PERGAMON

Computers & Graphics 24 (2000) 353–361

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Data Visualization

Data structures for optimizing linear spline approximations

Oliver Kreylos*, Bernd Hamann

Center for Image Processing and Integrated Computing (CIPIC), Department of Computer Science, University of California, One Shields Avenue, Davis, CA 95616-8562, USA

Abstract

We describe the algorithms and data structures used for optimizing linear spline approximations of bivariate functions. Our method creates a random initial triangulation of a given data set and then employs a simulated annealing algorithm to improve this initial approximation. In every iteration step, the current approximation is changed in a random but local way, and the distance measure between it and the data is re-calculated. Depending on the difference between the old and new distance measures, an iteration step is either accepted or rejected. We discuss the basic operations and data structures of our optimization technique. We present a variant of the half-edge data structure and associated algorithms.

Keywords: Linear spline approximation; Simulated annealing; Half-edge data structure; Image approximation; Triangulations

1. Introduction

In several applications one is concerned with the representation of complex geometries or complex physical phenomena at multiple levels of resolution. In the context of computer graphics and scientific visualization, so-called *multiresolution methods* are crucial for the analysis of very large numerical data sets [1–5]. Examples include high-resolution terrain data (digital elevation maps) and high-resolution, three-dimensional imaging data (e.g., magnetic resonance imaging data).

In [6,7] we introduced an approach for the construction of multiresolution approximations of very large scattered data sets using an iterative optimization algorithm based on *simulated annealing* [8–11]. In this paper, we discuss the optimization procedure in more detail, concentrating on the following special case:

(1) We only consider scattered data sets resulting from bivariate scalar- or vector-valued functions sampled at sites which are randomly distributed over the functions' domains, and

(2) we only discuss optimizing a single approximation level, where the number of vertices to be used for the optimization is fixed.

An example for this special case is the approximation of a color image, interpreted as a bivariate vector-valued function, by a triangulation consisting of a predefined number of vertices, see Section 6.

1.1. Finding optimal approximations

Our approach to finding an optimal or near-optimal linear spline approximation for a given, fixed number of vertices N_k is based on an iterative optimization algorithm. First, we create an initial configuration, then we improve this configuration by changing its vertex placement and its triangulation in every step. We judge a configuration's quality by its L^2 distance from the scattered data set. Since this optimization problem is high dimensional and generally involves local minima in abundance, the algorithm of simulated annealing is well suited to construct "good" linear spline approximations [8,11].

Simulated annealing is an iterative method that applies random changes to the current configuration and accepts a step depending on the resulting change of the error measure and a value called "temperature". This value determines the probability of accepting a step that

* Corresponding author. Tel.: +1-530-754-9470.
E-mail address: kreylos@cs.ucdavis.edu (O. Kreylos).

0097-8493/00/$ - see front matter
PII: S0097-8493(00)00031-5

increased the error measure: The higher the temperature, the higher the probability of accepting a bad step. The so-called "annealing schedule" determines how fast the temperature is decreased during the iteration.

In the case of bivariate scattered data sets the quality of a configuration depends on both vortex placement and triangulation. There are two different ways how we can proceed:

(1) One can ignore the optimization of the triangulation by enforcing a fixed triangulation type throughout the iteration process; an obvious candidate is the Delaunay triangulation [12].
(2) One can attempt to optimize both parts of the configuration in parallel. For example, before each step one could randomly decide to either move a vertex or *rotate* a common edge of two adjacent triangles, which are forming a convex quadrilateral.

2. The optimization algorithm

We now describe the individual steps of our algorithm. Algorithm 1 is a high-level description. The subsequent sections describe the important steps in more detail.

Algorithm 1: Optimizing linear spline approximations using simulated annealing.

```
Create initial configuration (vertex placement and
triangulation);
Determine initial temperature and create annealing
schedule;
while iteration is not finished {
  Change current configuration;
  Calculate change in error measure;
  Undo iteration if rejected by simulated annealing;}
return current configuration;
```

2.1. Creating an initial configuration

Our approximations are defined over the original sites' convex hull. To achieve this, our algorithm has to calculate the convex hull of all sites contained in the data set first; then it has to include all those vertices whose sites are located on the convex hull's boundary into the initial configuration. Afterwards, further vertices are randomly selected from the data set and inserted into the configuration. Thus, an initial configuration is created by triangulating a convex polygon (the data set's convex hull), and then inserting vertices into an existing triangulation at arbitrary sites. We also want all initial configurations to be Delaunay triangulations; to ensure this, we have to restore the Delaunay property after each vertex insertion using the algorithm described by Guibas et al. [13]. Thus, a data structure has to support the following primitive operations:

(1) create a Delaunay triangulation of a convex polygon;
(2) in a triangulation, find the triangle containing an arbitrary point;
(3) insert a new vertex into the triangulation by splitting the triangle containing that point into three parts;
(4) restore the Delaunay property after a vertex insertion by rotating those edges in the triangulation violating it.

2.2. Creating an annealing schedule

A reasonable heuristic to define the initial temperature is to apply some steps of the iteration scheme and to define the initial temperature in a way that the annealing algorithm initially accepts an "expected bad" step with a probability of one-half. Next, we lower the temperature in steps, leaving it constant for a fixed number of iterations and scaling it by a fixed factor afterwards.

2.3. Changing the current configuration

The simulated annealing algorithm's core is its iteration step. In principle, one can use many methods to change the current configuration, but we have found out that the "split" approach, described by Algorithm 2, works very well.

Algorithm 2: Changing the current configuration.

```
if (accept With Probability (move Probability)) { /* move
a vertex */
  Choose an interior vertex v;
  Estimate v's contribution vE to the error measure;
  if (vE < localMovementFactor · E)
    Move v globally;
  else
    Move v locally;
  if (move Probability = = 1)/* Vertex movements
only? */
    Restore Delaunay property;}
else {/* rotate an edge */
  Choose a rotatable edge e;
  Rotate edge e; }
```

The constant *moveProbability* is used to control the behavior of the optimization process. If this constant's value is one, the algorithm moves a vertex in every step, and after each vertex movement the current triangulation is updated to satisfy the Delaunay property. In the other case the algorithm can either move a vertex or swap an edge, thereby optimizing both vertex placement and triangulation simultaneously.

2.3.1. Estimating the error contribution of a vertex

To estimate how much the removal of an interior vertex v would increase the overall L^2 error measure, see Section 2.4, we use the following approach: we construct an approximating least-squares plane H for all vertices surrounding v. Then we calculate h as v's ordinate-direction distance from H and A as the area of v's platelet, see Fig. 1. We define the error contribution as $\sqrt{Ah^2/3}$, to ensure that the ratio of the vertex error contribution and the used L^2 error measure is scale-invariant.

In order to calculate H, we need a basic operation that enumerates all vertices surrounding a given interior vertex of a triangulation.

2.3.2. Moving a vertex globally

If v's error contribution is smaller than a constant *localMovementFactor* times the current error measure E, we assume that v is currently located in a "flat" region of the function and should be moved away from this region. We move v *globally* to a randomly chosen new site not already being part of the current configuration. By doing this we assure that vertices get driven away from nearly flat regions of a function in early stages of the iteration. Fig. 2 shows the steps which have to be performed to move a vertex globally. According to that figure, global vertex movements can be broken down into the following primitive operations:

(1) remove a vertex from a triangulation;
(2) insert a vertex into a triangulation at the site of an arbitrary vertex from the original data set;
(3) restore the Delaunay property locally at the vertex old and new site by rotating those edges in the triangulation violating it.

2.3.3. Moving a vertex locally

When the vertex' error contribution is larger than *localMovementFactor* $\cdot E$, we assume it is currently located in an "important," high-curvature region of the target function, and we attempt to find a better site for this vertex by moving it *locally* to a new, unoccupied site in its platelet. To move a vertex locally, we "slide" the vertex on the line from its old to its new site, dragging the edges connecting it to all surrounding vertices along. Whenever a surrounding simplex becomes degenerate during the vertex motion, we rotate one edge of the affected simplex before moving the vertex any further, see Fig. 3. Local vertex movements cannot be decomposed into simpler operations and thus have to be supported directly by the data structure.

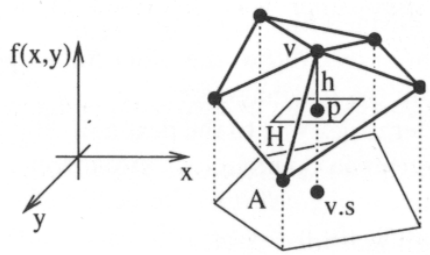

Fig. 1. Estimating how much a vertex contributes to the error measure.

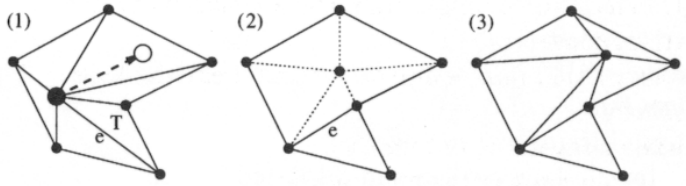

Fig. 3. Moving a vertex locally: (1) initial state; (2) rotating edge e to prevent triangle T from becoming degenerate; (3) resulting state.

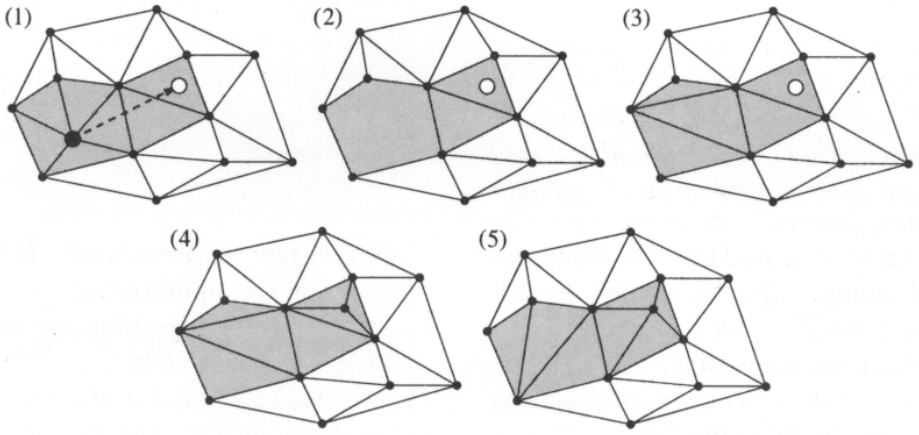

Fig. 2. Moving a vertex globally. (1) initial state; (2) removing the vertex; (3) filling the resulting hole; (4) inserting the vertex at its new site; (5) restoring the Delaunay property (only if we ignore the triangulation during optimization).

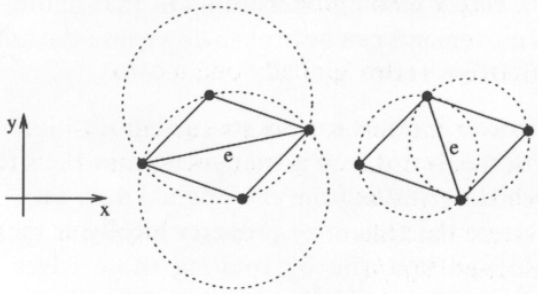

Fig. 4. Rotating an edge e: (1) initial state; (2) resulting state. The dotted circles denote the circumcircles of the two triangles before and after rotating edge e.

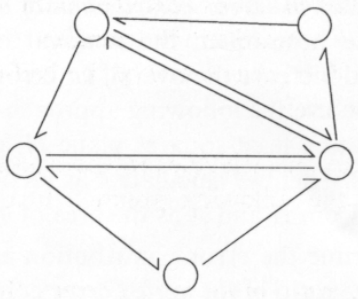

Fig. 5. Representation of a triangulation as a half-edge mesh.

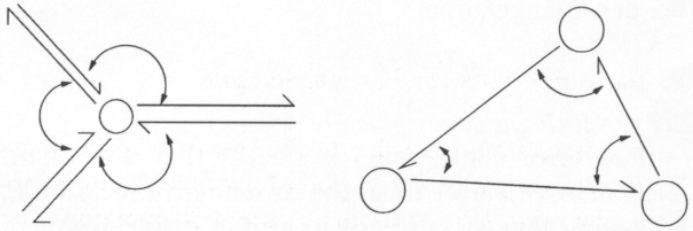

Fig. 6. Implicit representation of vertices and faces: (1) vertex loop, (2) face loop.

2.3.4. Rotating an edge

The simplest possible way to change the triangulation in an iteration step is to randomly pick an edge, which is shared by two triangles forming a convex quadrilateral, and then to rotate this edge to form the other possible triangulation of that quadrilateral, see Fig. 4.

2.4. Calculating the error measure

To calculate the L^2 distance between a configuration C and the scattered data set S being approximated, we use Algorithm 3. This algorithm can be used for vector-valued data sets without change, as long as one uses the Euclidean metric.

Algorithm 3: Calculating the error measure.

```
error = 0.0;
area = 0.0;
numDataVertices = 0;
for all triangles t in C do {
  for all data vertices s in S located in t do {
    Interpolate value tv of t at site of s;
    error += (tv − s)^2;
    area += area of triangle t; }
  numDataVertices += number of data vertices located in t; }
return sqrt((area · error)/numDataVertices);
```

Algorithm 3 requires the following primitive operations:

(1) enumerate all triangles in a configuration;
(2) enumerate scattered data located in a triangle contained in a configuration.

3. The half-edge data structure

It turns out that a variation of the *half-edge data structure* allows for easy and efficient implementation of all primitive operations needed for our linear spline optimization technique. Using this representation, a triangulation is described by a set of half-edges, where each edge in the triangulation is "divided" into two directed edges having opposite directions, see Fig. 5. Each half-edge structure is defined by

(1) the half-edge's line equation;
(2) a pointer *otherHalf* to the other half-edge constructed from the original edge;
(3) a pointer *vertexNext* to the next half-edge around the start vertex in counterclockwise direction;
(4) a pointer *triangleNext* to the next half-edge around the triangle the half-edge belongs to, also in counterclockwise direction.

The fourth element is redundant, but it transforms the sets of half-edges around vertices and triangles to double-linked lists, thus simplifying certain manipulations. In this basic data structure neither vertices nor triangles are stored explicitly; both are implicitly represented as loops in the half-edge graph, see Fig. 6.

For our purposes, we enhance the basic structure by explicitly listing vertices and triangles. By adding vertices to the representation, the primitive operations of finding random vertices and moving vertices are more easily implemented. Each vertex structure is defined by

(1) the vertex' site;
(2) a pointer *firstEdge* to one outgoing half-edge.

By adding triangles and associating data vertices with the triangles containing them we ensure efficient re-calculation of the error measure after local configuration changes. Each triangle structure is defined by

(1) a single-linked list of scattered data vertices whose sites are contained in the triangle;
(2) a pointer *firstEdge* to one of the half-edges delimiting the triangle.

To link vertices and triangles to the half-edge mesh, we have to add the following pointers to each half-edge structure:

(1) a pointer *startVertex* to the vertex the half-edge is starting at;
(2) a pointer *triangle* to the triangle the half-edge is delimiting.

4. Basic algorithms and estimated costs

In this section, we estimate the run time of the most important operations, expressed for optimization problems containing n vertices in the triangulation and N original data vertices.

4.1. Finding the triangle containing a given point

This problem, which arises in creating the initial configuration and in global movements, is solved by a breadth-first search applied to the graph of triangles, where the start point of the search is set to be the last triangle that was reported. The number of triangles in a triangulation with n vertices is $O(n)$; this ensures $O(n)$-behaviour per point for randomly arranged points and allows for speed-ups to up to $O(1)$ per point when points are inserted neighbour by neighbour. The latter is the common case when processing bitmaps or other gridded data.

4.2. Creating the initial configuration

To create the initial configuration one has to

(1) calculate the convex hull of the sites of all N data vertices ($O(N \log N)$);
(2) triangulate the convex hull ($O(\sqrt{N})$, when assuming that $O(\sqrt{N})$ vertices are lying on the boundary);
(3) insert $n - c$ randomly chosen data vertices into the triangulation, where c is the number of vertices lying on the boundary of the convex hull ($O(n^2)$);
(4) associate all data vertices with the triangles containing them ($O(Nn)$ for random data, $O(N + n)$ for gridded data: when scanning rows from left to right and right to left, vertices are inserted neighbour by neighbour);
(5) calculate the initial error measure (ON)).

Thus, the creation step requires $O(N(\log N + n))$ operations for random data, and $O(N \log N + n^2)$ operations for the common case of gridded data.

4.3. Moving a vertex globally

To move a vertex globally one has to

(1) remove the vertex from its current position ($O(1)$);
(2) re-associate the data vertices inside the vertex old platelet ($O(N/n)$);
(3) re-calculate the error measure inside the vertex old platelet ($O(N/n)$);
(4) locate the triangle containing the vertex new site ($O(n)$);
(5) split the triangle containing the vertex new site ($O(1)$);
(6) re-associate the data vertices inside the vertex new platelet ($O(N/n)$);
(7) re-calculate the error measure inside the vertex new platelet ($O(N/n)$);
(8) calculate the new overall error measure ($O(n)$).

We are assuming that, in the expected case, the number of triangles per platelet is bounded by a small constant. If we decided to uphold the Delaunay property throughout the algorithm, we would have to restore it after removing and inserting a vertex by rotating edges in the triangulation violating the property. Knuth points out that the expected number of edges to rotate is $O(1)$ [13]. This adds another $O(N/n)$ operations to re-associate and re-calculate the changed triangles. In summary, a global vertex movement requires $O(N/n + n)$ operations. We decided not to update the overall error measure incrementally to minimize the impact of numerical imprecision on the optimization result.

4.4. Moving a vertex locally

To move a vertex locally one has to

(1) move the vertex to its new position while updating its platelet on the fly ($O(N/n)$);
(2) re-associate the data vertices inside the vertex new platelet ($O(N/n)$);
(3) re-calculate the error measure inside the vertex new platelet ($O(N/n)$);
(4) calculate the new overall error measure ($O(n)$).

We are assuming that local movements do not move a vertex very far from its initial platelet. Again, to restore the Delaunay property one has to add another $O(N/n)$ for edge rotations and re-calculations. In summary, a local vertex movement requires $O(N/n + n)$ operations.

4.5. Rotating an edge

To rotate an edge one has to

(1) rotate the edge ($O(1)$);
(2) re-associate the data vertices inside the affected triangles ($O(N/n)$);

Fig. 7. A ray-traced image showing the Utah Teapot and the Utah Teacup at a resolution of 640×480 pixels.

(3) re-calculate the error measure inside the affected triangles ($O(N/n)$);
(4) calculate the new overall error measure ($O(n)$).

In summary, an edge rotation requires $O(N/n + n)$ operations.

4.6. Rejecting a configuration change

Whenever an iteration step is rejected by the simulated annealing algorithm, we have to undo the configuration change. This results in applying the same principal operation again, requiring $O(N/n + n)$ operations. Since the probability of a change being rejected increases when the error measure approaches a global minimum, the iteration steps tend to take twice as long towards the end of the iteration — almost all operations have to be undone.

5. Possible efficiency improvements

We have shown that the asymptotical run-time behaviour for all types of iteration steps is $O(N/n + n)$. Our experiments have indicated that re-associating the data vertices and re-calculating the error measure are the most important factors for overall run-time behaviour.[1] We believe that parallelizing these operations, either on a triangle-by-triangle basis or on a vertex-by-vertex basis, would result in considerable accelerations. Choosing the former might be easier, since the error measure contributions of distinct triangles are independent of each other. On the other hand, since the number of triangles per platelet is usually small, the maximum speed-up factor would be small as well. The latter way of parallelization might be more complex and might require more interprocessor communication, but it might allow for higher possible speed-up factors, since the number of data vertices per triangle is usually large.

Accelerating the point-location algorithm, for example by a grid-cell approach, does not seem very promising, since the triangulation changes in every iteration step (inducing overhead to update the point-location structure), and because only 2–10% of iteration steps are global vertex movements.

[1] The $O(n)$ part introduced by re-calculating the error norm after every change could be eliminated by using incremental updates; this is dangerous, however, because build-up of numerical imprecision can keep the iteration from converging.

6. Examples and results

We examine our algorithm's run-time behaviour in detail for one specific example. The source data sets are two ray-traced images of the Utah Teacup and the Utah Teapot, see Fig. 7, placed on a table texture mapped with a wood texture containing very strong high-frequency components. The two images have resolutions of 320×240 and 640×480 pixels, respectively. We have approximated the images at several levels of resolution, listing the error measures and elapsed run times after certain numbers of iterations in Table 1. Each table entry shows the error measure after application of the

Table 1
Error measures and elapsed run times for Utah Teapot and Utah Teacup images

Image res. (pixels)	No. of vertices	Number of iterations					
		0	50,000	100,000	150,000	300,000	500,000
320 × 240	1000	45.0536	24.8158	21.2617	18.5788	15.1631	13.6342
		0.0 s	43.0 s	87.4 s	132.6 s	281.4 s	482.5 s
	2000	39.3842	25.2299	23.1803	20.8313	17.7345	14.9551
		0.0 s	47.1 s	90.3 s	133.2 s	260.9 s	436.6 s
640 × 480	1000	92.9234	50.5108	43.8778	39.3941	31.6659	28.2213
		0.0 s	132.6 s	271.4 s	415.3 s	900.2 s	1557.0 s
	2000	81.5383	50.7222	45.4626	41.9275	35.3801	30.0904
		0.0 s	90.9 s	179.1 s	267.18 s	532.9 s	911.2 s

Fig. 8. The approximation of the 320 × 240 image, using 1000 vertices. Left: after 50,000 iterations, right: after 500,000 iterations.

Fig. 9. The approximation of the 320 × 240 image, using 2000 vertices. Left: after 50,000 iterations, right: after 500,000 iterations.

indicated number of iterations in the top row and the run time in the bottom row. Run times were measured on an SGI Onyx 2 workstation using one MIPS R10K processor running at 195 MHz. See Figs. 8–12 for approximation results. When comparing the results for approximations using 1000 vertices and 2000 vertices, one finds out that the latter are worse; this special source bitmap is a difficult case for the algorithm, because it exhibits strong high-frequency components and noise in the lower half of the image.

7. Conclusions and future work

We have provided an in-depth analysis of the data structures, algorithms and performance of our linear spline optimization algorithm based on simulated annealing. We have provided approximate asymptotic run-time bounds for our algorithm and discussed possible ways to increase the algorithm's performance.

The main areas for future research are the generalization of our algorithm to functions of three and more

Fig. 10. The approximation of the 640×480 image, using 1000 vertices. Left: after 50,000 iterations, right: after 500,000 iterations.

Fig. 11. The approximation of the 640×480 image, using 2000 vertices. Left: after 50,000 iterations, right: after 500,000 iterations.

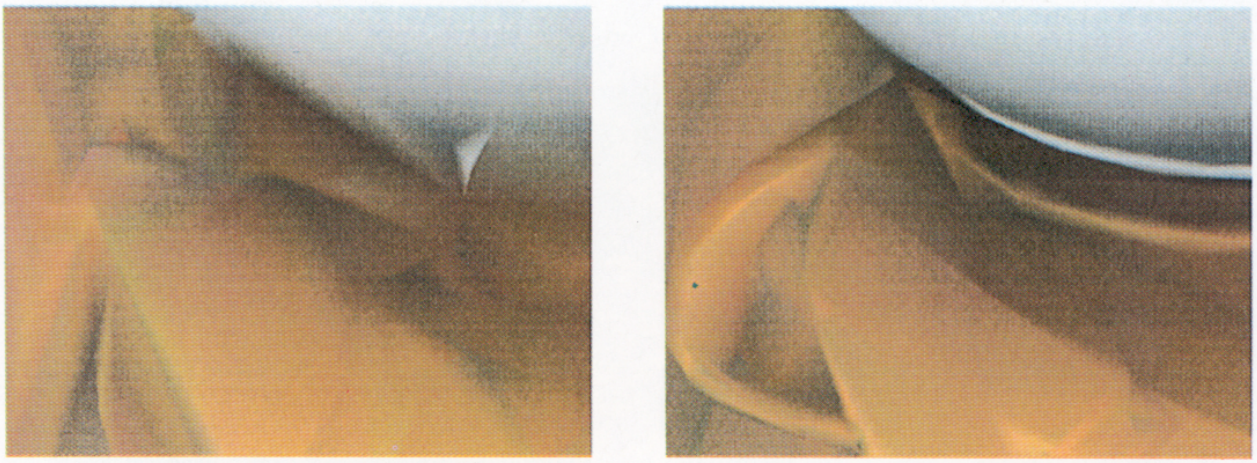

Fig. 12. Close-ups of the approximation of the 320×240 image, using 1000 vertices. Left: after 50,000 iterations, right: after 500,000 iterations.

variables and the application of our method to large-scale scientific visualization and image and video compression. If one treats video data as time-varying bivariate vector-valued functions and exploits the strong frame coherence of video streams especially in teleconferencing, our algorithm might lead to a new real-time video compression method. We also believe that our algorithm will be very helpful for the construction and visualization of hierarchies of linear spline approximations of massive numerically simulated data sets. Since the output of our algorithm is a direct geometric representation of the data sets, rapid display of large-scale data sets would be possible. Another interesting application might be the interactive generation of isosurfaces for different isovalues. Since data sets are approximated using only a small number of primitives, one would not have to generate massive sets of polygons first (for example by using a marching cube algorithm) and decimate them in a second step.

Acknowledgements

This work was supported by the National Science Foundation under contract ACI 9624034 (CAREER Award); 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; and the North Atlantic Treaty Organization (NATO) under contract CRG.971628 awarded to the University of California, Davis. We also acknowledge the support of ALSTOM Schilling Robotics, Davis, California; Chevron; and Silicon Graphics, Inc. We thank the members of the Visualization Thrust at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

References

[1] Bonneau GP, Hahmann S, Nielson GM. BLaC-wavelets: a multiresolution analysis with non-nested spaces. In: Yagel R, Nielson GM, editors. Visualization '96. Los Alamitos, CA: IEEE Computer Society Press, 1996. p. 43–8.

[2] Eck M, DeRose AD, Duchamp T, Hoppe H, Lounsbery M, Stuetzle W. In: Cook R, editor. Multiresolution analysis of arbitrary meshes. Proceedings of the SIGGRAPH 1995. New York, NY: ACM Press, 1995. p. 173–82.

[3] Gieng TS, Hamann B, Joy KI, Schussman GL, Trotts IJ. Construction hierarchies for triangle meshes. IEEE Transactions on Visualization and Computer Graphics 1998;4(2):145–61.

[4] Hamann B. A data reduction scheme for triangulated surfaces. Computer Aided Geometric Design 1994;11(2):197–214.

[5] Hamann B, Jordan BW, Wiley DA. On a construction of a hierarchy of best linear spline approximations using repeated bisection. IEEE Transactions on Visualization and Computer Graphics 1999;5(1/2):30–46, 190 (errata).

[6] Kreylos O, Hamann B. On simulated annealing and the construction of linear spline approximations for scattered data. In: Gröller E, Löffelmann H, Ribarsky W, editors. Proceedings of the EUROGRAPHICS-IEEE TCCG Symposium on Visualization, Data Visualization '99. Vienna, Austria: Springer, 1999. p. 189–98.

[7] Kreylos O, Hamann B. On simulated annealing and the construction of linear spline approximations to scattered data. IEEE Transactions on Visualization and Computer Graphics, 1999, submitted for publication.

[8] Schumaker LL. Computing optimal triangulations using simulated annealing. Computer Aided Geometric Design 1993;10:329–45.

[9] Nielson GM. Scattered data modeling. IEEE Computer Graphics and Applications 1993;13(1):60–70.

[10] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical recipes in C, 2nd edn. Cambridge, MA: Cambridge University Press, 1992.

[11] Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, Teller E. Equations of state calculations by fast computing machines. Journal of Chemical Physics 1953;21:1087–92.

[12] de Berg M, van Kreveld M, Overmars M, Schwarzkpf O. Computational Geometry. New York, NY: Springer, 1990.

[13] Guibas IJ, Knuth DE, Sharir M. Randomized incremental construction of Delaunay and Voronoï diagrams. In: Proceedings of the 17th International Colloquium Automata, Languages and Programming, Springer Verlag Lecture Notes in Computer Science, Vol. 443. Berlin: Springer, 1990. p. 414–31.