles require that the point set is triangulated in a postprocess using a local Delaunay triangulation or alpha shapes.

The triangulations produced by particle systems have the Delaunay-like property where triangles are close to equilateral, and particle-based methods can be modified and adapt to the particle density of local surface curvature. These methods have vertex distributions qualitatively similar to those of the new algorithm. However, the particle-based algorithm (described in the “Experimental results” section) takes much longer to execute than the...
new algorithm, probably because every particle continues to be processed as long as at least one particle hasn’t reached equilibrium.

The pretessellation method considers implicit surfaces defined as a combination of certain primitive surfaces, each of which has a known surface tessellation. Each primitive’s surface tessellation is terminated where it enters a blending area with another primitive, then the corresponding boundary curves of adjacent primitives are identified, and finally a mesh is constructed in the gap between corresponding curves.

Like the pretessellation method, the new algorithm builds a mesh over part of the surface and subsequently fills the gap, which is left uncovered. However, the new algorithm handles arbitrary implicit functions, which makes the structure of the gap much more complicated. With the pretessellation method, the gap consists of relatively straight sections separating pairs of primitives, and each section can be meshed independently. The new algorithm must handle general gaps, as shown in Figure 1. It’s likely, however, that the pretessellation method is much faster for the surfaces that it handles.

The shrinkwrap method builds a surrounding polygonized mesh and incrementally shrinks this mesh onto the implicit surface. The mesh vertices move along the integral lines of the implicit function and the mesh is refined as necessary. The new algorithm polygonizes some surfaces that we believe the shrinkwrap method can’t polygonize.

References

angle with at least two edges whose lengths are appropriate for the local surface curvature.

The isosceles triangle growing operation performs two tests. The first test checks that each new edge makes an angle of at least 45 degrees with its neighbor in the old mesh. This ensures that later triangles will be close to equilateral.

The second test checks that the new triangle, T, doesn’t approach existing triangles too closely. This ensures that the growing phase will not produce a gap
Settling a Point

Many places in the algorithm use a procedure to settle a point \( v \) onto the implicit surface, assuming that \( v \) is already near the surface. To do so, a point \( v' \) on the other side of the surface is determined. Repeated bisection of the interval between \( v \) and \( v' \) yields a point on the surface. To determine \( v' \), the vector

\[
\hat{n} = -\frac{f(v)\nabla f(v)}{\|f(v)\nabla f(v)\|}
\]

which points from \( v \) to the surface is computed. Then \( f \) is sampled at increasing distances from \( v \) in the direction of \( \hat{n} \) until a point \( v' \) is found such that the sign of \( f(v') \) is opposite that of \( f(v) \).

Ear cutting. Ear cutting is applied when two adjacent active edges have an external angle smaller than 70 degrees. The three vertices on these two edges define a new triangle. Because of the second test, a triangle from another branch can’t intrude into the area between these three vertices. A value of 70 degrees was chosen to allow somewhat nonequilateral triangles. For angles larger than 70 degrees, the heuristics in the next section are more likely to produce a good local triangulation.

Filling phase

Once the growing phase terminates, a connected polygonal gap remains to be triangulated. The filling phase starts by associating each vertex of the gap with its closest neighbor vertex. The closest neighbor relationship is defined carefully to make the line segment between a vertex and its closest neighbor a good segment in the final triangulation.

Closest neighbors and bridges. For a vertex \( v \) on the gap, let \( N_v \) be the normal to the implicit surface at \( v \), let \( v_1 \) be the vertex preceding \( v \) on the boundary of the gap, and let \( v_2 \) be the vertex succeeding \( v \) (vertices are ordered counterclockwise around the gap, as seen from above the surface).

Every vertex \( v \) on the gap has two associated planes, \( P^1_v \) and \( P^2_v \). \( P^1_v \) embeds \( v, v_1, \) and \( N_v \), and is oriented such that the normal to the plane points into the gap. \( P^2_v \) is defined similarly except that it embeds \( v_2 \) instead of \( v_1 \) (see Figure 4).
A vertex \( u \) is above \( P_1 \), if it's contained in the half-space where the normal of \( P_1 \) points, and is farther from \( P_1 \) than one-tenth the distance between \( v \) and \( v_1 \). The definition of “\( u \) is above \( P_2 \)” is similar. The value of one-tenth was chosen to avoid extremely nonequilateral triangles, which would occur if, for example, the triangle \( v \rightarrow v_1 \rightarrow w \) was formed with a vertex \( w \) that was closer to \( P_1 \) than one-tenth the distance between \( v \) and \( v_1 \).

A vertex \( v \) on the gap is convex if the interior angle (that is, inside the gap polygon) made with \( v_1 \) and \( v_2 \) is less than \( \pi \) and concave otherwise.

Every vertex \( v \) has an associated set of neighbors, which consists of other vertices on the gap. If \( v \) is convex, its neighbors are vertices above \( P_1 \) and \( P_2 \), and if it is concave, its neighbors are above at least one of \( P_1 \) and \( P_2 \).

The closest neighbor \( u \) to a vertex \( v \) is the neighbor of \( v \) with the shortest Euclidean distance to \( v \). We denote this relationship as \( v \rightarrow u \). Note that the closest neighbor may not exist and that this relationship isn't necessarily commutative.

A pair of vertices \( v_1 \) and \( v_2 \) is a bridge if \( v_1 \rightarrow v_2 \) and \( v_2 \rightarrow v_1 \). We denote a bridge as \( v_1 \leftrightarrow v_2 \). See Figure 5.

To compute the nearest neighbors, all gap vertices are inserted into an octree. For each vertex \( v \), a radius \( r \) is initialized to twice the distance between \( v \) and its predecessor. The octree is queried for vertices within distance \( r \) of \( v \). If this set contains neighbors, the nearest of these is selected. Otherwise, the radius is doubled and the procedure is repeated.

Filling the gap. We define a gap as a simple polygon (on the implicit surface) for which each vertex stores its nearest neighbor. An initial gap is created from the polygon produced by the growing phase. This initial gap is placed into an otherwise empty gap queue.

The algorithm starts by removing the first gap from the queue. Heuristics are applied to add triangles to this gap: If a gap splits into several disjoint gaps (because of a heuristic), these gaps are placed at the tail of the queue. If a bridge \( u \leftrightarrow v \) is longer than 150 percent of the optimal local edge length, the bridge is split by adding a vertex at the midpoint of \( u \) and \( v \), and settling this point to the surface. The value of 150 percent was chosen because it's halfway between one edge of the correct length (100 percent) and two edges of the correct length (200 percent).

3. X-filling. An X-sequence (see Figure 6, next page) is a sequence \( S = \{v_1, v_2, v_3, v_4\} \) of adjacent vertices with the properties: (1) \( v_3 \rightarrow v_4 \), (2) \( v_1 \rightarrow v_2 \), and (3) neither \( v_2 \) nor \( v_3 \) is the closest neighbor of any vertex not in \( S \).

If \( G \) contains an X-sequence, \( \{v_1, v_2, v_3, v_4\} \), then the distance between \( v_1 \) and \( v_4 \) is checked against the local optimal edge length. If the distance between the vertices exceeds 150 percent of optimal, a vertex \( v \) is generated on the midpoint between \( v_1 \) and \( v_4 \). The X is then filled with either two or three triangles.

4. Ear filling. An ear is a sequence \( \{v_1, v_2, v_3\} \) of adjacent vertices such that \( v_1 \leftrightarrow v_2 \) and no vertex of \( G \) has \( v_2 \) as its closest neighbor. If \( G \) contains an ear, \( \{v_1, v_2, v_3\} \), then the distance between \( v_1 \) and \( v_3 \) is split if it exceeds 150 percent of the optimal local edge length. The ear is then filled with one or two triangles.

5. Convex polygon filling. If every vertex of \( G \) is convex, a new vertex is placed at the average of the vertices of \( G \) and is settled to the surface. A fan of triangles is created around the new vertex. \( G \) has been filled and isn't returned to the queue.

6. Relaxed ear filling. A relaxed ear, \( \{v_1, v_2, v_3\} \), has the same properties as an ear except that either \( v_1 \leftrightarrow v_3 \) or \( v_1 \rightarrow v_3 \) is acceptable (whereas an ear has \( v_1 \leftrightarrow v_3 \)). The relaxed ear filling procedure is otherwise the same as ear filling.
6 Gap Xs filled with and without the addition of another vertex. Dashed lines indicate the nearest neighbor relationships.

7. Concave vertex bisection. If all of the preceding heuristics fail, \( G \) must contain at least one concave vertex. The vertex \( v_1 \) that has the largest interior angle is selected from \( G \), and a corresponding vertex \( v_2 \) in the gap is found such that the line segment \( (v_1, v_2) \) most closely bisects the interior angle at \( v_1 \). \( G \) is split into two components along this line segment.

If the distance between these vertices is more than 150 percent of the optimal local edge length, the edge is split, settling the midpoint to the surface. The two new gaps are placed at the tail of the queue.

After the gap queue becomes empty, each edge which separates a pair of adjacent triangles is flipped (so that it joins the other two vertices of the two adjacent triangles) if this results in a better local triangulation. To avoid a cascade of flips, each edge is considered only once. Typically, between zero and one percent of edges flip. Figure 7 shows a triangulation after the filling phase.

A final improvement could be made by applying Laplacian smoothing (although we didn’t implement this): Each vertex is moved to the geometric center of its adjacent vertices if and only if this results in an improvement to the local triangulation.

Experimental results

Figure 8 shows several meshes that our algorithm produced. Two other polygonizers were implemented for

8 Meshes produced by the new algorithm: (a) cube, (b) pretzel, (c) spiral, (d) cone, and (e) torus.
comparison: Bloomenthal’s cell-based polygonizer and Witkin and Heckbert’s particle-based modeler. We modified these to make particle density proportional to the local surface curvature.

**Triangle quality**

The new algorithm consistently produces close to equilateral triangles, as shown in Figure 9. The particle-based polygonizer produces somewhat less than equilateral triangles, and the cell-based polygonizer produces fairly poor triangles.

The new algorithm also produces edges whose edge-length to surface-curvature ratios are tightly clustered around the desired ratio set by the user, as shown in Figure 10. The particle-based polygonizer had a qualitatively similar distribution of ratios.

**Execution times**

Table 1 reports the execution times and shows the following:

### Table 1. Execution results on a 500-MHz Pentium PC. These tests used a smaller setting of $\rho$ than Figures 2 and 8.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time (seconds)</th>
<th>Triangles per Triangle</th>
<th>Settling per Triangle</th>
<th>Function Evaluations per Settling</th>
<th>Function Evaluations per Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peanut</td>
<td>10</td>
<td>6,546</td>
<td>5.7</td>
<td>6.2</td>
<td>35.3</td>
</tr>
<tr>
<td>Spiral</td>
<td>102</td>
<td>18,072</td>
<td>5.6</td>
<td>9.0</td>
<td>50.3</td>
</tr>
<tr>
<td>Cube</td>
<td>190</td>
<td>56,292</td>
<td>5.6</td>
<td>4.8</td>
<td>27.1</td>
</tr>
<tr>
<td>Pretzel</td>
<td>587</td>
<td>40,924</td>
<td>5.7</td>
<td>5.8</td>
<td>33.3</td>
</tr>
</tbody>
</table>

(a) Peanut model with (a) the new algorithm, (b) a particle-based algorithm, and (c) a cell-based algorithm. The distributions are shown for the ratio of shortest edge length to longest edge length of each triangle. Distributions for the other objects were qualitatively similar. A ratio of 1.0 corresponds to an equilateral triangle.

(b) Cube with the new algorithm. (b) Pretzel with the new algorithm. (c) Peanut with the particle-based algorithm. The distribution of actual edge-length to surface-curvature ratios, with a desired ratio of $\rho = 0.2$ set by the user. Distributions for other objects were qualitatively similar.
The new implicit surface polygonizer produces close-to-equilateral, curvature-dependent triangulations of $G^1$ surfaces.

We use it when the quality of triangulation is important.

1. a point is settled to the surface about 5.6 times per triangle of the final mesh;
2. between 4.8 and 9.0 implicit function evaluations are required for each settling; and
3. between 27 and 50 function evaluations are required for each triangle of the final mesh.

The new algorithm is typically 20 times slower than the cell-based algorithm, but five times faster than the curvature-dependent, particle-based algorithm. The particle-based algorithm processes every particle at each iteration, but the new algorithm creates a new triangle (or triangles), which thereafter remains fixed.

The new algorithm is fast enough for any application that builds the polygonization in an offline, preprocessing step. The large number of implicit function evaluations per triangle is principally due to curvature evaluations (70 percent of all implicit function evaluations are used for this). While we could have evaluated the curvature directly for some models, this isn’t possible for general implicit functions represented as “black boxes,” so we decided against this.

**Failure modes**

Accurate curvature measurement is critical to the triangle growing phase. If the curvature isn’t measured accurately, small triangles may be placed close to large ones and this situation will be handled badly by the filling phase, which implicitly expects similar triangle sizes on opposite sides of a narrow gap.

The same problem occurs if the rate of change of curvature is high or if the surface is not $G^1$. In this case, there are almost flat areas adjacent to highly curved areas and the growing phase produces large triangles close to small triangles. Again, the filling phase has difficulty in this situation. The pretzel and peanut are particularly stressful tests since they have areas in which the curvature changes very rapidly (see Figure 7a for an example).

We do have a possible fix which we haven’t implemented yet. The situation is detected when the nearest neighbor of a vertex $v$ is adjacent to a triangle sized quite differently than a triangle adjacent to $v$. In this case, subdivide each large triangle into four smaller triangles by adding a vertex at the midpoint of each triangle edge and joining the new vertices (care must be taken to repair any T vertices that occur). Next, attempt to grow triangles from the new, subdivided edges that border the gap. Finally, continue with the gap filling phase, which may involve further subdivisions of large triangles.

If the peanut model is changed so that it looks more like two spheres connected by a thin strand, the area on one sphere from which the strand emanates can be triangulated improperly during the triangle growing phase. When this happens, a triangle is created that covers this area, so the strand and the other sphere are never reached. Note that cell- and particle-based tilers following the surface instead of sampling all space can have the same problems (but a topology-guaranteeing polygonizer wouldn’t).

**Details of the particle-based algorithm**

For comparison, we implemented the Witkin-Heckbert particle-based modeler. This modeler centers a Gaussian repulsion field around each particle, and particles move according to the forces exerted between them and neighboring particles. A spatial grid was used to efficiently determine the neighboring particles. Particles are created automatically in sparsely populated areas and are deleted automatically in densely populated areas. Once the particles achieve equilibrium (that is, the maximum particle velocity falls below a threshold), triangulation is possible. We built the triangulation by computing alpha shapes at various resolutions and stitching together the resulting triangle meshes. We didn’t count this step in the execution times of the particle-based method.

The modeler was modified to make the standard deviation of the Gaussian field around a particle proportional to the surface’s local radius of curvature. Despite this, particles in dense areas would repel particles in sparse areas, simply because more particles exist in the dense areas pushing on the particles in the sparse areas. As a result, particles migrated from high curvature areas into adjacent low curvature areas and more particles were created continually in the high curvature areas, replacing those that left. To prevent migration, we gave more weight to the forces exerted by particles with larger Gaussian repulsion fields (that is, particles in the sparser areas). Specifically, the force exerted by particle $j$ on particle $i$ was weighted by the apparent angle, as seen from $i$, of a disk centered at $j$ with a radius equal to the standard deviation of $j$’s Gaussian repulsion field.

Adding a Laplacian smoothing step after the particles reach equilibrium might improve the output of the particle-based modeler. Also, Paul Heckbert (via a personal communication) suggested using an elliptical repulsion field and having the axes of the ellipse aligned with the directions of minimum and maximum curvature.

**Conclusions and future work**

The new implicit surface polygonizer produces close-to-equilateral, curvature-dependent triangulations of $G^1$ surfaces. We use it when the quality of triangulation is important—for example, with renderers and finite element solvers. This new polygonizer, however, is slower than the classical cell-based polygonizer, so we probably wouldn’t use it for interactive applications, such as surface editing.

The algorithm could be extended to include non-manifold and non-$G^1$ surfaces. We would have to identify sharp edges on the surface and generate triangles around them. We would join the triangles to the rest of
the mesh in the gap filling phase. We would identify sharp edges in the growing phase by detecting high-surface curvature and by following the line of maximum curvature along the surface.

We can make the algorithm much faster for surfaces where we compute the curvature analytically. This is often the case, especially for surfaces constructed from a limited set of primitives. In these cases, the algorithm directly evaluates the curvature instead of using its slow numerical computation.

Finally, we could use the gap filling technique in other applications, such as building models from laser range data. When making an object’s model, we require several data sets to cover the object from all sides. We triangulate these data sets and stitch them together, using the gap filling technique to perform the stitching.

**Acknowledgments**

We wish to thank Paul Bourke of the Swinburne University of Technology for the image of the human corTEX and Bhushan Nagar and James Rini of the University of Toronto for the snake venom data. We also want to thank the anonymous reviewers for their many helpful suggestions.

**References**


Tasso Karkanis received his BSc in physics from the University of Waterloo in 1995 and his MSc in computer graphics from the University of Toronto in 2000. He is currently employed at Alias|Wavefront in Toronto, Canada.

James Stewart is an associate professor in the Dynamic Graphics Project of the Department of Computer Science at the University of Toronto, Canada. His research interests include visibility problems, terrain rendering, cloth rendering, and acceleration techniques that exploit graphics hardware. More information appears at www.dgp.toronto.edu/~jstewart.

Readers may contact Karkanis at Alias|Wavefront, 210 King St. E., Toronto, Ontario, Canada, M5A1J7, email tkarkanis@aw.sgi.com.