A data-dependent gradient quantization scheme for the acceleration of volume rendering

Peer-Timo Bremer[a], Oliver Kreylos[a] and Bernd Hamann[a]

[a]Center for Image Processing and
Integrated Computing (CIPIC)
Department of Computer Science
University of California
Davis, CA 95616-8562, U.S.A.

ABSTRACT

Volume rendering requires the use of gradient information used as surface normal information, for application of lighting models. However, for interactive applications on-the-fly calculation of gradients is too slow. The common solution to this problem is to quantize gradients of trivariate scalar fields and pre-compute a look-up table prior to the application of a volume rendering method. A number of techniques have been proposed for the quantization of normal vectors, but few have been applied to or adapted for the purpose of volume rendering.

We describe an new data-dependent method to quantize gradients using an even number of vectors in a table. The quantization scheme we use is based on a tessellation of the unit sphere. This tessellation represents an "optimally" distributed set of unit normal vectors. Staring with a random tessellation, we optimize the size and distribution of the tiles (on the unit sphere) with a simulated annealing approach.

Keywords: Gradient quantization, volume rendering, lighting models, tessellation, triangulation

1. INTRODUCTION

In the field of volume visualization, the quantization of normal vectors – defined by the gradients of some underlying trivariate scalar-valued function – has become more and more important. Each cell in a typically rectilinear volumetric grid usually implies its own normal vector, resulting in a large number of different vectors. In general, there can be as many different normal vectors (gradients) as there are vertices in a discrete volumetric data set. Especially for real-time viewing and virtual reality (VR), on-the-fly calculation of the normals is too slow for very large data sets. Considering the sizes of 3D data sets created by modern imaging devices, for example, storing all normals is not possible.

The pre-computation and quantization of normal vectors is therefore a necessary means to achieve frame rates high enough for interactive volume rendering applications. The most intuitive way to represent a quantization defined by a finite set of unit normal vectors is through the use of a tessellation of the unit sphere. Each normal corresponds to one point on the unit sphere, and the vertices (or faces) of the tessellation represent the look-up vectors and the distribution of the normal vectors.

One can use the vertices of the tessellation as look-up vectors. Assigning each of the original normal vectors to the nearest vertex is equivalent to computing the Voronoi diagram of the vertices, see Renka.[1] This diagram partitions the sphere and one assigns each original normal according to its Voronoi region, see Figure 1 (a). One can also use the faces of the tessellation to define a distribution on the unit sphere. The look-up vectors are then the normal vectors of the faces, see Figure 1 (b).

Further author information: (Send correspondence to)
{bremerp,kreylos,hamann}@cs.ucdavis.edu

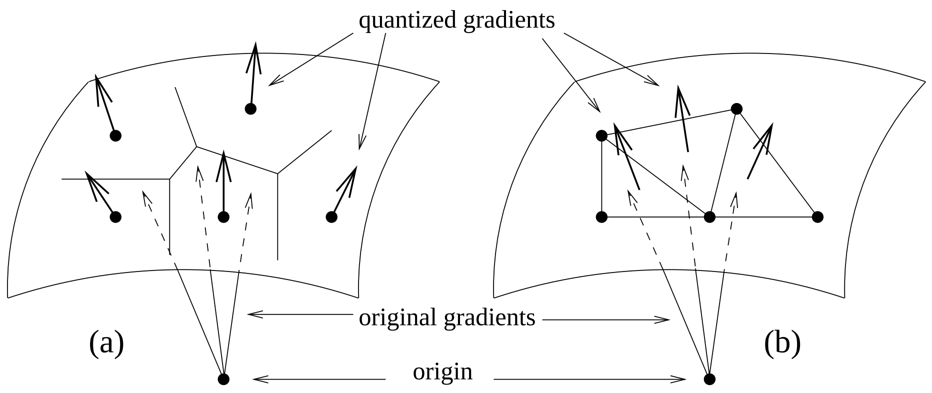

Figure 1. Representation of a normal field quantization as a tessellation (left:quantized gradients associated with the vertices, right: quantized gradients associated with normal vectors of triangles).

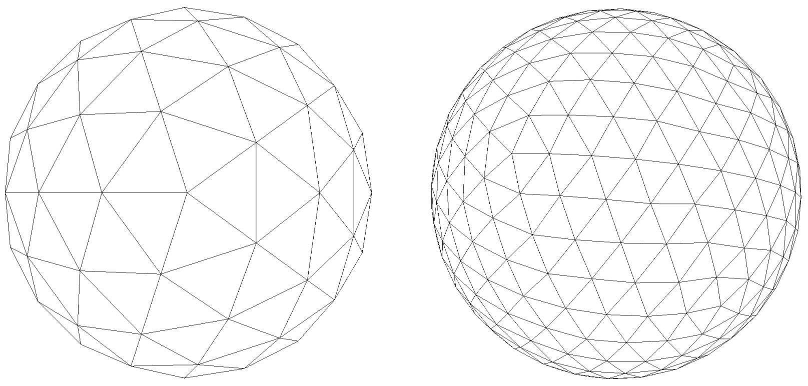

Figure 2. Regular tessellation of unit sphere - refinement levels one and two.

2. RELATED WORK

Several quantization techniques are known in the *clustering* literature, see, for example, Jain et al..[2] For volume rendering, we need a clustering algorithm that uses a distance metric as cluster criterion. Additionally, we want to quantize considering a user-defined number of clusters, and power-of-two numbers are advantageous. Using a power-of-two number of clusters eliminates memory overhead from using not all possible entries in a look-up table. The last important property is data dependency. For example, let us consider the data set of an airplane wing. The vast majority of normal vectors of points on a wing points either upwards or downwards. However, a uniform quantization uses the same number of entries in the look-up table for all normal directions.

Most volume rendering methods are based on a quantization using a regular subdivision of the unit sphere. For example, Gelder and Kim[3] tessellate the unit sphere into triangles and use the vertices of the triangles as look-up vectors. They start by combining the regular icosahedron with its dual, the dodecahedron. This results in a triangulation consisting of 32 vertices and 60 triangles. Additional refinements are based on a regular 1-to-4 subdivision of the initial triangles. The general rule for the number of vertices after i refinement steps is $|\mathrm{v}| = 30 * 4^i + 2$. However, for the common case of an 8-bit look-up table, their method can only use 122 quantized vectors. Even using the normals of the triangles as look-up vectors only allows the use of the 240 normals of the first refinement level instead of 256 possible vectors.

Other methods used to create a look-up table are described by Glassner.[4] The first technique he proposes partitions each coordinate separately. For an 8-bit table, one can use the first four bits for the x-coordinate and the second four bits for the y-coordinate, partitioning each axis by equidistant intervals. Yet another quantization

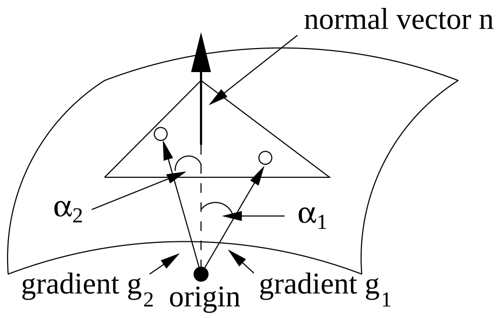

Figure 3. Parameters considered by target function.

method Glassner describes uses an equal-area spiral, which starts at the origin and winds outward. Each position along this spiral represents one normal.

Our method starts with a random triangulation of the unit sphere with a user-defined even number of triangles. We use the normal vectors of the triangles as look-up vectors. In volume rendering, the zero vector must be included in a look-up table, because there might be vertices of the data set where the gradient vanishes or is undefined. Therefore, out method still "wastes" one entry in an 8-bit table, thus representing 255 real entries – 14 more than possible with regular subdivision. In general, our method wastes at most one entry in any given look-up table, where the regular subdivision scheme tends to become more wasteful as the tables get larger. (A 9-bit table uses 483 of 512 entry possibilities, and a 10-bit table 961 of 1024 entry possibilities.) Additionally, our method is data-dependent and uses a simulated annealing approach to optimize the partitioning of the sphere into tiles/triangles, see section 3.

Another data-dependent approach is a generalization of Lloyd's algorithm.[5][6] It iteratively optimizes the partitioning based on a random start partition. However, this generalization uses a greedy approach that strongly relies on the start partition to reach a global optimum. Our tests suggest that the given problem has a large number of local optima, which is an argument against using a greedy algorithm.

3. THE ALGORITHM

We start with a random convex triangulation of the sphere. We then assign each of the original gradients computed for a given volumetric data set to a certain triangle, see section 3.1, and apply a method to change the triangulation and the data assignment randomly, see section 3.2. We optimize the quantization using a *simulated-annealing* algorithm, also called *Metropolis algorithm.*[7] Simulated annealing models the state transition from fluid to crystalline state of metals. From an algorithmic view point, this process is an optimization process with extremely high dimension. To apply simulated annealing to a general optimization problem, one needs to formulate the given problem as a cooling process. In our application, the temperature of the process is represented by a target function that defines a global error function, which is defined as $GlobalError = \sum_i \big(1 - cos(\alpha_i)\big)$, where α_i is the angle between the gradient g_i and the normal n_{g_i} of the triangle to which g_i is assigned, see Figure 3. This error function is easy to compute, since $1 - cos(\alpha_i) = 1 - g_i * n_{g_i}$ ("*" indicating the inner product). Mathematically, the value of $GlobalError$ is proportional to the sum of the squared distances, as

$$||g-n||^2 \quad = \quad 2\Big\{1 - \cos\big(\angle(g,n)\big)\Big\}. \tag{1}$$

We change the tessellation randomly, moving random vertices by a small angle – between $10° - 15°$. After each change, the assignment of gradients to tiles/triangles is updated and the resulting target function is re-computed. We accept or do not accept a change following the rules of simulated annealing. The overall goal is to minimize the angle difference between the original gradients and the look-up table. Therefore, visual artifacts resulting from look-up-table-based volume rendering will also be minimized.

```
procedure quantize(Gradients G, Tessellation T)
{
    createRandomTessellation(T);
    assignGradients(G, T);
    temperature = calculateStartTemperature(G, T, START_PROB);
    for(i=0; i<CYCLE_NR; i++) { /* During each CYCLE the temperature is updated */
        for(j=0; j<STEP_NR; j++) { /* Number of steps for each temperature */
            moveRandomPoint(G, T, temperature);
            error_diff = reassignGradients(G, T);
            p = randomBetween(0, 1);
            if (p ≤ exp(-error_diff/temperature)) /* Probability to accept the move */
                finalizeChange(G, T);
            else
                reverseMove(G, T);
        }
        temperature *= TEMP_FAC;
    }
}
```

Figure 4. Pseudocode for gradient quantization.

3.1. Gradient Assignment

There are two possible ways to assign gradients to triangles of the tessellation: 1) One can assign the gradient to the triangle whose normal vector has the smallest angle difference; 2) one can use the convex triangulation as a partitioning of space, assigning each gradient, interpreted as a normal vector, to the triangle it intersects. The first

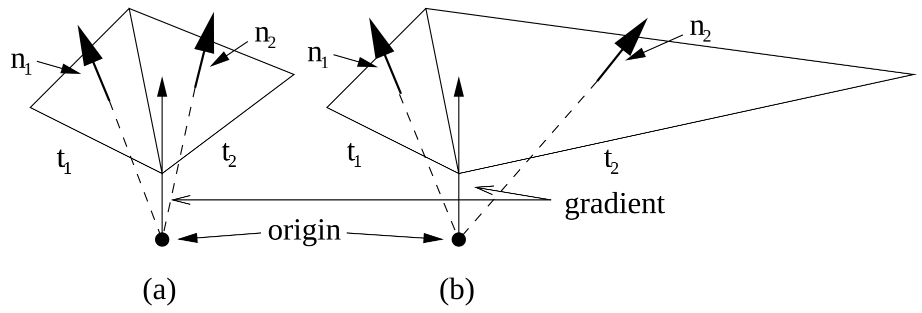

Figure 5. Gradient assignment.

method is more accurate. For example, let us consider the situation depicted Figure 5: The vectors n_1 and n_2 represent the normals of triangles. Assigning gradient g to triangle t_1, following method 2, results in a larger angle difference than necessary. However, after a configuration change we can compute the assignment applying method 2 locally. During a vertex movement, we store all changed triangles. Since the border of this triangle patch (all triangles that share the vertex we move) did not change, all gradients that were assigned to this patch before must be re-assigned to a triangle of this patch. This is not possible when one uses method 1. Using method 1, one has to use global, or at least much larger, local test areas. As it turns out, the re-assigning of gradients and re-computing of the target function are the expensive operations of our algorithm. Furthermore, accuracy is only an issue for long and skinny triangles, which do not occur often in the resulting triangulations. Therefore, we use the partitioning of the sphere to guide gradient assignment.

3.2. Configuration Changes

To change a tessellation, we use a modified version of the algorithms described in Bremer et al.[8] and Kreylos and Hamann.[9] They use three different operations: edge rotation, vertex removal, and vertex movement. Edge rotation only changes the triangulation of two neighboring triangles. To move a vertex Bremer et al. randomly choose a new position inside a small sphere around the original one. To check the validity of the resulting triangulation all involved triangles are projected onto the plane defined by the vertex normal of the "changing" vertex. This can lead to degenerate triangles or triangles with wrong orientations. These conflicts can be resolved by swapping the appropriate edges. To remove a vertex, the shortest edge emanating from the vertex is collapsed. For out application, we only require the vertex movement operation. (The algorithm described in 8. was designed for parametric surfaces and was later modified for arbitrary triangulated two-manifold surfaces. It is based on projecting the platelet of a point to be moved without causing self-intersections onto a properly chosen plane. The algorithm changes a configuration by sequentially moving vertices and rotating edges, always avoiding degenerate triangles.)

In the case of a sphere tessellation, an appropriate projection plane is easy to find. The algorithm of Bremer et al. only guarantees the absence of local self-intersections. For our algorithm, the triangulation has to be globally convex. This implies two conditions: 1) The triangulation must be parameterizable on the unit sphere, and 2) the triangulation must be locally convex. Condition 2) implies condition 1). Therefore, a straightforward approach to extend their algorithm is to ensure local convexity during configuration changes. However, this is difficult. Considering the

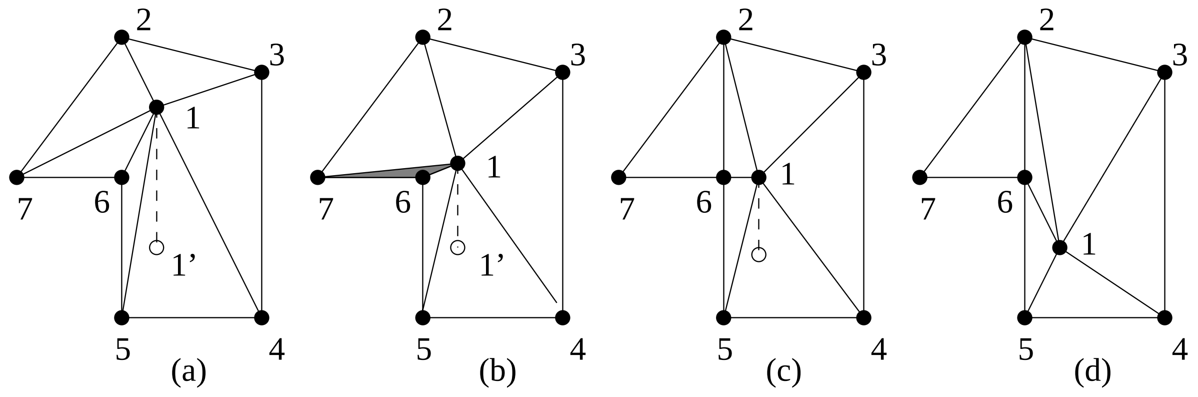

Figure 6. Vertex movement.

configuration show in Figure 6 as a projection of a small part of a tessellation onto a plane, one wants to move vertex 1 to the new position "∘". The algorithm of 8. handles this situation as if one was dragging the vertex along the dotted line. Since the shaded triangle degenerates during the movement, one rotates the appropriate edge preserving all conditions the triangulation must satisfy. All necessary computations can be performed without actually changing the coordinates of the vertex until all topological changes are done. If one wants to preserve convexity during the movement, one needs to change 1's coordinates throughout the movement. This is more difficult to implement and can increase the number of operations necessary. Since Figure 6 represents a projection onto a plane, it is possible that the edge connecting vertices 2 and 3, as part of the sphere tessellation, is convex at stage (a), concave at stage (b), and convex again at the final stage (d).

We use a different approach. During vertex movement, we preserve only the weaker condition 1) (parametrizability) and store all edges that are affected by the change. Depending on the distance by which we move a vertex, there are cases, when one cannot preserve condition 1). When this is the case we do not move the vertex. After vertex movement, we process the stored edges. For each edge, we test for convexity. If the edge is convex it is removed; if it is concave, the edge is rotated and all affected edges are added to the list.

There are cases when it is impossible to execute a movement, because we cannot preserve the constraints. As the distances of vertex movements increase, the rotating patterns become more complex, and cases not permitting movement happen more often. This results in unnecessary computations. However, the computational cost is small, compared to the cost of re-assigning normal vectors and re-computing the target function.

3.3. Implementation Issues

The most expensive part of our algorithm is the re-assignment of gradients. Our implementation attempts to avoid any redundant calculations. This becomes especially important when applying simulated annealing. For each proposed vertex movement, one has to re-assign gradients and re-compute the target function. However, a move might get rejected, and all effects have to be reversed. How to reverse a movement itself is described in detail in 8. It is our goal to avoid re-assigning gradients a second time. Each triangle has two associated lists of indices referring to a gradient array, see Figure 7 (a). The Old_List contains all gradients currently assigned to this triangle. During a vertex movement, all these gradients must be re-assigned. However, we do not want to change the Old_List until we know that the move is not rejected. Instead, we re-assign not the elements of the Old_List but "twins" of these elements. This assignment is stored in the New_List, see Figure 7 (b). If the move is accepted, we switch Old_List and New_List, and the former twins become the original, see Figure 7 (c). If the move is rejected, we delete the New_List. This method enables us to re-assign each gradient only once during a vertex movement. Unfortunately, we must store each gradient index twice. Compared to the memory requirements of the gradients themselves (three double-precision numbers per gradient), the memory requirement for one list element (one integer and two pointer variables) is minor.

Our implementation also allows us to use pre-quantized gradients. In this case, each initial gradient has an associated multiplicity, that denotes how many original gradients it represents. During error calculation, the error is multiplied by this multiplicity.

4. RESULTS

We have tested our algorithm for different resolutions and different data sets, comparing it to the regular subdivision scheme described in section 2. One important factor for any simulated annealing scheme is the choice of the cooling schedule. One needs to define an initial temperature and a cooling factor. We define the initial temperature in the following way: We track how the error function would change if k moves were performed. By Averaging the changes these moves would imply we get an approximation of the expected change of error function E(error) for a random move. The user defines a probability START_PROB, and we calculate the initial temperature so that a move that increases the error function by E(error) is accepted with the probability START_PROB. The second factor is the cooling factor. After each cycle, see Figure 4, we multiply the current temperature by TEMP_FAC to simulate cooling. We have performed several test sequences to find the "best" parameter combination, and some results are shown in Figure 8. It is interesting to notice that, for a large range of parameters, the simulating annealing is relatively insensitive to parameter changes. This effect is even more pronounced than the graphs suggest, since the graphs show averages over several runs using the same settings. While the differences in certain parts of the graphs between two curves range between 100 and 200, the range of possible values for the same parameter setting is also between 100 and 200.

We first tested our method for analytical functions. For example, we chose 5000 uniformly distributed normalized gradient vectors of the function $x^2 + 15y^2 + 15z^2$ and quantized them to 254 normals. The resulting tessellation is shown in Figure 9. The two real data sets used for Table 1 are a volumetric data set of a human foot and the geometry of an airplane wing. We pre-quantized the normals of the foot data set to 64 bits to reduce data size. However, we kept track of how many original normals were assigned to one 64-bit normal and assigned weights accordingly. The results are shown in Table 1. The two error norms norm 1 and 2 are shown. We used error norm 1 (see Section 3) during the quantization and error norm 2 (the square error norm defined as $\sqrt{\sum_i ||g_i - n_{g_i}||^2}$). Using 254 normals, our method yields a result similar to the one obtained from regular subdivision using 960 normals. Using 1022 normals produces results similar to those obtained when using 3840 normals based on regular subdivision. Results are shown in Figures 10 to 15.

All tests were performed on an SGI Onyx2 with 512MB using one R10000 processor with 195MHz. The optimizations required between four and 8 minutes. However, since re-assigning normals is the dominating factor for run time, the algorithm runs faster when using more normals to quantize to. For very large data sets, our implementation exhibits a high level of parallelism that could be exploited.

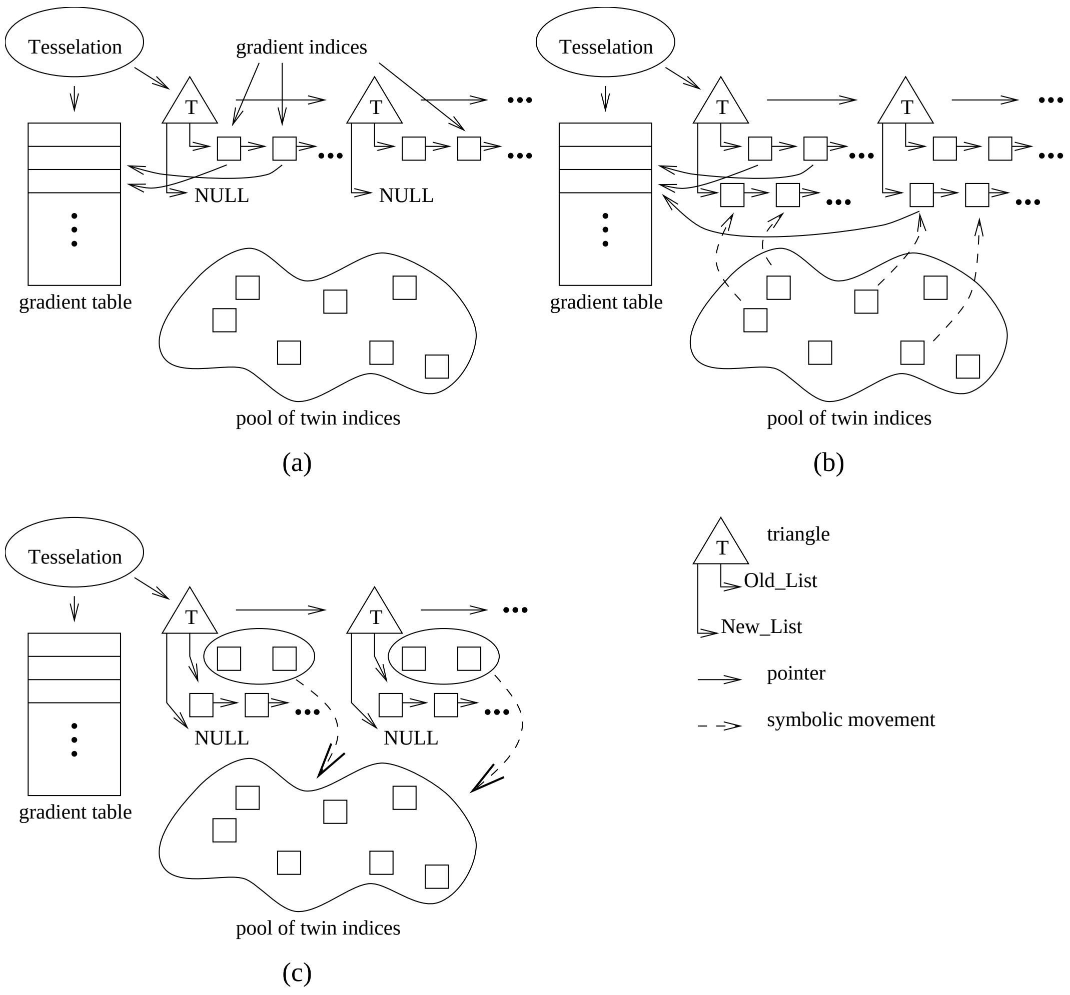

Figure 7. Data structure: (a) original; (b) gradient re-assignment; (c) finalize movement.

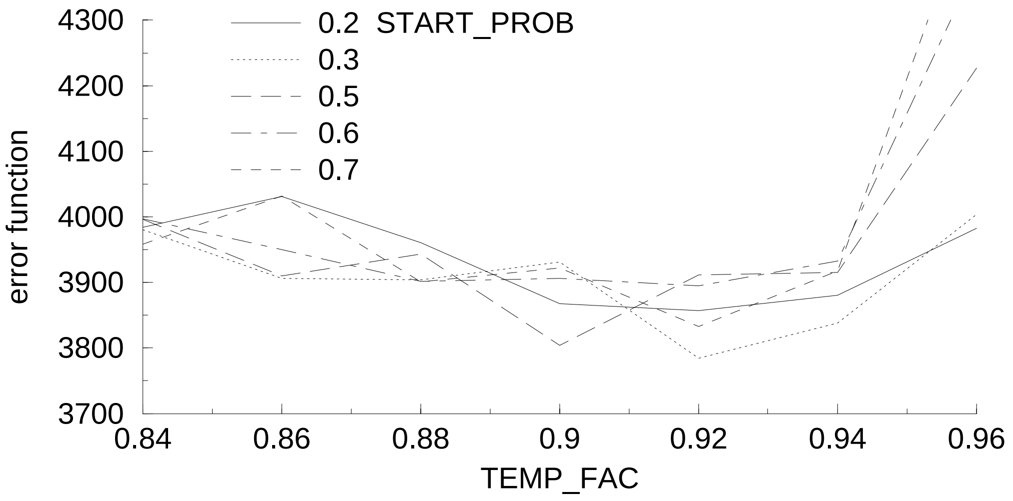

Figure 8. Behavior of error function for different parameter settings (foot data set using 254 normals.)

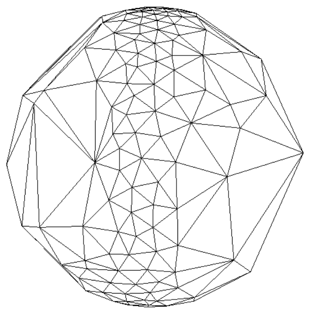

Figure 9. Tessellation of the sphere for the ellipsoid $x^2 + 15y^2 + 15z^2$ using 510 normals.

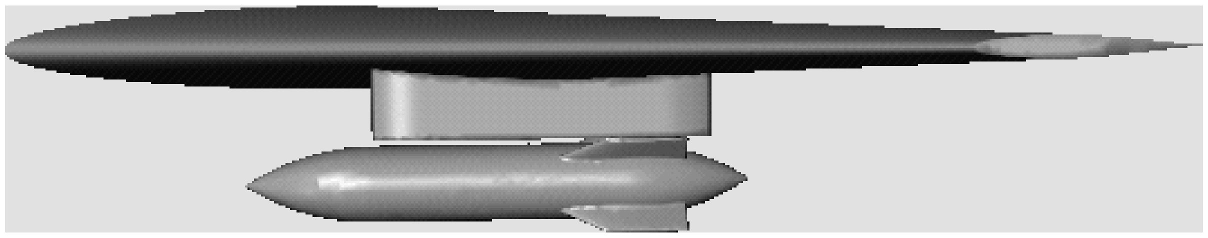

Figure 10. Airplane wing: view one original.

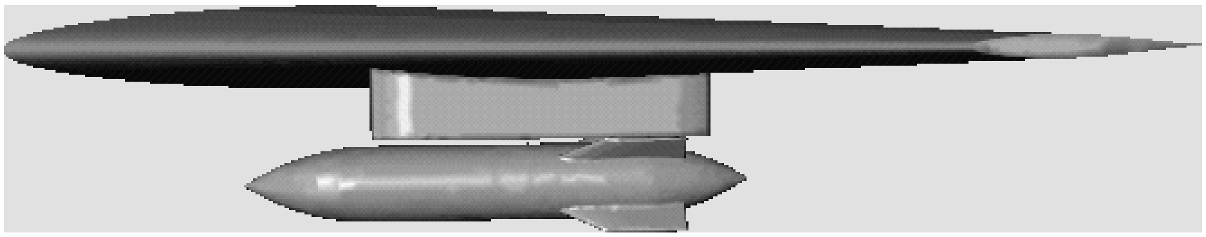

Figure 11. Airplane wing: view one using 254 optimized normals.

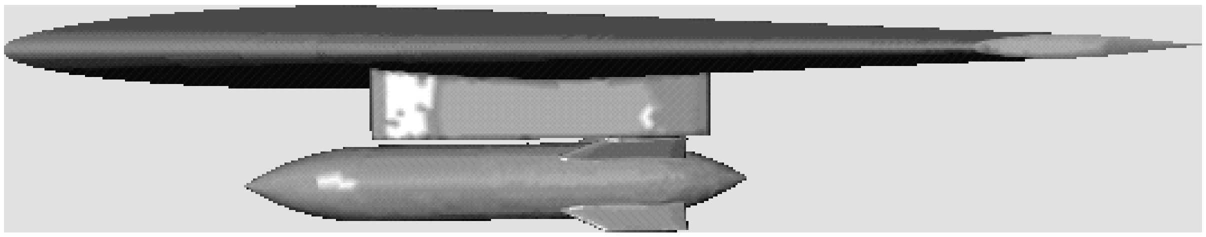

Figure 12. Airplane wing: view one using 240 evenly distributed normals.

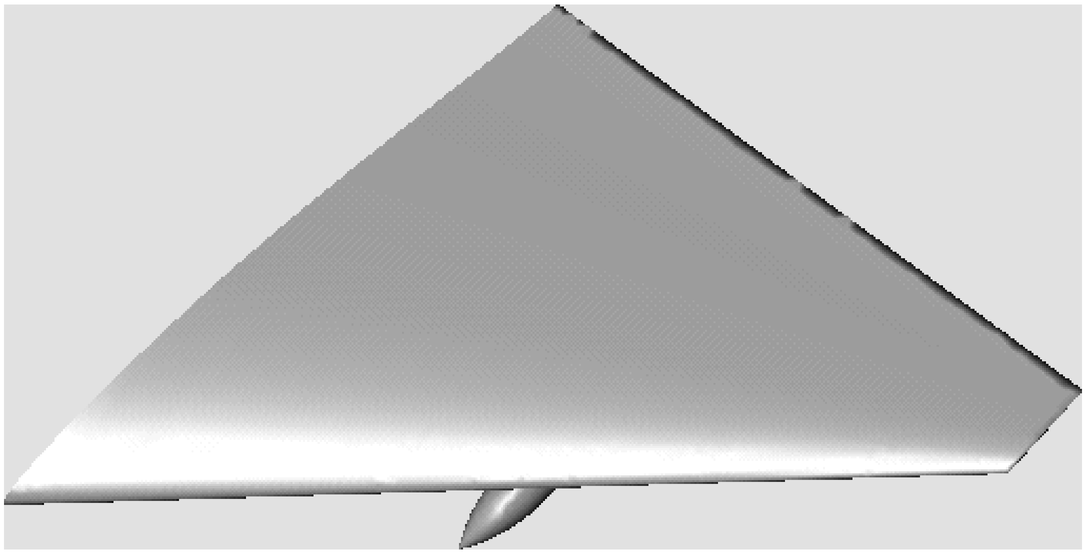

Figure 13. Airplane wing: view two using the original normals.

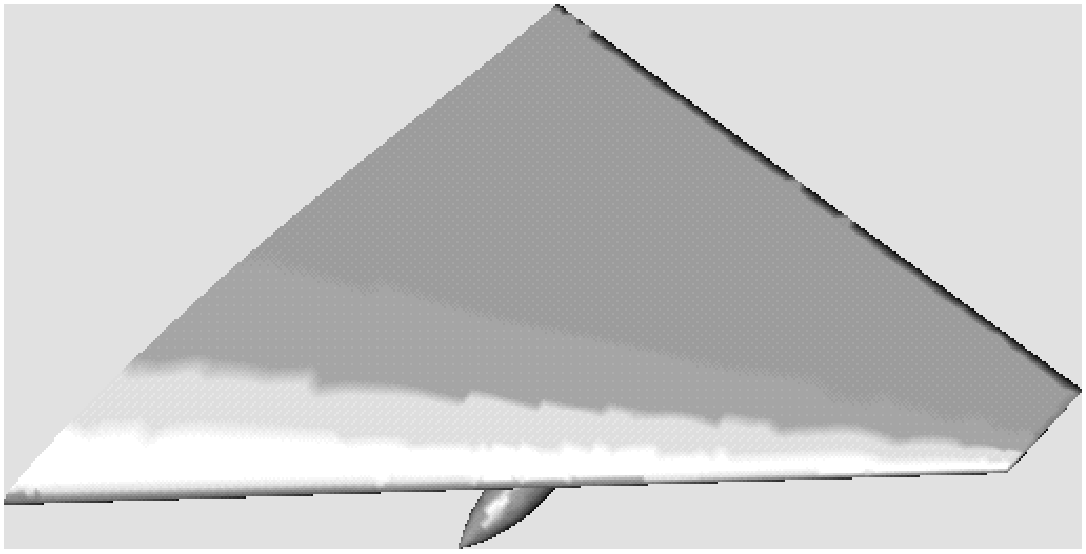

Figure 14. Airplane wing: view two using 254 optimized normals.

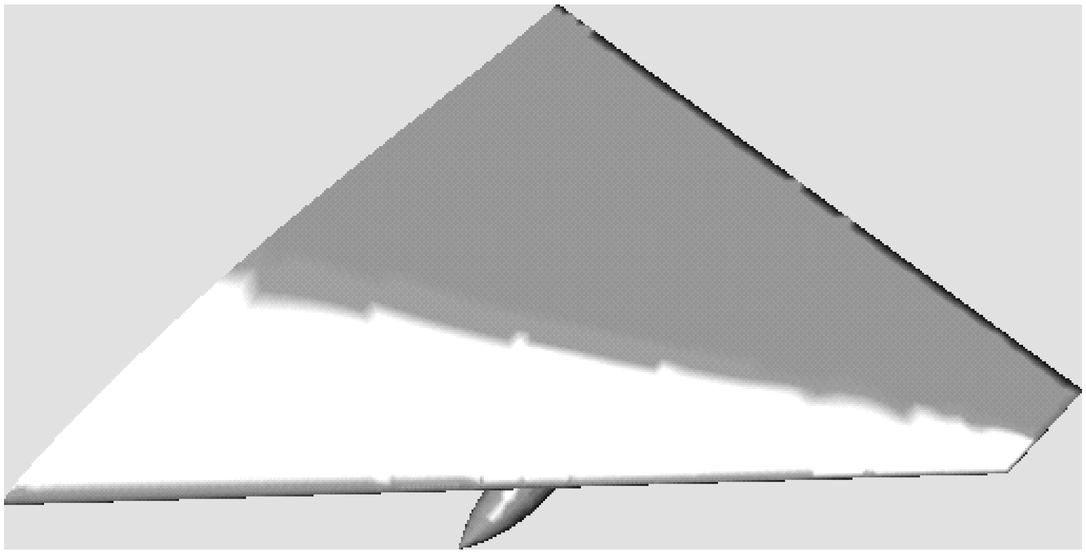

Figure 15. Airplane wing: view two using 240 evenly distributed normals.

Algorithm	# of normals	Foot data set		Airplane wing	
		error norm 1	error norm 2	error norm 1	error norm 2
	254	3788.26	43.52	14.39	2.68
Simulated	510	1761.74	29.68	6.51	1.80
Annealing	1022	754.50	19.42	4.03	1.42
	240	13030.88	80.72	74.18	6.10
Regular	960	3800.05	43.59	15.61	2.79
Subdivision	3840	906.63	21.29	4.11	1.43

Table 1. Error for foot and head data sets using different numbers of normals to quantize to.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under contracts ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the Office of Naval Research under contract N00014-97-1-0222; the Army Research Office under contract ARO 36598-MA-RIP; the NASA Ames Research Center through an NRA award under contract NAG2-1216; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; the Lawrence Berkeley National Laboratory; the Los Alamos National Laboratory; and the North Atlantic Treaty Organization (NATO) under contract CRG.971628. We also acknowledge the support of ALSTOM Schilling Robotics and SGI. We thank the members of the Visualization and Graphics Research Group of the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis and the members of the Welfen Laboratory at the University of Hannover, Germany.

REFERENCES

1. R. J. Renka, "Algorithm 772 stripack: delaunay triangulation and voronoi diagram on the surface of a sphere," *ACM Transactions on Mathematical Software* **23**, pp. 416–434, 1997.
2. A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Surveys* **31**, pp. 264–323, 1999.
3. A. van Gelder and K. Kim, "Direct rendering with shading via three-dimensional textures," in *Symposium on Volume Visualization*, *Proc. IEEE* , pp. 23–30, 1996.
4. A. S. Glassner, "Normal coding," in *Graphic Gems*, pp. 257–264, Academic Press, New York, 1990.
5. Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantization design," *IEEE Transactions on Communications* **28**, pp. 84–95, 1980.
6. P. Ning and L. Hesselink, "Vector quantization for volume rendering," in *Workshop on Volume Visualization*, *ACM Proc.* , pp. 69–74, 1992.
7. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equations of state calculations by fast computing machine," *J. Chem. Phys.* **21**, pp. 1087–1091, 1953.
8. P. T. Bremer, B. Hamann, O. Kreylos, and F. E. Wolter, "Simplification of closed triangulated surfaces using simulated annealing," in *Mathematical Methods in CAGD: Oslo 2000*, T. Lyche and L. Schumaker, eds., Vanderbilt University Press, Nashville, Tennesse, 2001 (to appear).
9. O. Kreylos and B. Hamann, "On simulated annealin and the construction of linear spline approximations for scattered data," in *Symposium on Visualization*, E. Groeller, M. Loeffelmann, and W. Ribarsky, eds., *Proc. Joint EUROGRAPHICS-IEEE TVCG* , pp. 189–198, Springer-Verlag, 1999.