87a

Animation of Geometric Algorithms: A Video Review

Edited by Marc H. Brown and John Hershberger

June 6, 1992
Reformatted for electronic distribution on March 18, 1993

Animation of Geometric Algorithms: A Video Review

Edited by Marc H. Brown and John Hershberger

June 6, 1992

Reformatted for electronic distribution on March 18, 1993

Abstract

Geometric algorithms and data structures are often easiest to understand visually, in terms of the geometric objects they manipulate. Indeed, most papers in computational geometry rely on diagrams to communicate the intuition behind the results. Algorithm animation uses dynamic visual images to explain algorithms. Thus it is natural to present geometric algorithms, which are inherently dynamic, via algorithm animation.

The accompanying videotape presents a video review of geometric animations; the review was premiered at the *1992 ACM Symposium on Computational Geometry*. The video review includes single-algorithm animations and sample graphic displays from “workbench” systems for implementing multiple geometric algorithms. This report contains short descriptions of each video segment.

Preface

This booklet and the accompanying videotape contain animations of a variety of computational geometry algorithms. Computational geometry has existed as a field for almost two decades, and interactive systems for producing algorithm animations have been available for nearly a decade. A collection like this one is overdue.

The ten segments in this tape cover a wide range of algorithms and represent a wide range of software systems. There are animations of fundamental algorithms, such as convex hulls, triangulations, and Voronoi diagrams; there are algorithms that helped shape the field in the early and mid-80's, such as topological sweep and optimal line-segment intersection; and there are recently developed algorithms, such as minimax triangulations. There are single-algorithm animations, general purpose algorithm animation systems, and geometric workbenches. There are sequential algorithms and parallel algorithms. And there is some humor.

We received eighteen submissions from fourteen different institutions. We thank the members of the Video Program Committee for their help in evaluating the entries. Besides the editors, the members of the committee were Marshall Bern (Xerox PARC), Leo Guibas (DEC SRC), Jack Snoeyink (University of British Columbia), and Seth Teller (UC Berkeley).

The ten video segments are independent of each other. We suggest that you use the timings and sequence numbers in the table of contents to view segments of particular interest to you. For example, teachers of introductory computational geometry courses might find the fundamental algorithms illustrated in segments $\boxed{\textbf{2}}$ and $\boxed{\textbf{8}}$ particularly helpful. Seminars and individual researchers might be more interested in getting an intuitive look at more recent algorithms in segments such as $\boxed{\textbf{1}}$, $\boxed{\textbf{3}}$, and $\boxed{\textbf{6}}$.

We believe that algorithm animation has a powerful role to play in communicating the ideas of geometric algorithms; the accompanying videotape illustrates some of this power. We hope this video review will inspire others to implement and animate their own algorithms, so that animation will become increasingly common in computational geometry. Most of all, we hope you will enjoy the material as much as we enjoyed putting it together.

Marc H. Brown
John Hershberger

Table of Contents

Real-Time Closest Pairs of Moving Points

Simon Kahan

Max-Planck Institut für Informatik
Saarbrücken, Germany

$\boxed{\mathbf{1}}$

Tracking moving objects is a fundamental component of many real-time systems applications. This videotape illustrates an algorithm that solves the special case in which the number of objects is so large that continual tracking of all objects is impractical, and an algorithm that computes the closest-pair of point objects.

Overview

Input to the static closest pairs problem is given by the positions of N points in d-space, and output is a list of pairs of points whose distance is minimal over all pairs. In the on-line closest pairs problem, the number of points changes with arbitrary insertion and deletion operations that, along with closest pair queries, form an input sequence of requests that must be satisfied upon arrival, with no look-ahead.

In this presentation, we address a real-time closest pair problem. More specifically, our problem falls into the class of Data in Motion problems introduced in [22] and further developed in [23] and [14]. The input is given by N points, just as in the static problem, but these points are forever changing with time and correspond to positions of objects external to the physical computational device. Consequently, current positions are not readily available within the computer: instead, each position is acquired via an explicit *update* operation per object. Each update is executed as part of the real-time program. An update operation may be viewed as a call to an oracle or sensor that extracts the desired position from the world in which the objects are moving. In effect, the program chooses its input as a subset of the continuously changing positional data.

Because updates require an excursion from nominal processing, we assume they consume a significant amount of real time; certainly far more than that required by a single RAM operation. An important effect is that all real positions are constantly changing, so those acquired via previous update operations and stored in memory soon become inaccurate, or *stale*. In order for the problem to be meaningful, we assume that a bound on the maximum rate of positional change is known; thus, stale positions approximate true positions. In fact, a point must reside within a ball having radius equal to the product of the rate bound and the time since the point's last update, and centered about the stale position; the ball is known as the point's *free-range*. More general assumptions on positional uncertainty and update costs are described in [21].

The desired output of our real-time closest pair problem is the continual identification of a true closest pair. The output is viewed not as a single discrete entity as in the static problem, nor as a series of separate query responses as in the on-line problem, but instead as a discrete-valued function of real-time. Real-time is discretized into *frames*, each having constant duration.

The goal is to output a function that matches the identity of the true closest pair in as many frames as possible. A major constraint is that updates consume real time, so only a few can be issued per frame. A strategy is needed that chooses these updates based on stale data so as to identify the closest pair with as much confidence as possible. No matter what update strategy is chosen, errors are inevitable in the worst case of all positions nearly coinciding.

The theory of data in motion provides a method for evaluating update strategies in a framework similar to that just described. Instead of computing the percentage of errors committed, which may depend on specifics of the object motion, the method compares (1) the number of updates performed by the strategy before the closest pair can be correctly deduced from stored data and rate bounds, to (2) the minimal number required in order to *prove* the answer is correct. Although comparison by difference is a possibility, the ratio of these two quantities turns out to be a more convenient performance measure for the closest pairs problem. A strategy's worst case ratio – defined as the maximum over all possible memory configurations and current positions – is proposed as a good indicator of the strategy's performance, and is called the strategy's *lucky ratio*. This is analogous to the competitive ratio in on-line algorithms theory. The smaller the lucky ratio, the better the strategy. Or so the theory goes …

The accompanying video illustrates three strategies for the closest pair problem: LRU, Greedy, and Pairs. The LRU strategy is to update the least recently updated position. This is sensible in that it guarantees a minimal upper bound on how stale the stored positions can become. However, the theory predicts that the LRU strategy will perform poorly: its lucky ratio is N, the highest the ratio can ever be.

The Greedy strategy makes use of the known rate bound. It identifies a pair of free-ranges whose boundaries are closest over all pairs, and then updates either of the corresponding positions. Since the free-ranges are spheres whose centers are just the stale positions stored in memory and whose radii are easily computed, identifying the closest pair of free-ranges amounts to the problem of locating the closest pair of spheres. The free-range of the updated position collapses to a point. The process is repeated until the stored positions are sufficient to determine the closest pair with certainty. The Pairs strategy is very similar to the Greedy strategy: it differs only in that it updates *both* positions corresponding to the closest pair of free-ranges. The lucky ratio of the Greedy strategy turns out to be N, just as for the LRU strategy, while the Pairs strategy has a lucky ratio of 2, which is optimal [23].

Besides contradicting our intuition, as explained in the video, there are several respects in which the Data in Motion abstraction is an imperfect model of our real-time closest points problem: Lucky ratios are a worst-case measure, and worst-case situations may occur infrequently in practice. Therefore, a strategy having a smaller lucky ratio might in fact be less efficient in empirical tests. In addition, the ratio is based on the relative number of updates per query, not the relative number of errors committed. Also, the abstraction deals with a single isolated instance of a closest pair query, whereas in the real-time problem the closest pair is to be maintained at all times: what happens when there is not time to do all the necessary updates? What if the closest pair is obvious with no updates? The former issue may result in errors; the latter is a matter of implementation: the Greedy and Pairs implementations perform updates in LRU fashion when the closest pair is already determined based on stale data.

The software used to produce the simulations in the video is true to the real-time paradigm. Yet the main result – that the Pairs strategy is optimal – jibes with experiment: in all simulations performed thus far, under a variety of kinds of motion, the most accurate strategy is consistently the Pairs strategy. This corroboration suggests that the Data in Motion theory is worth consideration when faced with real-time motion problems.

Production Information

The video simulation was created by a C program built upon the Simple Raster Graphics Package (SRGP) available from Brown University and ran on a SUN SPARC-station. The spaghetti source code is available from the author upon request. Filming was off the screen via an ordinary camcorder in a dark room. Attempts to directly record the image electronically did not substantially improve the clarity.

Acknowledgments

Thanks to Joan Lawry for help with the frustrating, on-location filming.

The XYZ GeoBench: Animation of Geometric Algorithms

Peter Schorn Adrian Brüngger Michele De Lorenzi

Institut für Theoretische Informatik
ETH, CH-8092 Zürich

$\boxed{\textbf{2}}$

This videotape shows the XYZ GeoBench (eXperimental geometrY Zürich) [26, 29, 30], a workbench for geometric computation. XYZ GeoBench provides an interactive front-end on the Macintosh computer to the XYZ Program Library that contains many standard algorithms for 2-d problems and some for higher dimensional problems. The execution of all algorithms can be animated.

Contents of the video

The video shows some animations of standard algorithms and a brief demonstration of the experimental use of the workbench. In particular we show the computation of the convex hull, finding a closest pair, and computing all-nearest-neighbors-to-the-left. (An animation of traveling salesman heuristics, alluded to in the video, was omitted for reasons of length.)

Convex hulls: The first animation shows Graham's incremental algorithm [17] while the second animation shows Preparata and Hong's divide and conquer method [27].

Proximity problems: We present a plane sweep for computing the closest-pair [19] and a plane-sweep for computing for each given point a nearest neighbor-to-the-left [20].

Experimenting with the XYZ GeoBench: The final demonstration shows how the GeoBench can be used interactively to approximate the Voronoi-diagram of a scene of convex objects. The objects (line segments, circles, a hexagon) are entered using the mouse, then they are covered with evenly spaced points and the Voronoi-diagram of this point set is computed using Fortune's sweep [13]. The result gives a good impression of how the Voronoi-diagram of the more complex objects would look like.

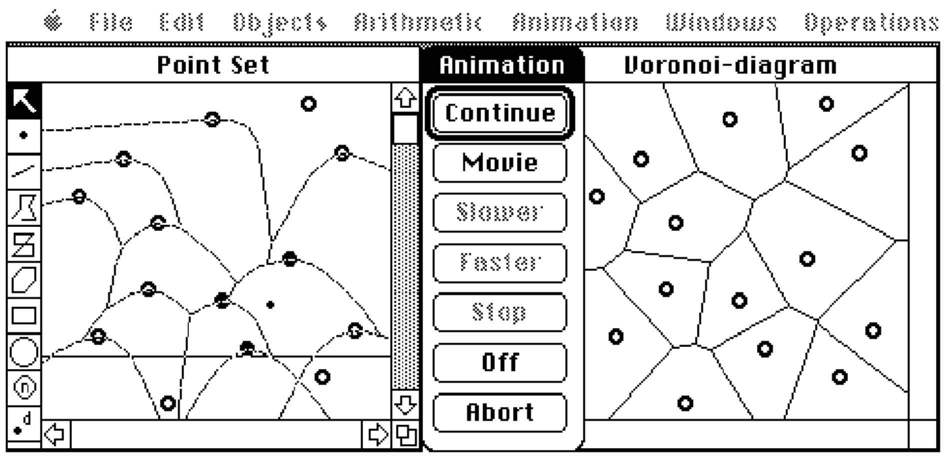
Algorithm animation in the XYZ GeoBench

Algorithm animation is used for demonstrating and debugging. We have chosen a simple, yet powerful, approach to animation. There is only one version of an implementation into which code pertaining to the animation is included via conditional compilation. This code checks whether animation for this particular algorithm is turned on. If yes, it directly updates the currently visible state of the algorithm and waits for the user to let it proceed. Finding a graphical representation of a geometrical algorithm's state is usually simple, since geometric objects have standard graphical representations. The GeoBench supports the drawing, highlighting and flashing of all geometric objects in a uniform way which lets the implementor easily create animations. Animation code has the following general structure:

```
...
{Geometric algorithm changing internal state.}
{$IFC myAlgAnim }
{Conditional compilation: animation code can be removed easily for higher efficiency.}
  if animationFlag[myAlgAnim] then {check whether animation is turned on}
    {Update graphical state information, usually draw some objects.}
    waitForClick(animationFlag[myAlgAnim]); {display dialog box}
    {Update graphical state information, usually erase some objects.}
  end;
{$ENDC }
...
```

The procedure `waitForClick` provides the only interface between the user and the algorithm currently animated. It supports single step mode and a movie mode with user selectable speed (see the "Animation" dialog box in the screen dump below. Updating the visualization of the internal state is facilitated by the convention that all drawing on the screen is done using XOR graphics which has the benefit that erasing is the same as drawing. Animating an algorithm consists of choosing a representation of the internal state (e.g. position of the sweep line, objects in the y-table, deactivated objects, etc.) and determining appropriate locations in the program where this information needs to be updated. Algorithm animation is implemented for all nontrivial geometric algorithms.

The following screen dump shows the XYZ GeoBench while animating the computation of a Voronoi diagram using Fortune's sweep [13]:



Production of the video

The video signal of a Macintosh IIfx was fed into a scan converter which was connected to a BetaCam video recorder. The titles were created using MacroMind Director on the Macintosh. Production time was about three man-days.

System Availability

The XYZ GeoBench system, including documentation, is available via anonymous ftp from `neptune.inf.ethz.ch` (129.132.101.33) in directory `XYZ`.

Optimal Two-Dimensional Triangulations

Herbert Edelsbrunner Roman Waupotitsch

Department of Computer Science
University of Illinois at Urbana-Champaign

3

This video illustrates the MINMAX*er system.* MINMAX*er implements various versions of the edge-insertion paradigm for computing optimal two-dimensional triangulations. It also includes code for the edge-flip heuristics that can be used to obtain locally optimal triangulations.* MINMAX*er was originally designed for educational purposes and to study the efficiency of algorithms based on the edge-insertion paradigm. It can also be used for a comparative study of the quality achieved by various other triangulations introduced in the literature.*

Overview

The visual interface offers two main options. First, it allows the user to follow the progress by showing insertions and deletions of edges in the wire-frame representation. A second option colors the triangles according to their respective measure. With this option one can get a global impression how and how fast the quality of the triangulation improves during the iteration.

We briefly review the edge-insertion paradigm. The purpose of the method is to find the triangulation that minimizes the maximum measure over all possible triangulations of a given point set. Such measures are for example the largest angle or the slope of a triangle. The algorithm starts with an arbitrary triangulation $\mathcal{T}$ and iterates until no improvement is possible. A single iteration adds a new edge to the triangulation. All edges that intersect this new edge must of course be deleted. The resulting polygons are now retriangulated. Delicate details of this scheme can be found in [2, 12].

The implementation supports the case where the original triangulation contains constraining edges. Furthermore, it lexicographically optimizes the entire vector of measures, not just the worst one. Because of this property it usually computes a unique optimum.

For the implementation we use integer arithmetic with ad hoc tie-breaking rules. The triangulations are stored using so the called quadedge data structure. The visual interface uses the GL graphics library on a Personal Iris Workstation.

Acknowledgments

We thank Ping Fu, Karl Hess, Susan Goode, and Robin Bargar at NCSA for their help in producing this video. The second author thanks Ernst Mücke for his help and advice concerning the graphics interface. At this point we also like to mention Tiow Seng Tan's contribution who was the first to see that the edge-insertion paradigm can indeed compute optimal triangulations.

Funding for this research was provided in part by the National Science Foundation under grant CCR 89-21421.

Boolean Formulae for Simple Polygons

John Hershberger Marc H. Brown

DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

4

An animation of an algorithm in action is a powerful tool for exploring the algorithm's behavior. It has proven to be helpful in teaching computer science courses, designing and analyzing algorithms, producing technical drawings, tuning performance, and documenting programs. As systems for animating algorithms are becoming more powerful and easier for programmers to use, it is becoming increasingly important to identify the techniques that an algorithm animator must use This video illustrates many of the techniques for algorithm animation reported in the literature thus far [4, 5], especially recent results related to the use of color.

Contents of the video

The algorithm in the videotape finds a representation of a simple polygon's interior as a monotone Boolean combination of the halfplanes determined by its edges [9]. A *simple polygon* is a closed polygonal path, free of self-intersections; a *monotone Boolean combination* is a Boolean formula containing only unions ("+") and intersections ("*")—no negations are allowed.

The rest of this note itemizes some of the important algorithm animation techniques to note while watching the videotape.

Multiple views. It is more effective to illustrate an algorithm with several different views than with a single monolithic view. A monolithic view concentrates all the

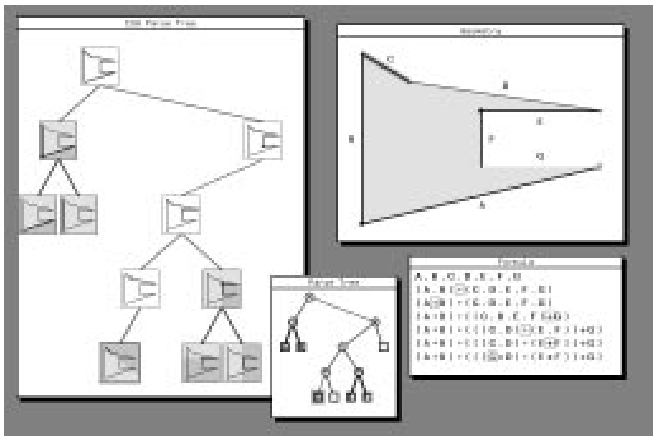


Figure 1

information about an algorithm into one dynamic image. However, to depict a complicated algorithm in detail, or multiple aspects of even a simple algorithm, a single monolithic view must encode so much information that it quickly becomes difficult for the user to pick out the details of interest from the wealth of information in the view. Conversely, when each view displays a small number of aspects of the algorithm, each view is easy to comprehend in isolation, and the composition of several views is more informative than the sum of their individual contributions. The views in the figures display the polygon itself, the Boolean formula and its development, and the parse tree corresponding to the formula.

Static history. Especially when animation is used to explain an unfamiliar algorithm, it is helpful to present a static view of the history of the algorithm and its data structures. Such a view is similar to the way an example might be presented in a textbook; it allows the user to become familiar with the behavior of the algorithm at his own speed, and to focus on the crucial events where significant changes happen, without paying too much attention to repetitive events.

In Figure 1, the Formula view shows the development of a Boolean formula over time, as parentheses and operators are added. The CSG Parse Tree view on the left also embodies a static history: it displays the planar region corresponding to every subformula ever constructed during the algorithm. The Parse Tree view in Figure 2 is a compact version of the same tree that omits the displayed regions.

Amount of input data. It is instructive to introduce an animation on a small problem instance, preferably with textual annotation, to relate the visual displays to the user's previous understanding. With small amounts of data, items can be explic-

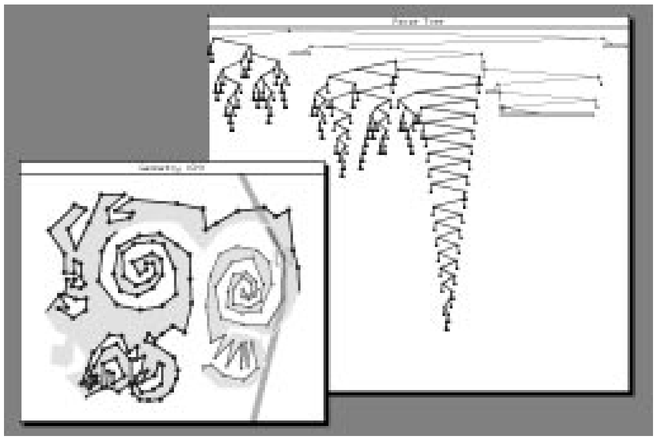


Figure 2

itly labeled, and the user can easily understand the connections between the views (see Figure 1). Once these connections are established, it is appropriate to introduce larger, more interesting data sets in which the dynamic capabilities of the animation are more fully utilized (see Figure 2). Of course, information such as labels must be hidden when displaying these large problem instances, since they would clutter the view unnecessarily.

Pathological input data. It is often instructive to choose pathological data to push an algorithm to extreme behavior. For example, in the video the algorithm is run using both perfectly convex polygons and tight spirals as input. Each input produced a characteristic parse tree (balanced or skewed). When the algorithm is run on less contrived data, we can easily pick out the unbalanced subtrees of the parse tree corresponding to the spirals of the input polygon.

Color unites multiple views. When multiple views show different aspects of the same data structure, or different representations of logically related objects, an application can create a smoother, more harmonious picture by painting corresponding features with the same colors in all the views. The polygon decomposition animation uses the colors blue, red, and black to denote objects that have been, are being, or have not yet been processed, respectively. This idea is applied uniformly; combined with the visual prominence of the color red, this makes it easy to see the connection between the active edges of the polygon and the corresponding active sites in the formula and parse tree views.

Color reveals an algorithm's state. As mentioned, red, blue, and black are used to indicate the state of a vertex. In general, as an indicator of algorithm state, color enhances and complements other techniques in at least three ways. First, it gives an extra dimension for state display—one can encode information in the color of objects, as well as in their shape, size, texture, etc. Second, it allows denser presentation of information: fewer pixels are needed to make a color change visible than to make a change in the shape of an object visible. Third, color is good for displaying global patterns. For example, if a group of small black dots changes to a mixture of red and blue dots, it will be much easier to perceive global patterns than if, say, the circular dots changed to a mixture of black triangles and squares.

Color highlights areas of interest. By temporarily painting a small region with a transparent, contrasting color, an algorithm can focus attention on the painted area. Because the highlight color is transparent, it does not interact visually with the data elements on the screen, but simply draws the eye to them. A second use of highlighting is to display transient computations without permanently altering the on-screen state. For example, in Figure 2 (but not in the video), a brown highlight shows a convex hull that is an essential part of the algorithm, but which changes too rapidly to belong to the relatively stable state displayed in the rest of the view.

Color emphasizes patterns. In Figure 2, each deep subtree in the right view grows downward at the same time as the highlighted vertex runs inward along one of the spirals in the left view. The colors of the subtrees and the spirals also change in concert. The kinetic connection between the two views underlines the linkage between spirals and deep parse tree subtrees.

SHASTRA: A Distributed and Collaborative Design Environment

Chandrajit L. Bajaj

Department of Computer Science
Purdue University
West Lafayette, IN 47907

5

State of the art in Computer Aided Geometric Design (CAGD) is still a single user, single workstation, monolithic environment. At Purdue we are developing a research prototype software environment called SHASTRA where multiple users (say, a collaborative engineering design team) create, share, manipulate, simulate, and visualize complex geometric designs over a distributed network of workstations and supercomputers.

System Overview

SHASTRA is a highly extensible, collaborative, distributed geometric design and manipulation environment. At its core is a powerful collaboration substrate to support multi-user applications, and a distribution substrate which emphasizes distributed problem solving.

Under the umbrella of Project SHASTRA, we have developed software toolkits GANITH, SHILP and VAIDAK. The GANITH algebraic geometry toolkit manipulates polynomial (i.e. algebraic) equations in any number of variables. The SHILP modeling and display toolkit manipulates curved solid objects with algebraic surface boundaries. The VAIDAK medical imaging and model reconstruction toolkit manipulates medical image volume data.

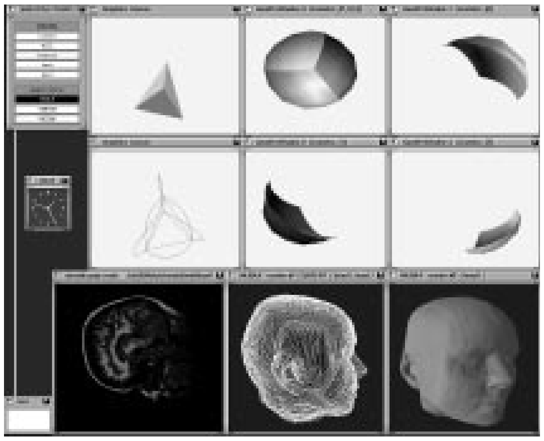


Figure 3

Though these toolkits run as independent processes with separate user interfaces, they share a common infrastructure of numeric, symbolic, and graphics algorithms. The toolkit processes link to each other and communicate data structures (images, polynomials, solids, etc.) via inter-process communication facilities using an XDR-based protocol.

SHASTRA provides these systems with connection management and data communication facilities enabling component systems to use facilities and operations provided by sibling systems, effectively integrating them into a large scientific manipulation system. It also provides them with a collaboration substrate to support cooperative and collaborative design.

Video Demonstration

Figure 3 depicts a smooth, three dimensional model of a human head which was interactively reconstructed by a team of three, working collaboratively on networked SHASTRA applications. The designers first collaborate on reconstructing a single polyhedral approximation of the head from given MRI image date using communicating VAIDAK applications. The polyhedral model is then communicated to and shared by three instances of SHILP, one per user. Users work on disjoint portions of the head and interactively smooth the polyhedral approximation by replacing them with piecewise algebraic surface patches via simultaneous remote calls to multiple GANITH instances which support interpolation and least-squares approximation operations. The polyhedral smoothing operation has been implemented as a fully distributed algorithm so that a SHILP user can simultaneously instantiate a GANITH process and a remote call, separately for independent faces of the polyhedral surface. Substantial speedup is obtained for complex designs via this distributed parallelism.

The video shows an animation of the above computation sequence. Note in particular the integration of SHILP, GANITH and VAIDAK.

Implementation Notes

All geometric computations and animation were performed in the SHASTRA distributed and collaborative geometric design environment. The distributed toolkits of SHASTRA run on any UNIX workstation which support X-11. The animation on the video tape was done using a network of SPARCstations and Personal Irises. Scan conversion from RGB to NTSC and recording on S-VHS tape was achieved by an in-house Panasonic video editing system.

Contributors

The major contributors in the development of the various applications under the SHASTRA umbrella are:

Graduate Students: Vinod Anupam, Steve Cutchin, Jindon Chen, Tamal Dey (Ph.D. August 1991), Insung Ihm (Ph.D. August 1991), Kunihiko Okamura, Andrew Royappa.

Undergraduate Students: Brian Bailey, Andrew Burnett, Daniel Schikore.

Acknowledgments

Funding for this project was provided in part by NSF grants CCR 90-00028, DMS 91-01424, AFOSR contract 91-0276, and funds from AT&T and from NSC.

Tetrahedral Break-Up

Leonidas Palios Mark Phillips

The Geometry Center
University of Minnesota
1300 South Second Street
Minneapolis, MN 55454

6

The accompanying videotape contains an animation of an algorithm by Chazelle and Palios to tetrahedralize a nonconvex polyhedron, that is, partition it into tetrahedra [8].

The Algorithm

The algorithm works in two phases. The goal in the first phase is to reduce the size of the polyhedron to be proportional to the number of its reflex edges. To achieve this, we identify vertices whose incident edges all exhibit interior dihedral angles that do not exceed π, and remove cone-shaped pieces of the polyhedron with those vertices as the apexes. As the operation looks very much like pulling a ski hat off someone's head, this phase is called *pull-off* phase. In the second phase, the *fence-off* phase, vertical fences are erected through each edge of the polyhedron partitioning it into cylindrical pieces. Each piece can be defined by (i) specifying a horizontal base polygon, (ii) lifting it vertically into an infinite cylinder, and (iii) clipping the cylinder between two planes (which do not intersect inside the cylinder). The triangulation of the base polygons of these pieces and the addition of the vertical fences of the new edges leads to a refined partition into cylindrical pieces whose bases are triangles; each such piece can then be easily decomposed into at most three tetrahedra.

The algorithm partitions a polyhedron of n vertices, r reflex edges and zero genus into $O(n + r^2)$ tetrahedra, which is asymptotically optimal in the worst case (see [6]). It runs in $O((n + r^2) \log r)$ time, and requires $O(n + r^2)$ space. In the process, $O(r^2)$ Steiner points are introduced. (Note that the problem of deciding whether a polyhedron can be partitioned into tetrahedra when no Steiner points are allowed is NP-complete [28].) The computed partition into tetrahedra is not a cell complex; the algorithm can, however, be slightly modified to produce a cell complex without altering the stated time complexity and size of the partition.

The Animation

The key tool in the production of the animation is *Geomview*, a general-purpose 3D viewing program developed at the Geometry Center. *Geomview* can display geometric objects that change under the control of an external program, while allowing the user to control the viewpoint and other aspects of the display.

The first step in the animation was to use an existing C program which implements the described algorithm to decompose the example polyhedron. We then wrote a C program to choreograph the data; this program generated geometric and motion description statements which were passed to *Geomview* via a Unix pipe. The camera location remained under interactive user control which allowed us to focus on the area of interest without having to program camera motions in advance.

Production Information

The animation was recorded in real time on a Silicon Graphics VGX workstation with a Sierra Video Systems RGB to SVHS transcoder to generate video output.

System Availability

Geomview is available on the Internet via anonymous ftp from `geom.umn.edu` (128.101.25.31). Email inquiries may be sent to `software@geom.umn.edu`.

Acknowledgments

This work was supported by the National Science Foundation, the Department of Energy, Minnesota Technology, Inc., and the University of Minnesota.

Compliant Motion in a Simple Polygon

Joseph Friedman

D. E. Shaw & Co.
120 W. 45th Street
New York, NY 10036

7

Compliant motion *is a motion paradigm that's particularly suited for robots with limited sensing and control uncertainty. Under the simplest variant of the compliant motion model, a robot following a commanded direction α travels through free space in direction α until it encounters an obstacle. Then it slides along the obstacle until either friction is too large, or α no longer points into the obstacle. In the former case, it stops; in the latter, it resumes travel through free space in direction α. This type of motion enables the robot to "grope" its way towards the goal [10, 24].*

The compliant motion planning problem *is that of determining, for a given environment and a starting and goal position, all the commanded directions that can take a robot from the starting position to the goal (the* good directions*), if any such directions exist.*

Overview

This video presents an animation of a compliant motion planning algorithm by Friedman, Hershberger, and Snoeyink [15]. Given a fixed environment and a goal vertex in the environment, the algorithm analyzes the environment, breaking it up into regions. The regions are formed in such a way that all the starting points in the same region have similar good directions. This reduces the compliant motion planning problem to the well-studied point location problem.

When the environment is a fixed simple polygon and the goal is a fixed vertex of the environment, the good directions for any starting point form a single interval, bounded by a *start* direction and a *stop* direction. We produce two subdivisions of the environment; we use the *start* subdivision to determine the start direction, and the *stop* subdivision for the stop direction. The regions in each subdivision are triangles and trapezoids.

In order to produce the regions, we work our way backwards from the goal and follow the changes in the α-*backprojection*, which is a subset of the environment containing all the starting points for which α is a good direction. As α rotates, the α-backprojection gains and loses regions, and we accumulate these regions in the appropriate subdivision. We compute the regions only for directions α at which the α-backprojection undergoes major changes. Such directions are called *events*, and can be attributed to either a vertex or an edge of the environment.

The Video

The video starts with the first phase of the algorithm on a simple polygonal room. The goal is the bottom corner of the room. As the direction α (indicated by the blue arrow in the center) rotates, the bottom copy of the room shows the corresponding α-backprojections. Events are indicated by the flashing edge or vertex that caused them, and for each event we generate new regions in the start subdivision (the green subdivision in the top left copy of the room) or the stop subdivision (the red subdivision in the top right copy).

When the direction α completes a full turn, the preprocessing is complete, and the video concludes with three queries for the initial position of the robot. For every query point, we consult the start and stop subdivisions for the corresponding bound of the good direction interval, and then we highlight the path followed by a robot commanded by the middle direction of the good interval to prove that the robot can indeed reach the goal.

Production Information

We used a Silicon Graphics IRIS 240 GTX workstation to generate the video. The animation program is approximately 6,500 lines of C (including an interactive environment editor), and utilizes Silicon Graphics' GL$^{\mathrm{TM}}$ software. Because of the high rendering speed of the IRIS GTX, we were able to record the video in real time.

Acknowledgments

I wish to thank my friends John Hershberger and Jack Snoeyink, who collaborated with me on the compliant motion research. I would also like to thank the following individuals from Silicon Graphics for helping with the video. Reuel Nash and Jim Winget graciously gave us permission to use SGI facilities. Mike Schulman and Tim Heidmann helped with the audio visual equipment.

Workbench for Computational Geometry

P. Epstein J. Kavanagh A. Knight J. May T. Nguyen
J.-R. Sack

Computational Geometry Workbench Project
School of Computer Science
Carleton University, Ottawa
Canada K1S 5B6

8

In this video we illustrate a few of the algorithms that have been developed using our Workbench for Computational Geometry. The Workbench is not primarily an algorithm animation system, but is designed as a general geometrical programming environment, providing tools for: creating, editing, and manipulating geometric objects; demonstrating and animating geometric algorithms; and most importantly, for implementing and maintaining complex geometric algorithms.

Video Contents

The video first shows several triangulation algorithms. The importance of triangulations for a variety of practical applications is well known. In addition to those significant applications, triangulation has become an important tool for other problems in computational geometry. The triangulation algorithms demonstrated are:

- Triangulation via "ear removal"
- Triangulation of monotone polygons [16]
- Triangulation of simple polygons via monotone decomposition [16]
- Triangulation of simple polygons via efficient trapezoidation [33]

- Randomized triangulation [31]

The second part of the video shows algorithms based on triangulation. A variety of problems can be solved in linear time, once a triangulation is known. Among these are shortest path problems:

- Point-to-Point Shortest Path [25]
- Shortest Path Tree from a given Point [18]

and link distance problems [32]:

- Construction of window tree
- Execution of a link query

Animation Facilities

In the Workbench, animation of an algorithm is the responsibility of the algorithm implementor. Code to perform the animation is added, using a simple and well-defined protocol, to the implementation. This approach considerably simplifies the animation system, but has a cost in flexibility. In particular, the system does not currently support different animations of the same algorithm or multiple views in an animation. Animation is simplified considerably by its restriction to the domain of computational geometry. The workbench is an object-oriented system, dealing primarily with geometric objects that have a clear graphical representation. Many animations can be represented by simple manipulations of the same objects used in computation. This model can easily be extended to arbitrary abstract data types by defining an appropriate visual representation, but to date our animations are purely geometric.

The code for animating an algorithm consists mainly of commands to add a “frame” for a particular geometric object or set of objects. These frames are rendered and stored as Macintosh PICTs, which improves speed and memory usage for rendering.

Animating an algorithm usually affects its time and space requirements. This must not be allowed to affect algorithms which are not being animated. The workbench accomplishes this by constructing *blocks*, unevaluated sections of code, which are passed to the animation. If animation is not active, these are discarded with a very small constant overhead. If animation is active, they are evaluated to produce a frame. Thus the same code can be used for animated and non-animated versions without modification or recompilation.

A separate process controls display of the animation, allowing animation to be simultaneous with the computation. This process renders each frame as it is sent, as well as storing it for later replay. The animation process is user-controllable, with commands modeled after a tape recorder, such as *play*, *record*, *cue*, *fast-forward*, and *stop*.

The animation system was designed to allow control of animation depth for algorithms which make use of other algorithms as intermediate steps. This is done by maintaining a hierarchy of animations. When one algorithm calls another, a sub-animation is constructed, containing the animation of the called algorithm. In general, the result of executing a complex algorithm defines an *animation tree*. The depth of the animation tree defines the *depth of the animation*. The depth can be controlled by the user during creation and play-back by using the *deeper* and *shallower* commands. This allows a user to view execution of an algorithm at an appropriate level of detail.

Some algorithms may require interaction with the user *during* algorithm execution. For example, a shortest path algorithm might ask the user for the source and destination of the path. We would expect this request to be made during algorithm execution but not during play-back of the animation. The animation system supports such input/output interactions.

Production Information

This video was recorded using a Macintosh II equipped with a NuVista+ video card, whose output was connected directly to a video recorder.

Acknowledgments

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, Carleton University, and the University of Passau.

Topologically Sweeping an Arrangement: A Parallel Implementation

Marc H. Brown Harald Rosenberger

DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

9

This videotape shows the workings of a topological *sweepline, as it visits the $O(n^2)$ intersections in an arrangement of n lines in the plane.*

Overview

An ordinary sweepline is a vertical line that visits the $O(n^2)$ intersections in an arrangement of n lines in the plane by sweeping across the arrangement from left to right. Such a sweepline uses only $O(n)$ working storage, but, because it sorts the intersections in x-order, it spends $O(n^2 \log n)$ time. In many cases the sorting is unnecessary; it is enough just to visit all the intersections in any order. A *topological* sweepline visits the intersections in optimal $O(n^2)$ time by sacrificing the straightness of the ordinary sweepline, while retaining the $O(n)$ space bound. [11].

In between visiting intersections, the topological sweepline crosses n edges of the arrangement—an upper and a lower edge from each convex face it crosses. The active edges—those crossed by the sweepline—are shown in red in the "Sweep Line" view. The light blue edges have already been swept, and the thin black edges remain to be swept. The black dots show intersections that the sweepline could visit next. The sweepline can choose arbitrarily which black dot to advance over next, or can even advance over all of them in parallel. The lower and upper "Horizon" views display the data structures that the algorithm uses to identify intersections that it can visit next.

Implementation Notes

The animation was implemented in an in-house dialect of Modula-2 using the Zeus algorithm animation system [3], and runs on a 5-processor Firefly workstation. Details about the techniques used in developing this animation (e.g., the use of color and the choice of input data) are available elsewhere [4].

The New Jersey Line-Segment-Saw Massacre

Ayellet Tal Bernard Chazelle David Dobkin

Department of Computer Science
Princeton University
Princeton, NJ 08544

10

This videotape shows a line segment intersection algorithm in action and illustrates its most important features. The algorithm, due to Chazelle and Edelsbrunner [7], has an optimal running time of $O(n \log n + k)$, where n is the number of line segments and k is the number of pairwise intersections.

Overview

As in the classical Bentley-Ottmann method [1], the Chazelle and Edelsbrunner [7] algorithm operates in a sweepline fashion by scanning the segments from left to right, and maintaining the vertical visibility map of the region swept along the way. Two important differences are that (i) the schedule includes only the endpoints of the segments and not the intersection points, and (ii) the cross section along the sweepline is maintained in a lazy fashion, meaning that the nodes of the tree representing the cross section might correspond to segments stranded behind past the sweepline. Also, the loop invariant for the sweepline is not simply that the portion of the map left of it should be maintained but also that the map associated with all the segments intersecting the sweepline be available as well. Segments are cut up into smaller pieces in preprocessing, so as to enforce a normalization condition related to the schedule of insertions.

Production Notes

The animation comes from a system currently being developed at Princeton by Ayellet Tal and David Dobkin. This system is intended to ease the interface between geometric code and the graphics device. It is built on top of Cheyenne, a device independent graphics library developed by David Dobkin and Eleftheros Koutsofios. The program runs on Sun and Silicon Graphics IRIS workstation. Recording was done at the Interactive Computer Graphics Lab at Princeton and editing was done with the assistance of the Princeton Department of Media Services.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant Number CCR90–02352.

References

[1] J. L. Bentley and T. A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.

[2] M. Bern, H. Edelsbrunner, D. Eppstein, S. Mitchel, and T. S. Tan. Edge insertion for optimal triangulations. In *Proc. 1st Latin American Sympos. Theoret. Informatics*, pages 46–60, 1992.

[3] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proc. IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[4] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. In *Proc. IEEE Workshop on Visual Languages*, pages 10–17, 1991.

[5] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, January 1985.

[6] B. Chazelle. Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM Journal on Computing*, 13:488–507, 1984.

[7] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[8] B. Chazelle and L. Palios. Triangulating a nonconvex polytope. *Discrete and Computational Geometry*, 5:505–526, 1990.

[9] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. *Computer Graphics*, 22(4):31–40, 1988.

[10] B. R. Donald. Error detection and recovery for robot motion planning with uncertainty. Technical Report MIT-AI-TR 982, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1987.

[11] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *J. Comput. System Sci.*, 38:165–194, 1989.

[12] H. Edelsbrunner, T. S. Tan, and R. Waupotitsch. An $O(n^2 \log n)$ time algorithm for the minmax angle triangulation. In *Proc. 6th Ann. Sympos. Comput. Geom.*, pages 44–52, 1990.

[13] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[14] P. G. Franciosa, C. Gaibisso, and M. Talamo. Optimal algorithms for the maxima set problem for data in motion. Technical Report RAP. 03.92, Dipartimento di Informatica e Sistemistica, Università Degli Studi di Roma "La Sapienza", February 1992.

[15] J. Friedman, J. Hershberger, and J. Snoeyink. Compliant motion in a simple polygon. In *Proceedings of the 5th ACM Symposium on Computational Geometry*, pages 175–186, 1989.

[16] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Inf. Process. Lett.*, 7(4):175–179, 1978.

[17] R. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[18] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[19] K. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26:255–261, 1988.

[20] K. Hinrichs, J. Nievergelt, and P. Schorn. An all-round sweep algorithm for 2-dimensional nearest-neighbor problems. *Acta Informatica*, 1990. To appear.

[21] S. H. Kahan. Real-time tracking strategies. In preparation.

[22] S. H. Kahan. A model for data in motion. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, 1991.

[23] S. H. Kahan. *Real-Time Processing of Moving Data*. PhD thesis, University of Washington, 1991.

[24] J.-C. Latombe. *Robot Motion Planning*. The Kluwer international series in engineering and computer science, SECS 0124. Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[25] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[26] J. Nievergelt, P. Schorn, M. De Lorenzi, C. Ammann, and A. Brüngger. XYZ: A project in experimental geometric computation. In *Proc. Computational Geometry – Methods, Algorithms and Applications '91*, pages 171–186. Springer, 1991. LNCS 553.

[27] F. Preparata and S. Hong. Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM*, 20(2):87–93, 1977.

[28] J. Ruppert and R. Seidel. On the difficulty of tetrahedralizing 3-dimensional nonconvex polyhedra. In *Proc. 5th Annual ACM Symposium on Computational Geometry*, pages 380–392, 1989.

[29] P. Schorn. *Robust Algorithms in a Program Library for Geometric Computation*. PhD thesis, ETH, Zurich, Switzerland, 1991.

[30] P. Schorn. The XYZ GeoBench: A programming environment for geometric algorithms. In *Proc. Computational Geometry – Methods, Algorithms and Applications '91*, pages 187–202. Springer, 1991. LNCS 553.

[31] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comp. Geom.: Theory and Appl.*, 1:51–64, 1991.

[32] S. Suri. *Minimum Link Paths in Polygons and Related Problems*. PhD thesis, The Johns Hopkins University, 1987.

[33] R. E. Tarjan and C. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17:143–178, 1988.