GPU-based Tolerance Volumes for Mesh Processing

Mario Botsch David Bommes Christoph Vogel Leif Kobbelt
Computer Graphics Group
RWTH Aachen University

Abstract

In an increasing number of applications triangle meshes represent a flexible and efficient alternative to traditional NURBS-based surface representations. Especially in engineering applications it is crucial to guarantee that a prescribed approximation tolerance to a given reference geometry is respected for any combination of geometric algorithms that are applied when processing a triangle mesh.

We propose a simple and generic method for computing the distance of a given polygonal mesh to the reference surface, based on a linear approximation of its signed distance field. Exploiting the hardware acceleration of modern GPUs allows us to perform up to 3M triangle checks per second, enabling real-time distance evaluations even for complex geometries. An additional feature of our approach is the accurate high-quality distance visualization of dynamically changing meshes at a rate of 15M triangles per second.

Due to its generality, the presented approach can be used to enhance any mesh processing method by global error control, guaranteeing the resulting mesh to stay within a prescribed error tolerance. The application examples that we present include mesh decimation, mesh smoothing and freeform mesh deformation.

1. Introduction

In most computer graphics applications, triangle meshes are the standard surface representation, since they offer high flexibility, as well as very efficient computing and hardware accelerated rendering. In recent years, triangle meshes also gained increasing attention in the field of engineering applications, starting to assist or even replace classical NURBS based systems in many areas.

For this type of applications an approximation tolerance always has to be guaranteed, for instance to ensure that the results of a numerical simulation are meaningful. As a consequence, it is crucial to provide an *exact* or at least conservative *global error* bound for all algorithms that are to be applied to the mesh before it is used in a simulation.

As the input meshes are in most cases generated either by tesselating CAD surfaces or by scanning and reverse-engineering a physical prototype, a typical mesh pre-processing session includes geometric optimization (de-noising, fairing) and topological optimization (simplification, re-meshing). As the exact result of these optimizations is often hard to predict, they are usually applied repeatedly to (regions of) the input mesh in any order.

Although each of these mesh processing algorithms may provide some kind of error bound on its own, these individual errors can accumulate during multiple optimization loops. In order to prevent this, a global approximation error to the initial reference surface has to be taken into account. To allow for greatest flexibility this global error measure should be independent of the individual algorithms to be applied to the mesh.

In the context of mesh decimation, Klein et al. [9] proposed to compute the approximation error between an original mesh and its decimated version using the two-sided Hausdorff distance. Although this distance represents the exact deviation between two surfaces, it is computationally too expensive for complex meshes.

When decimating densely sampled or 3D-scanned input meshes, Kobbelt et al. [11] argue that, like in the scattered data interpolation setup, it is sufficient to consider the one-sided Hausdorff distance from the original vertices to the decimated mesh only. This error measure can be implemented quite efficiently if the underlying atomic simplification operator is the half-edge collapse. In this case all vertices of the current mesh coincide with original vertices, such that only the distances from the already removed original vertices have to be taken into account. For more general mesh processing algorithms, however, this optimization technique is not applicable, since, e.g., after one mesh smoothing iteration all vertices have been relocated, such that the distances from *all* the original vertices have to be computed.

An alternative and very intuitive global error measure is provided by *simplification envelopes* [3] that guarantee the mesh to stay within a prescribed tolerance volume around the initial reference mesh. In the original paper polygonal meshes are used for constructing the simplification envelope by offsetting the reference mesh in positive and negative normal direction and for performing the inside test for a given candidate triangle. Both problems are hard to solve robustly using triangle meshes, resulting in an algorithmically and computationally very complex method.

Comparing the strengths and drawbacks of explicit and implicit surface representations [10], the latter ones are clearly preferable for the required inside tests of a given tolerance volume. Zelinka and Garland [19] therefore proposed to discretize the characteristic function of the tolerance volume into a uniform binary *permission grid*. For querying a candidate triangle it is rasterized into the grid and tested to pass only through "valid" grid cells that lie completely inside the tolerance volume. However, the resulting piecewise constant approximation suffers from aliasing artifacts, requiring a rather fine resolution of the grid, that, in turn, is limited by the main memory.

A more memory efficient representation was proposed by Frisken et al. [6], using an adaptively sampled piecewise linear approximation of the signed distance field. An also piecewise linear, but C^{-1} approximation of the distance field was shown to lead to a further reduction of memory consumption [18]. Although these two approaches consume significantly fewer memory, testing whether a given triangle lies within an approximation tolerance gets more complicated.

We propose an approach that can be categorized to lie between permission grids and the latter two methods. We use a regularly sampled piecewise linear approximation to the signed distance function of the reference mesh, such that, compared to permission grids, the better approximation properties enable us to work with coarser grid resolutions. Although the distance test for a given triangle is more complicated than in [19], it is much simpler compared to [6, 18], since we use a regular sampling.

To check a given candidate triangle the distance values in its interior have to be determined by linearly interpolating the distance values stored at the sampled grid points. However, this tri-linear interpolation task is exactly what texture units of graphics hardware have been optimized for. By representing the piecewise linear distance volume as a 3D texture we are able to exploit the hardware acceleration of modern GPUs. Testing whether a given triangle lies within the tolerance volume then basically amounts to rendering it using the 3D distance texture — that is tri-linearly interpolated by the texture unit of the GPU.

An obvious application of our approach is the accurate and efficient visualization of the approximation error between two meshes by color-coding the respective per-pixel distance values using high-quality post-classification methods known from direct volume rendering [13]. In comparison with a 2D texture based error visualization like in [2], we do not have to pre-compute a per-triangle error texture, but exploit the 3D texturing hardware instead. As a result, we can visualize the distance of a dynamically changing mesh to a reference surface at a rate of 15M triangles/sec.

We first present an efficient method for initially computing a 3D distance grid in Sec. 2 and show how to use it as 3D distance texture in Sec. 3. Sec. 4 describes the implementation of a generic distance check for a given triangle on the GPU. Using these ingredients our method can be encapsulated into an easy-to-use module for distance checks that can be incorporated into any mesh processing algorithm. In Sec. 5 we show application examples for mesh decimation, mesh smoothing and mesh deformation.

2. Distance Field Generation

Given an initial reference surface represented by a triangle mesh, we compute a piecewise linear approximation of its signed distance field by sampling it at the nodes of a regular 3D grid.

In order to efficiently compute distance values at the grid nodes we use *fast marching* methods [16], as also proposed in [12]. We first have to initialize the marching process by computing the distance values in the immediate vicinity of the reference mesh. For each triangle we consider all grid nodes lying in the triangle's slightly enlarged bounding box and compute the exact distances of these nodes to the current triangle. As the bounding boxes of neighboring triangles overlap, we may compute several distance values for a grid node. In this case we simply store the minimal (absolute) value.

Since we want to approximate a *signed* distance field, we have to determine whether a grid node lies inside or outside the reference mesh for each distance computation. This test requires to consider the angle between the vector from the grid node $\mathbf{p}$ to its closest point on the triangle $\mathbf{c}$ and the normal $\mathbf{n}(\mathbf{c})$ at this point, i.e., $\mathbf{p}$ is defined to be inside iff $(\mathbf{p}-\mathbf{c})^T\mathbf{n}(\mathbf{c}) < 0$.

The robustness and reliability of this inside test strongly depends on the way by which the interpolated normal $\mathbf{n}(\mathbf{c})$ is computed. Using barycentric normal interpolation within the triangle and computing per-vertex normals using angle-weighted averaging of face normals has been proven to yield correct results [1].

After this initialization we use a standard fast marching method to derive distance values at the yet unknown grid nodes, using an isotropic marching with a constant speed

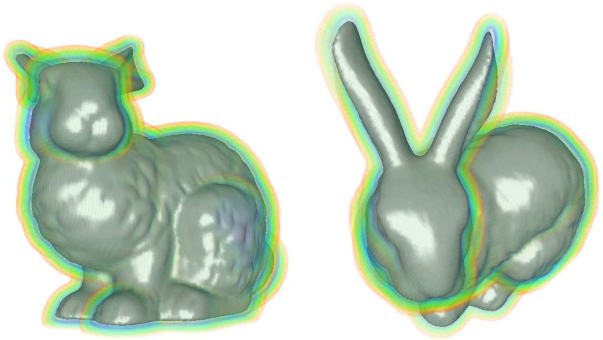

Figure 1. A volume rendering of an ε tolerance volume around the Stanford bunny model.

function being equal to 1. Starting from the already initialized grid nodes, all their immediate neighbors are inserted into a min-heap based on their distance from the advancing front. After conquering the nearest of these candidate nodes, all of its non-conquered neighbors are inserted into the heap. This evolution is usually continued until all grid points have been conquered.

In most cases we later want to test whether a given triangle lies within a user-defined tolerance volume of width ε, therefore distance values being greater than ε are not needed (cf. Fig. 1). As a consequence, we can stop the marching process as soon as a grid node of distance greater than ε is conquered. All remaining grid nodes are farther away and their distances are set to infinity.

Notice that the terms inside and outside are well-defined for closed solid input meshes only. If the input mesh is not closed, we can fall-back to an *unsigned* fast marching that we slightly adjusted to estimate the missing sign information similarly to the normal-based inside test of the initialization phase. This corresponds to a kind of extrusion of the mesh boundary and works well in practice if the width of the error volume is not too large.

In addition to the maximum error ε, the spatial extent and resolution of the grid have to be specified for the initialization. The spatial size of the grid is chosen to be the bounding box of the input mesh enlarged by ε in each direction. In order to provide an easier comparison to the method of [19], we use the same precision parameter $\rho > 1$ for adjusting the grid resolution R, although we will need significantly smaller values of ρ to achieve comparable results:

$$R(\rho) = \left\lceil \frac{\rho \cdot D}{\varepsilon \cdot \sqrt{3}} \right\rceil,$$

where ε again denotes the maximum tolerable error and D the length of the grid's diagonal, resulting in an edge length of grid cells $h = \frac{D}{R} = \frac{\varepsilon}{\rho}$.

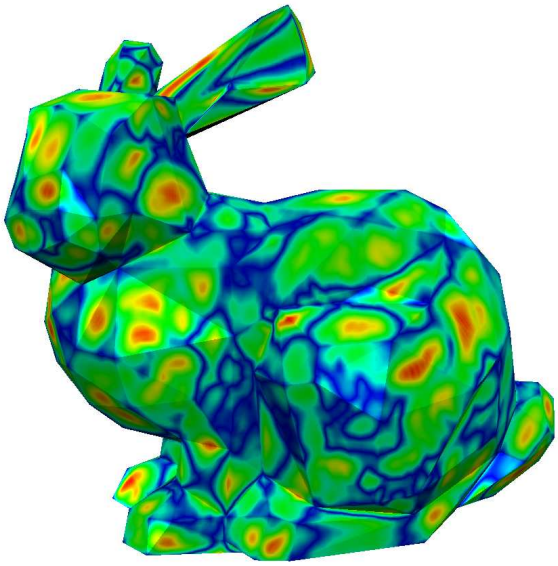

Figure 2. Using 3D distance textures and post-classification transfer functions results in a per-pixel accurate distance visualization.

The quality of the distance field approximation is determined by the grid resolution R, or the edge length h, respectively. The tri-linear interpolation within a grid cell may under-estimate the exact error by one half of the cell diagonal in the worst case. Hence, we adjust the user-specified error tolerance to

$$\varepsilon' = \varepsilon - \frac{\sqrt{3}}{2} h$$

in order to take this into account. In our experiments, however, an adjustment by $\frac{h}{2}$ turned out to be sufficient and closer to the expected under-estimation.

Since the distance field is smooth in most regions, and since we we are approximating it by a piecewise linear function, the actual error of our approximation decreases like $O(h^2)$ when increasing the grid resolution [4]. In contrast, the piecewise constant approximation of [19] improves just linearly. As we will show in Sec. 5, this allows to use coarser grid resolutions, i.e., smaller values for ρ, compared to them.

Although we use fast marching for sampling the signed distance field, any other method to compute distances from points to a reference triangle mesh could also be used. An interesting alternative would be a method like [8, 17], computing a distance field based on rendering generalized cone primitives. Combining such a method with our approach we could not only evaluate, but also generate the signed distance field completely on the GPU.

3. 3D Texture Setup

In order to use the 3D grid of signed distance values as an OpenGL 3D texture, we first have to adjust its resolution to be a power of two in each direction. This can easily be achieved by expanding the grid with empty rows, columns, or slices, and by adjusting its spatial extent accordingly. Notice, however, that the OpenGL extension `ARB_texture_non_power_of_two` relaxes this condition and avoids the waste of texture memory, but there is no implementation available yet.

Additionally we have to adjust the storage type, as the grid contains signed floating point values that are to be converted to unsigned byte values (e.g. `ALPHA8`). Mapping the range of $[-\varepsilon, \varepsilon]$ to $[0, 255]$ leads to a 8 bit quantization of the possible errors. Although this turned out to be sufficient in all our experiments, one could also use 16 or 32 bit integers for higher precision or even switch to floating point 3D textures (`ATI_texture_float`).

Rendering a triangle by accessing this 3D distance texture by texture coordinates based on vertex positions (relative to the grid) will rasterize the triangle and automatically compute the linear interpolation of the distance values in the triangle's interior. Since OpenGL assumes texture coordinates to be assigned to centers of grid cells, i.e., texels, instead of to grid nodes (see [15], p. 134), the texture coordinate $t(\mathbf{p})$ corresponding to the 3D point $\mathbf{p}$ is computed by

$$t(p) = \frac{\mathbf{p} - \mathbf{o} + \left[\frac{h}{2}, \frac{h}{2}, \frac{h}{2}\right]^T}{(R+1)h},$$

h denoting the edge length of a cell, R being the grid resolution and $\mathbf{o}$ the lower left front corner of the grid. If the grid resolution differs for x, y and z, the normalization is done component-wise. This computation of texture coordinates can be implemented very efficiently by using automatic texture coordinate generation in object space, such that it comes at no additional cost for the CPU.

The setup described so far can already be used for high-quality pixel-accurate distance visualization. A transfer function which maps the interpolated distance values to a given color range can be represented by a second RGB texture, such that a dependent texture lookup results in the desired color coding of per-pixel distance values (cf. Fig. 2). This corresponds to high-quality post-classification methods frequently used in hardware-accelerated direct volume rendering [13].

4. Distance Testing

In this section we describe how to use the 3D distance texture for implementing a generic test whether or not a given triangle lies completely within a tolerance volume around the reference surface. The idea is basically the same as for the distance visualization: we assign a special color to distance values greater than the prescribed tolerance, render the candidate triangle and detect whether this color appears in the frame buffer.

As described in the last section, rendering a triangle using the distance texture will interpolate and rasterize the distance function. Notice that the resulting texture values depend on the 3D texture coordinates only, such that we can adjust the vertex positions as long as the texture coordinates stay the same.

Instead of perpendicularly looking at each candidate triangle, we simply set its 2D vertex positions to be $(0, 0)$, $(l, 0)$ and $(0, l)$, but still use the correct texture coordinates computed from the actual 3D positions of its vertices. In order to have a sufficient resolution in the rasterization of the candidate triangle, the edge length l is determined such that the pixel resolution equals the 3D texture resolution. If $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$ denote the positions of the triangle's vertices, this edge length is

$$l = \left\lceil \frac{1}{h} \cdot \max\left\{\|\mathbf{p}_0 - \mathbf{p}_1\|, \|\mathbf{p}_1 - \mathbf{p}_2\|, \|\mathbf{p}_2 - \mathbf{p}_0\|\right\} \right\rceil.$$

In order to detect pixels violating the error bound ε, we use a transfer function assigning a completely transparent color ($\alpha = 0$) to distance values $\leq \varepsilon$ and an opaque color ($\alpha = 1$) otherwise. Using the OpenGL alpha test we discard all transparent pixels, such that as soon as one pixel is rendered, we know that the triangle violates the error bound. This, however, can easily be checked using occlusion queries (`ARB_occlusion_query`) that return the number of pixels being rendered during a query period.

Error-checking a given candidate triangle therefore amounts to computing the edge length l and texture coordinates $t(\mathbf{p}_i)$ and then rendering the 2D triangle $\Delta((0, 0), (l, 0), (0, l))$. Notice that we cannot use automatic generation of texture coordinates as the vertex positions are altered. Instead, we pass the vertex positions $\mathbf{p}_i$ in a texture coordinate register and use a texture matrix for computing $t(\mathbf{p}_i)$.

Since querying the number of rendered pixels stalls the rendering pipeline, frequent occlusion queries using just a small number of triangles each are not very efficient. However, the `ARB_occlusion_query` allows for several queries in parallel, and additionally — depending on the application — a single query may be used for testing several triangles at once by sequentially rendering all of them before checking the number of rendered pixels. An example situation may be to test all triangles being incident to a modified vertex. In the extreme case of the FFD modeling tool in Sec. 5, we even use just one query for error-checking the complete deformed mesh.

model	#tri. input	#tri. output	grid size	error	FM (s)	Deci (s)	total (s)	QEM (s)
Bunny	70K	530	$74 \times 73 \times 58$	0.92%	1.7	2.3	4.0	2.0
Horse	96k	414	$85 \times 71 \times 42$	0.99%	1.9	3.2	5.1	2.6
Venus	269k	410	$53 \times 74 \times 74$	0.96%	5.0	10.3	15.3	8.8
Buddha	1M	1955	$45 \times 101 \times 45$	1.00%	18.0	39.6	57.6	34.5

Table 1. This table reports results and timings for decimating several models using an error tolerance of 1% of the bounding box diagonal and a grid precision $\rho = 1.1$. The given error is the Hausdorff distance from the decimated mesh to the original one, computed using Metro [2]. Timings are given in seconds for fast marching (FM), decimation (Deci) and in total. The last column reports the timings for decimating to the same target complexity *without* any global error test.

5. Applications

We encapsulated the distance texture generation and the generic triangle test within an easy-to-use global error module. After initializing the distance texture by specifying a reference triangle mesh, the error tolerance ε, and the grid resolution, an arbitrary list of triangles can be tested. In addition, the distance texture can also be used to visualize the error by color-coding per-pixel distances. All timings we give in this section have been acquired on a 2.8GHz Pentium 4 and a GeforeFX 5950 graphics card.

The first example application we enhanced by this global error module is mesh decimation. Our implementation closely follows the one of Zelinka and Garland [19] to enable a comparison of both methods. In each iterative decimation step a priority queue based on error quadrics provides the best edge to be collapsed [7]. The corresponding edge collapse is simulated and all affected triangles are tested to stay within the prescribed error bound by our GPU-based distance module. If they do, the collapse is performed, otherwise it is discarded. Notice that in this setting each candidate edge collapse leads to one query for about six triangles simultaneously.

Table 1 lists the decimation results and timings for several models. Based on these values a comparison with permission grids (normalizing by the unbounded QEM method used in both papers) shows that our approach is faster by a factor of about 1.5–2 for the tested models. Note also that we derive comparable results using a grid precision factor $\rho = 1.1$, compared to $\rho = 4$ for permission grids. Although we use one byte for each grid node instead of one bit only, our memory consumption is still smaller by a factor of about 6. Notice that for larger grid resolutions, e.g., when using smaller errors, the advantage of our piecewise linear approximation pays off even more due to the quadratic approximation order.

The second algorithm to which we add an error control is mesh smoothing [5]. In each iteration an update vector for each vertex is computed based on its Laplacian or Bilaplacian vector. This update step is simulated for each vertex and its one-ring triangles are tested to stay within the error bound. If one of these triangles violates the error tolerance, the position of the vertex is simply reset to its previous value. This error-bounded Bilaplacian smoothing can be performed at a rate of about 270k vertices per second (cf. Fig. 3).

The final example is a freeform deformation based modeling tool [14]. A regular 3D lattice of control points is used to deform the space around a given surface using a trivariate tensor-product NURBS function. Evaluating this volumetric function at a 3D point $\mathbf{p}$ yields its new position after the modification. One drawback of this otherwise intuitive user interface is that it is hard to predict by which amount the surface is changed when moving several control points. After integrating our global error module, the distance visualization gives real-time feedback to the designer (15M triangles/sec), such that precise deformations can be performed at no additional cost. In addition, a global error check can be applied to all triangles being affected by a deformation using one global query only. Blocking a deformation that would otherwise violate the error tolerance ensures that the deformed model does not deviate too much from a given reference surface. As pipeline stalls are avoided when using one single query for all affected triangles, this error check can be applied at a rate of 3M triangles/sec.

6. Conclusion

We presented an efficient method for computing and evaluating tolerance volumes around a given reference surface. The initialization phase is based on highly efficient fast marching methods, requiring just a couple of seconds for moderately complex models.

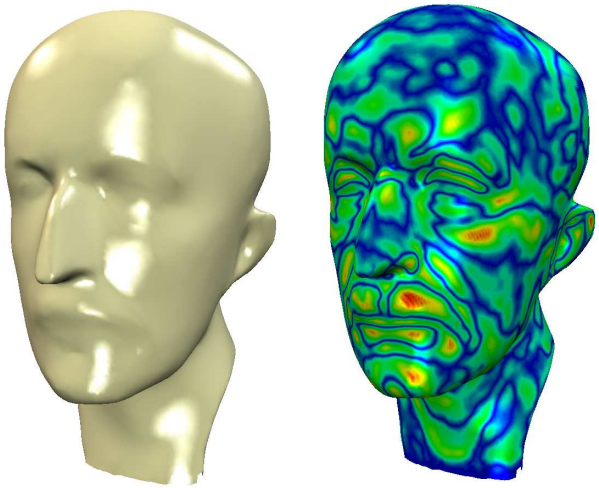

Figure 3. The result of an error bounded Bilaplacian smoothing of a $50k$ triangles Max Planck model.

Using this distance volume as a 3D texture exploits hardware acceleration and provides real-time visualization of the deviation from the reference geometry. As no 2D error textures have to be precomputed, our method can be used to visualize distances even for dynamically changing meshes.

Using special transfer functions and occlusion queries results in an application-independent global error check for a set of given candidate triangles. Exploiting hardware acceleration for the rasterization and tri-linear interpolation enables a highly efficient implementation. Since the performance of modern GPUs increases significantly faster than the performance of CPUs, the efficiency gain of GPU-based methods is expected to get even larger in the future.

Besides its high efficiency, one of the main advantages of our approach is that it is both easy to use and easy to implement, since all complicated algorithmic tasks are performed by the graphics card. Although texture memory is practically more limited than main memory, the possible grid resolutions have proven to be sufficient due to the piecewise linear distance field approximation.

Future work would include to delegate also the distance field generation to the GPU, as proposed by [17]. Additionally, our distance field evaluation is not restricted to triangle meshes, but is applicable to all primitives enabling OpenGL rendering. Hence, a promising direction of future research could be to enhance geometry processing methods working with other surface representations, like splines or point-based geometry, by our approach.

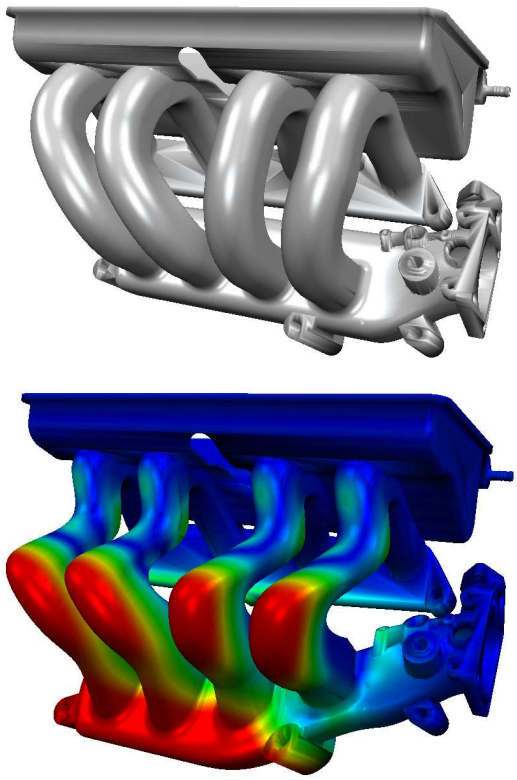

Figure 4. The presented distance visualization and error control can easily be integrated into any mesh processing algorithm, like freeform deformation based mesh modeling. In addition to visualizing the deviation from the initial state, deformation violating a prescribed tolerance can also be prevented efficiently.

Acknowledgements

We want to thank Michael Lambertz for implementing the distance visualization and Andreas Wiratanaya for integrating the distance framework into his freeform modeling tool.

References

[1] H. Aanæs and J. A. Bærentzen. Pseudo–normals for signed distance computation. In *Proceedings of Vision, Modeling and Visualization 03*, pages 407–413, 2003.

[2] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[3] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. Wright. Simplification

envelopes. In *Proceedings of ACM SIGGRAPH 96*, pages 119–128, 1996.

[4] P. Davis. *Interpolation and Approximation*. Dover Publications, 1975.

[5] M. Desbrun, M. Meyer, P. Schröder, and A. H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of ACM SIGGRAPH 99*, pages 317–324, 1999.

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

[7] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of ACM SIGGRAPH 97*, pages 209–216, 1997.

[8] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of ACM SIGGRAPH 99*, pages 277–286, 1999.

[9] R. Klein, G. Liebich, and W. Straßer. Mesh reduction with error control. In *Proceedings of Visualization 96*, pages 311–318, 1996.

[10] L. Kobbelt and M. Botsch. Freeform shape representations for efficient geometry processing. In *Proceedings of Shape Modeling International 03*, pages 111–118, 2003.

[11] L. Kobbelt, S. Campagna, and H.-P. Seidel. A general framework for mesh decimation. In *Proceedings of Graphics Interface 98*, pages 43–50, 1998.

[12] K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr. Level set surface editing operators. In *Proceedings of ACM SIGGRAPH 02*, pages 330–338, 2002.

[13] C. Rezk-Salama. *Volume Rendering Techniques for General Purpose Graphics Hardware*. PhD thesis, University of Erlangen-Nürnberg, 2001.

[14] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. In *Proceedings of ACM SIGGRAPH 86*, pages 151–159, 1986.

[15] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 1.5). http://www.opengl.org, 2003.

[16] J. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proceedings of the National Academy of Science*, volume 93, pages 1591–1595, 1996.

[17] A. Sud, M. A. Otaduy, and D. Manocha. DiFi: Fast 3D distance field computation using graphics hardware. In *Proceedings of Eurographics 04 (to appear)*, 2004.

[18] J. Wu and L. Kobbelt. Piecewise linear approximation of signed distance fields. In *Proceedings of Vision, Modeling and Visualization 03*, pages 513–520, 2003.

[19] S. Zelinka and M. Garland. Permission grids: practical, error-bounded simplification. *ACM Transactions on Graphics*, 21(2):207–229, 2002.