ELSEVIER

Computational Geometry 12 (1999) 125–152

Computational Geometry
Theory and Applications

Visualizing geometric algorithms over the Web ☆

James E. Baker [a,1], Isabel F. Cruz [b,2], Giuseppe Liotta [c,3], Roberto Tamassia [a,*]

[a] *Center for Geometric Computing, Department of Computer Science, Brown University, Providence, RI 02912–1910, USA*
[b] *Computer Science Department, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609–2280, USA*
[c] *Dipartimento di Ingegneria Elettronica e dell'Informazione, Universita' di Perugia, Via G. Duranti 93, 06131 Perugia, Italy*

Communicated by M.T. Goodrich; submitted 23 June 1997; accepted 30 January 1998

Abstract

The visual nature of geometry applications makes them a natural area where visualization can be an effective tool for demonstrating algorithms. In this paper we propose a new model, called *Mocha*, for interactive visualization of algorithms over the World Wide Web. Mocha is a distributed model with a client-server architecture that optimally partitions the software components of a typical algorithm execution and visualization system, and leverages the power of the Java language, which has become the standard for distributing interactive platform-independent applications across the Web. Mocha provides high levels of security, protects the algorithm code, places a light communication load on the Internet, and allows users with limited computing resources to access executions of computationally expensive algorithms. The user interface combines fast responsiveness with the powerful authoring capabilities of hypertext narratives.

We describe the architecture of Mocha, show its advantages over previous methods, and present a prototype that can be accessed by any user with a Java-enabled Web browser. The Mocha prototype has been widely accessed over the Web, as demonstrated by the statistics that we have collected, and the Mocha model has been adopted by other research groups. Mocha is currently part of a broader system, called *GeomNet*, which performs distributed geometric computing over the Internet.

Keywords: Geometric algorithm visualization; World Wide Web; Client-server architecture; Visual interface; Java; Multimedia

☆ Research supported in part by the National Science Foundation under grant CCR-9423847, CAREER award IRI-9896052, and grant CISE-9729878, by the U.S. Army Research Office under grant DAAH04-96-1-0013, by the N.A.T.O.-C.N.R. Advanced Fellowships Programm, and by EC ESPRIT Long Term Research Project ALCOM-IT under contract No. 20244.

* Corresponding author. E-mail: rt@cs.brown.edu

[1] E-mail: jib@cs.brown.edu.

[2] E-mail: ifc@cs.wpi.edu. Work by this author was performed in part at Brown University.

[3] E-mail: liotta@dis.uniroma1.it. Work by this author was performed in part at Brown University.

0925-7721/99/$ – see front matter
PII: S0925-7721(98)00038-8

1. Introduction

Algorithm visualization and animation appeal to the strengths of human perception by providing a visual representation of the data structures and by allowing for smooth image transitions between time-related visual representations that correspond to different states of the execution of the algorithm. The end-user can understand algorithms by following visually their step-by-step execution, otherwise a complex task if relying on the textual program alone. Extensive work has been done on algorithm animation and program visualization [8–11,14,19,22,26,36,40,45,46]. Interactive algorithm visualization is also a powerful tool to demonstrate new algorithms to others in an intuitive, often appealing fashion, which is therefore highly suitable for pedagogical use either by students individually or in class demonstrations [5,13,43,52]. Visualization can also be used as a debugging tool for large software applications [39].

The visual nature of geometry applications makes them a natural area where visualization can be an effective tool for communicating ideas. This is enhanced by the observation that much research in computational geometry occurs in two and three dimensions, where visualization is highly plausible. Given these observations, it is not surprising that there has been noticeable progress during the past few years in the production of visualizations of geometric algorithms and concepts (see, e.g., [1,15, 29,31,48,49] and the collections of videos accompanying the proceedings of the ACM Symposium on Computational Geometry since 1992). An extensive survey by Hausner and Dobkin on the visualization of geometric algorithms is also available [25].

There is every reason to believe that work on the visualizations of geometric algorithms will continue and even accelerate in the future [50]. There are many technical challenges in doing algorithm visualization and animation. For example, the development of a conceptual framework to modularize and simplify the design process (part of the *authoring* process), for which Stasko proposes the *path-transition paradigm* [44], 3D visualization [12,41,38], and automatic graph drawing [16,17]. Another challenge, which extends beyond algorithm animation, is the visualization of large sets of data [35]. As pointed out in [25], a major issue in the visualization of geometric algorithms is the realization of interactive animations, where the user has direct control over the input to the algorithm (e.g., through a geometric editor integrated with the animation system), the visualization (e.g., by selecting a 3D view and/or a 2D projection), and the timeline of the animation (e.g., through controls that allow the viewer to travel forward and backward in time).

A new challenge relates to making animation and interactive visualization available to a broad audience, in particular in a distributed computing platform such as the World Wide Web. Previous approaches use the X Window system and, more recently, the distributed and platform-independent programming environment associated with the Java language.

The *X Window* system [42] provides a basic client-server mechanism for algorithm animation over the Internet, which we shall call the *X model*. Namely, an animation program running on a remote machine can interact with the X server on the local user's machine (display) by opening there a window, sending graphic output (display requests) and receiving the user's keyboard and mouse actions (display events). While this mechanism is simple to implement, there are significant security problems associated with remote X sessions, and the communication load placed on the Internet is quite high.

The Java language [23] and its environment opens the possibility of embedding interactive applications in HTML documents, which are executed through the Web browser on the user machine after their code has been transferred. Java has become a *de facto* standard for distributing interactive platform-

independent applications across the Web. It is object-oriented, and therefore easily extensible, while providing a wide-range of authoring capabilities given its interface with the Web. In addition, Java has a variety of built-in security features that protect both the user and the provider of the animation. Java provides an attractive possibility for animations provided that the user's machine is powerful enough to run a sophisticated animation algorithm, and that the provider of the animation does not mind sharing the code with the rest of the world.

To overcome the problems inherent in the X and the Java model, we propose a new model, called *Mocha*, for interactive algorithm visualization over the World Wide Web. Mocha is a distributed model with a client-server architecture, which, given a list of criteria that we establish, optimally partitions the software components of a typical algorithm visualization and animation system, and leverages the power of the Java language. In the Mocha model, only the interface code is exported to the user machine, while the algorithm is executed on a server that runs on the provider's machine.

Mocha provides high levels of security (which are inherent to Java), protects the algorithm code, places a light communication load on the Internet, and allows users with limited computing resources to access animations of computationally expensive algorithms. The user interface combines fast responsiveness with the powerful authoring capabilities of hypertext narratives.

Our vision for Mocha is in fact a part of the even broader vision of *GeomNet* [4], a system for performing distributed geometric computing over the Internet. GeomNet consists of a family of *geometric computing servers* that execute a variety of geometric algorithms on behalf of remote clients, which can be either users interacting through a Web browser interface, or application programs connecting directly through sockets. GeomNet provides a variety of services including algorithm execution, algorithm animation, consistency checking of topological and geometric structures, experimental study and comparison of algorithms, and electronic commerce for "metered" services.

In Section 2, we describe the Mocha model and show its advantages over the X model and the Java model for algorithm animation over the Internet. In Section 3, we present a prototype of an animation system for geometric algorithms that can be accessed by any user with a Java-enabled Web browser at URL `http://www.cs.brown.edu/cgc/demos/Mocha.html`. Browsers supporting Java include Netscape Navigator, Microsoft Internet Explorer, and Sun's HotJava. Details of the design, architecture and implementation of Mocha are provided in Section 4.

2. Models for algorithm visualization over the Internet

In this section, we examine the currently used mechanisms for providing algorithm visualization over the Internet, and present our new architecture.

2.1. Components of an algorithm visualization system

Following the event-driven approach advocated by Brown [9] and the conceptual framework pioneered by Stasko [44–46], we view interactive algorithm visualization as an event-driven system of communicating processes: the *algorithm* augmented with annotations of interesting events, called algorithm operations, and the interactive visualization component that provides the multimedia visualization of the algorithm operations. We further subdivide the interactive visualization component

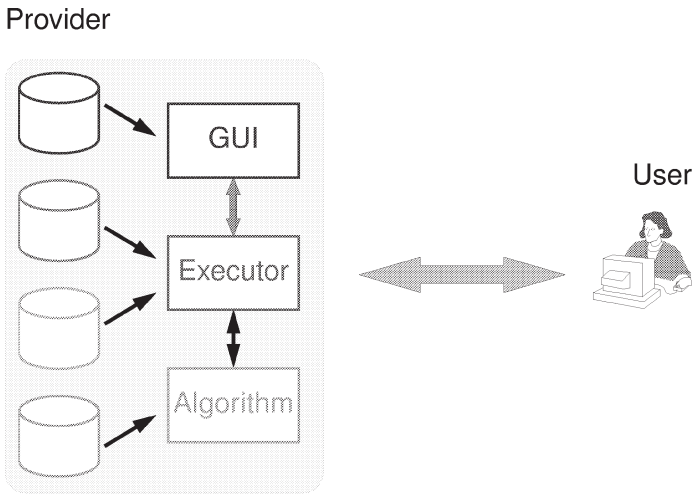


Fig. 1. Components of an interactive system for visualizing and animating algorithms.

into the *GUI* (graphical user interface), which handles the interaction with the user, and the *executor*, which maps algorithm operations and user requests into dynamic multimedia scenes.

The interaction of the user with the components of an interactive algorithm visualization system is illustrated in Fig. 1.

2.2. Comparison criteria

We use the following criteria to evaluate models for algorithm visualization over the Internet, where a distinction is made between the roles of the *user* of the visualization and of the *provider* of the visualization.

Security. Both the user and the provider should be protected from intruder attacks.

Code protection. The provider's code should be protected from possible software piracy.

Authoring. It should be easy for the provider to create new visualizations and make existing visualizations available on the Internet.

Communication complexity. The communication load on the Internet should be kept as light as possible.

Accessibility. Users with limited computing resources should be able to access visualizations of computationally expensive algorithms.

2.3. The X model

The *X Window* system [42] provides a basic client-server mechanism for algorithm visualization over the Internet, which we shall call the *X model*. Namely, an interactive visualization program running on a remote machine can interact with the X server on the local user's machine (display) by opening there a window, sending graphic output (display requests) and receiving the user's keyboard and mouse actions

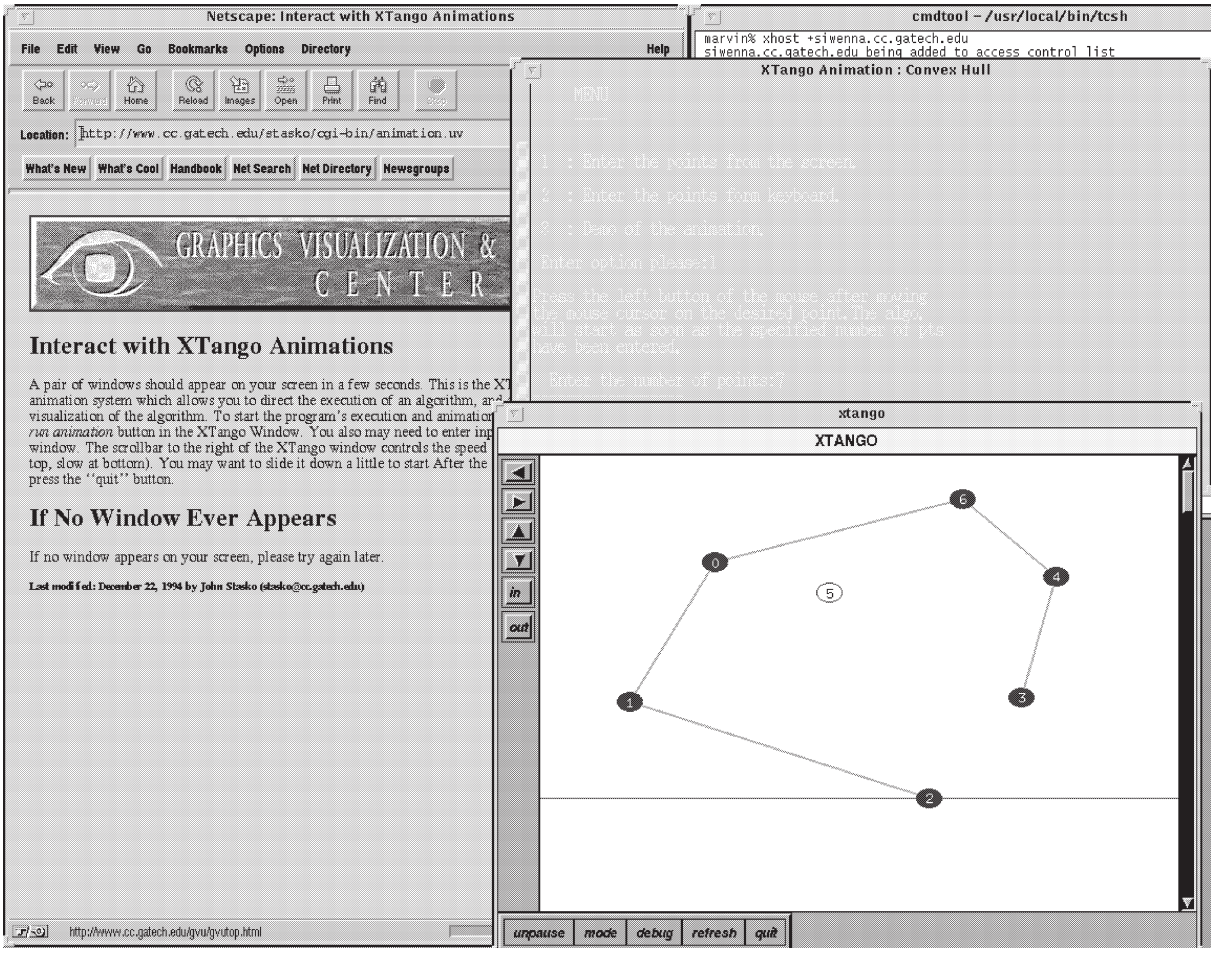


Fig. 2. X model example: a visualization of a convex hull algorithm using the XTango system.

(display events). For example, this mechanism is used to provide on-line demonstrations of the *XTango* visualization system [46], illustrated in Fig. 2, where the visualization program can be activated by a `cgi-bin` script (see, e.g., [24]) launched in a Web browsing session.

The X model of algorithm visualization is schematically illustrated in Fig. 3. The three main functional components of the visualization system (GUI, executor, and algorithm) reside on the remote machine of the provider. All communication is performed with the X system protocol.

The X model fully protects the entire visualization code, which is not revealed to the user. Authoring for the provider is relatively easy, since all the algorithmic, GUI, and visualization computations are performed in the machine of the provider. However, the provider cannot easily create a new visualization by re-using existing libraries unless they have been locally installed. Accessibility for the user is high, since the X model allows users to display complex visualizations on computers that are not very powerful.

The main drawbacks of the X model are security and communication complexity. There are significant security problems associated with the X model, both for the user and the provider. The user must give

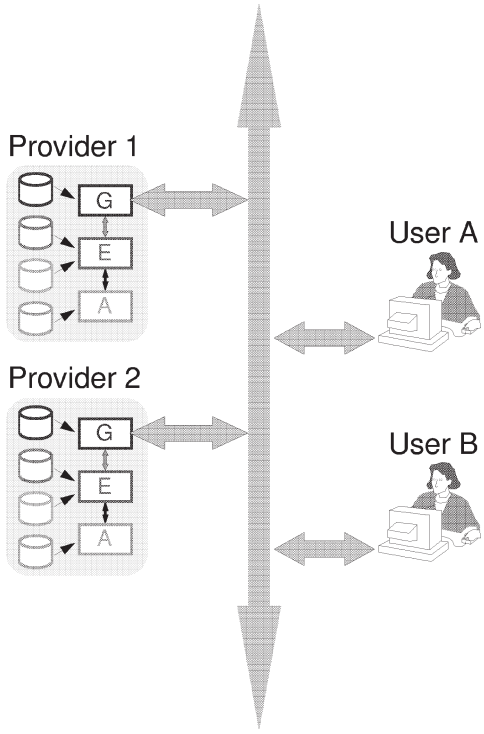


Fig. 3. Algorithm visualization in the X model. The vertical arrow denotes the Internet. The horizontal arrows denote the communication between the end-user and the remote X server. These arrows are drawn thick to indicate that the communication load of the X model is high. The users' computers are drawn small to indicate that low computation power is needed at the users' sites in the X model.

access to the visualization program with an `xhost +` command (or `xauth` authentication procedure), and hence allows a "Trojan horse" visualization program to manipulate the windows on the user's machine. Conversely, the provider allows the user to execute a program on its machine, and must take special precautions to prevent attacks from malicious users.

In the X model, the communication between the remote GUI and the user (display events and requests) is carried by the Internet. Low level graphic primitives (including those associated with mouse click and drag actions) are transported over the Internet. Hence, visualizations with complex dynamic scenes send large amounts of data through the Internet, which makes poor use of the Internet bandwidth and slows down the interaction with the user.

2.4. The Java model

Java [2] is an object-oriented language designed to be dynamically redistributable over the Internet, especially in conjunction with the World Wide Web. The Java virtual machine provides a uniform set of services across all platforms, such as graphics display, multithreading, and distributed objects, and it loads classes stored in a portable byte code as needed [33]. In the Web environment, these classes are generally transported to the Web browser through the HTTP protocol for execution on

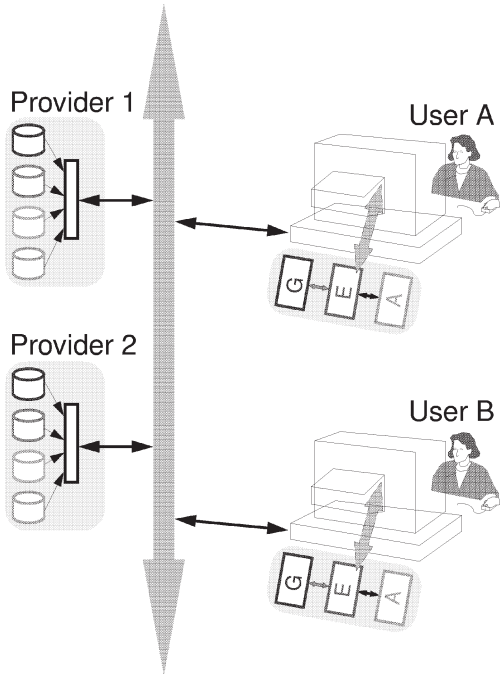


Fig. 4. Algorithm visualization in the Java model. The thin arrows indicate that the communication complexity of the Java model is low. The large computers of the users indicate that high computational power may be needed at the users' sites.

the user's machine as an applet contained on a Web page (other mechanisms exist; see, for example, `http://www.marimba.com`).

Java incorporates safety and security into its design as follows:

- no pointer arithmetic;
- checked array bounds;
- automatic garbage collection;
- code authentication of loaded modules;
- byte code running on a virtual machine but otherwise isolated from the user's local environment (this isolation is called the "sandbox" metaphor);
- exception handling;
- name-space management via hierarchical packages.

Although these features are worthy in themselves, they interlock to provide a high assurance of security. For example, the virtual machine provides for code authentication, arranged by the name space packages. Name space packages, exception handling, and lack of pointer arithmetic prevents access to the file system or network, except through privileged (local) packages and only if the user has enabled this right. Automatic garbage collection, no pointer arithmetic, and array bound checking prevents overflow errors that can be used for "cuckoo's egg"-type attacks [47].

Java provides a general framework for interactive applications over the World Wide Web. When specialized to the algorithm visualization domain, we obtain what we call the *Java model* of algorithm visualization, which is schematically illustrated in Fig. 4. The three main functional components of the

visualization system (GUI, executor, and algorithm) reside on the user's machine and receive their Java code and multimedia objects from a server on the provider's machine. All communication is performed with the HTTP protocol. A sorting visualization (inspired by Balsa [9]) that uses the Java model is shown in Fig. 5.

Thanks to its built-in features, the Java model provides a high level of security for both the user and the provider. Also, since the visualization program is transported over the Internet and locally executed on the user's machine, the communication complexity is low.

Authoring in the Java model is only partially satisfactory. On one hand, the provider can easily develop new visualizations by accessing remote repositories of images, sounds and Java code through the World Wide Web. On the other hand, the Java model forces the provider to implement in Java all the components of the visualization system: GUI, executor, and algorithm. The latter can be particularly disadvantageous since the provider may want to use directly existing algorithm libraries (e.g., LEDA [34]).

The main drawbacks of the Java model are code protection and accessibility. The user has full access to the Java byte code; hence, the entire algorithm visualization code is given away to the user in a form that can be "decompiled". Also, the user must have sufficient computing resources to execute the visualization program; hence, effective access to interactive visualizations of computationally expensive algorithms is denied to users with low-power machines.

2.5. The Mocha model

The Merriam-Webster OnLine dictionary (`http://www.m-w.com/`) defines "mocha" as follows:

mo·cha (*mō′kə*) *n* [Mocha, seaport in Arabia] **1a1:** superior arabica coffee with small green or yellowish beans grown in Arabia **1a2:** a coffee of superior quality **1b:** a flavoring made of a strong coffee infusion or of a mixture of cocoa or chocolate with coffee **2:** a pliable suede-finished glove from African sheepskins.

We introduce the *Mocha model* as a strong infusion of the Java programming language and execution environment with a mixture of client-server and framework paradigms for interactive algorithm visualization. Mocha consequently defines an architecture, as well as a design and implementation.

Much of Mocha's strength, like other Internet tools, is that it leverages existing tools, standards, and architectures, which enabled us to quickly deploy a very usable prototype (see Section 3). Our efforts have been directed to the definition of a common visualization protocol that allows a large number of different clients and servers to interact, without incurring large authoring costs to coordinate and maintain this architecture.

The Mocha model employs a novel configuration of software frameworks, client-server partitioning, mediators, and layered protocols to provide openness and extensibility. We further use the new architectural composition capabilities of Java that allow dynamic redistribution of executable code. In this section we present the main features of the Mocha model. A prototype system for animating geometric algorithms that uses the Mocha model is presented in Section 3 (see also Figs. 8–11, 14, 15 and 18). Details of the design, architecture and implementation of Mocha are provided in Section 4.

The Mocha model, which is schematically illustrated in Fig. 6, is a distributed architecture for algorithm visualization, where the algorithm is executed on the provider's machine(s), while the executor and GUI are executed on the user's machine. The above partitioning of the components maximizes ease of authoring and accessibility, protects the algorithm code, and has low communication complexity. The

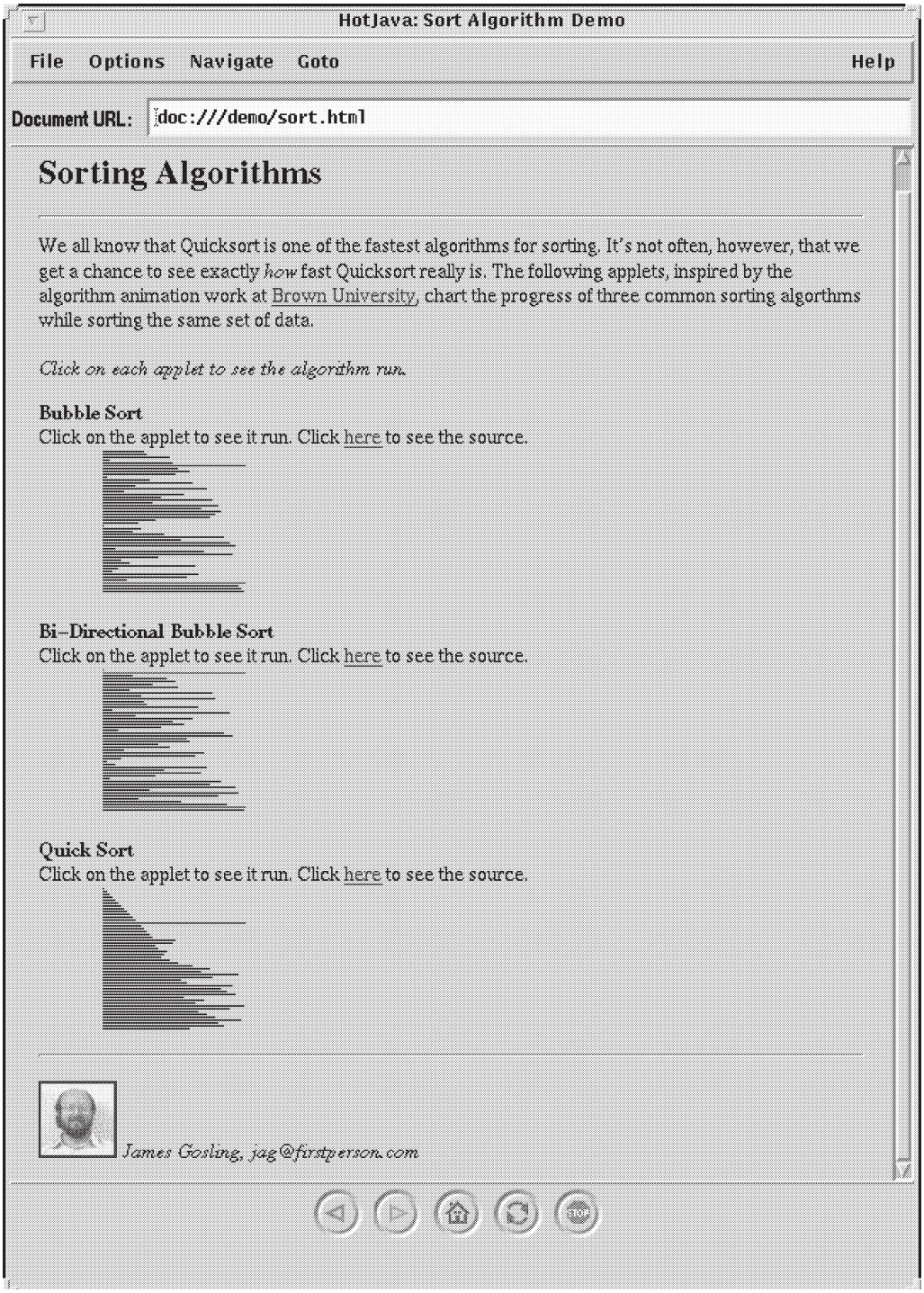


Fig. 5. A sorting visualization in the Java model.

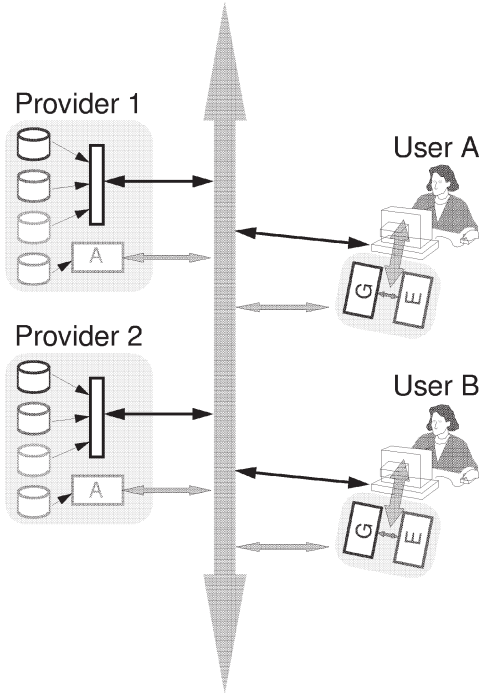


Fig. 6. Algorithm visualization in the Mocha model. The algorithm runs on the provider's computer, while the executor and GUI run on the user's computers. The thin arrows denote the light communication load associated with downloading the Java byte code and multimedia objects for the GUI and executor. The medium-thickness arrows indicate the moderate communication load for the exchange of data between the algorithm and the executor and GUI. The users' computers are drawn small to indicate that low computation power is needed at the users' sites in the Mocha model. Some local computation is requested, involving locally running visualization code and GUI code.

communication between the components over the Internet is achieved through a distributed client-server scheme and is performed with the HTTP protocol and a specific visualization protocol (see Section 4). Also, the security features of Java are used to guarantee security for the user and provider.

The provider employs a first server to send Java byte code and multimedia objects to the GUI on the user's machine by means of the HTTP protocol. The provider also employs another server to execute the algorithm and exchange data (inputs, operations, and control) with the executor and GUI on the user's machine by means of a newly designed visualization protocol, which is transported over Internet sockets.

We employ multithreading in the implementation of the GUI and executor to provide more responsive feedback to the user. Also, we use an object-oriented container/component software architecture that guarantees expandability. Finally, we have mediators that isolate the commonality between the interaction of the clients and servers and provide a high degree of interoperability.

2.6. Comparison

In this section we compare the X model, the Java model, and the Mocha model, with respect to the five quality criteria described in Section 2.2. Table 1 provides a qualitative comparison. An entry with a

Table 1
Comparison of the three models for algorithm visualization. For an explanation of the five criteria, see Section 2.2

	Security	Code protection	Authoring	Communication complexity	Accessibility
X model	−−	++	+	−−	+
Java model	++	−	+	++	−−
Mocha model	++	+	++	+	+

"++" means that the model of the corresponding row matches at best the quality criterion of the given column. In contrast, a "−−" stands for lack of that criterion within the model. Intermediate scores like "+" and "−" are also possible.

For example, both the X model and the Java model have one "+" with respect to their authoring since, as explained in Sections 2.3 and 2.4, both the models are only partially satisfactory with respect to this criterion. For a contrast, the Mocha model matches very well the authoring criterion and thus it is scored with a "++". Notice the versatility of the Mocha model, that for most criteria is as good as the best one of the other two models and never scores "−" in all other cases.

We have performed an experiment aimed at providing a quantitative comparison of the communication load in the Mocha model vs. the X model. The experimental setting is one of a user interacting with the Mocha prototype. We observe that when the Mocha model or Java model is used on a system using X for display operations, the underlying low-level GUI operations for the Web browser are provided by X. In a workstation environment, both the X client (the Web browser) and X server (the display of the Web browser) run locally on the given workstation. By comparing the X stream between the X client and the X server with the Mocha stream between the algorithm and the visualization component (executor and GUI), we can determine the additional cost of using the X model.

The experiment uses the following instrumentation to compare the communication load for an interactive visualization of the Delaunay triangulation (see Section 3.1).

- XScope, a utility provided in the standard X distribution, which monitors the X stream.
- An instrumented algorithm server supporting the Delaunay triangulation visualization.

The instrumentation data consist of the timestamp (to a hundredth of a second resolution) and the number of bytes transmitted. The two data streams are then synchronized by a marker event to align their timestamps. (The synchronization is required because the algorithm server is started before the Web browser.) The results of the experiment are shown in Fig. 7. We have found that even for our simple prototype, which does not maintain state in the client to support incremental updates, the Mocha model outperforms the X model by a factor of about three.

3. Interactive visualization of geometric algorithms with Mocha

In this section we describe a prototype that shows the feasibility of the Mocha model described in Section 2.5. This prototype, called *Mocha Geometry Server* and available on the Web at `http://www.cs.brown.edu/cgc/demos/Mocha.html`, provides visualizations of computational geometry algorithms.

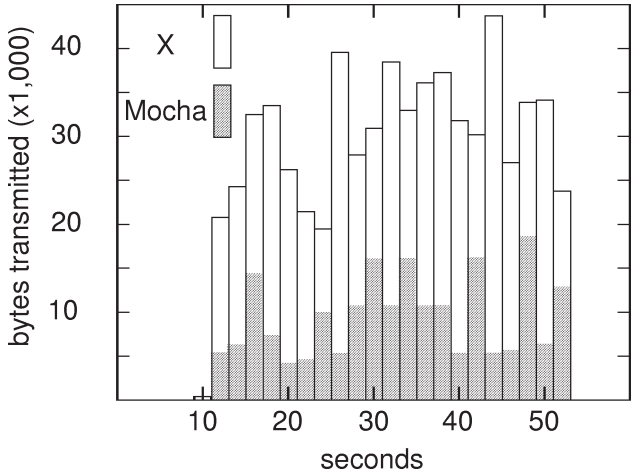


Fig. 7. Experimental comparison of the communication load in the X and Mocha models. The histograms show the sizes of X stream and of the Mocha stream for a typical user interaction involving inserting, deleting, and moving 25 points in the visualization of the Delaunay triangulation.

Geometric algorithms have interesting characteristics. For example, their representation is in a sense less abstract than that of other combinatorial algorithms, such as sorting, and are close to applications such as geographical databases and computer graphics. From the end-user point of view, the main characteristics of the Mocha Geometry Server are its *visual interface* and *interactivity*. The visual interface consists of a hypertextual document where the algorithms are displayed on canvases embodied in the hypertext. Interactivity is obtained by providing real-time responses to operations such as insertion, deletion, or movement of the geometric objects of interest in the application.

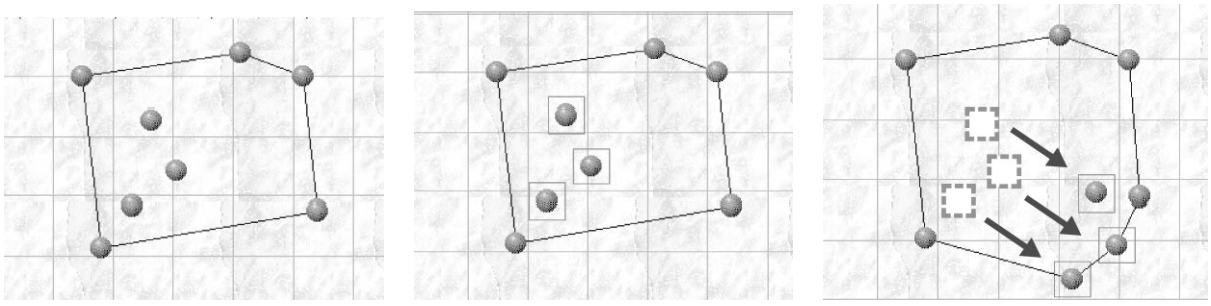
3.1. The LEDA visualization service

LEDA [34] is a C++ library of data structures and algorithms. It includes efficient and robust algorithms for computing the convex hull, the Voronoi diagram, and the Delaunay triangulation of a set of points in the plane. The *LEDA Server* is a component of the Mocha Geometry Server that provides visualizations of the above geometric algorithms in the LEDA library.

The hypertextual interface of the LEDA Server is an HTML document downloaded on the user's computer via the HTTP protocol. Of course, this document is itself but one of many hypertext narratives that could employ these applets. The hypertext interface is structured into three different sections, describing visualizations of convex hulls, Voronoi diagrams, and Delaunay triangulations, respectively. Each section has a canvas supporting standard user-interface facilities, such as zoom-in and zoom-out, addition, deletion and displacement of objects.

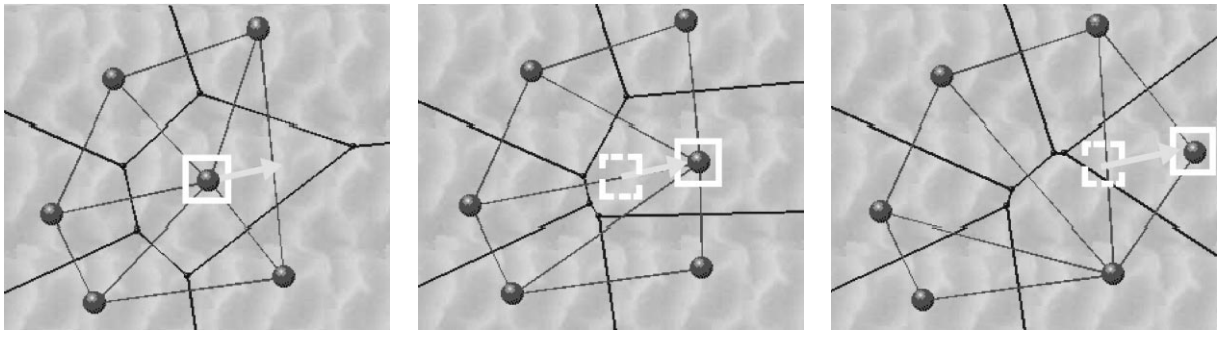
Suppose the user wants to visualize the convex hull computation using the LEDA Server. The visualization is interactive, since the user can perceive in real-time how the convex hull changes when a point is inserted, deleted or moved around in the canvas (see Fig. 8).

Similar interactive visualizations are provided for the LEDA algorithms that compute the Voronoi diagram and the Delaunay triangulation of a set of points in the plane (see Figs. 9–11).



(a) (b) (c)

Fig. 8. Convex hull visualization in Mocha: (a) initial view; (b) selection of three points; (c) dragging the selected point causes the convex hull to be interactively updated.



(a) (b) (c)

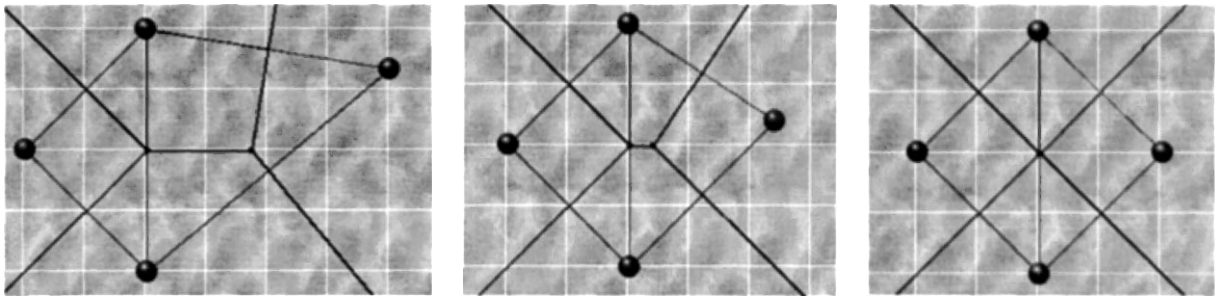
Fig. 9. Voronoi diagram and Delaunay triangulation visualization in Mocha: dragging the selected point causes the Voronoi diagram and Delaunay triangulation to be interactively updated.

Mocha has also revealed itself as a powerful tool for experimenting with existing implementations of geometric algorithms. For example, interacting with Mocha helped us discover a peculiar aspect of the LEDA implementation of Voronoi diagrams. In LEDA, each infinite ray of a Voronoi diagram is replaced by a finite-length segment that ends at a dummy point with "large" coordinates. Also, the dummy points are connected in a polygon by dummy edges. When displaying the Voronoi diagram, the dummy polygon is usually invisible. However, in the example of Fig. 11, the dummy polygon is a triangle and two of its edges become visible when the three sites are almost collinear.

The LEDA Server contains just three of the many geometric algorithms that are available in LEDA. It can be easily extended to include other geometric algorithms, as illustrated in the code fragment of Fig. 12.

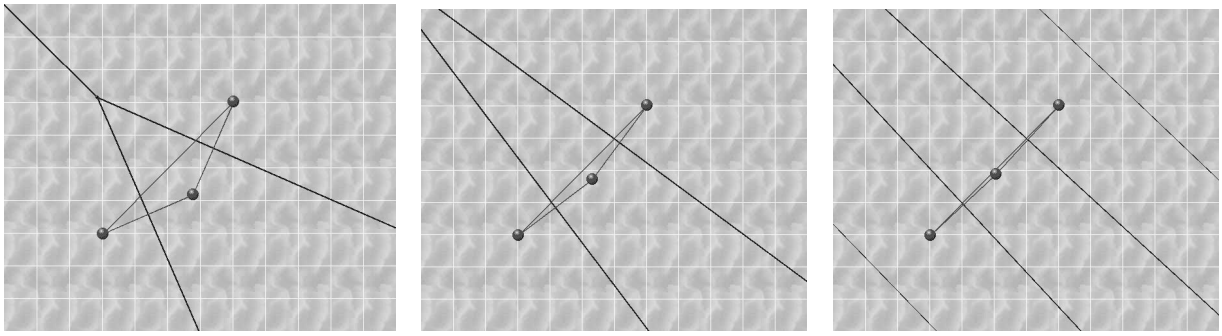
3.2. The proximity visualization service

A *proximity graph* represents a relation on a set of points by connecting with edges pairs of points that are deemed *close* by some proximity measure. A classical way to measure the closeness of two points p



(a) (b) (c)

Fig. 10. Voronoi diagram and Delaunay triangulation visualization in Mocha: dragging a point to create a degenerate configuration with four cocircular points where one of the vertices of the Voronoi diagram has degree 4.



(a) (b) (c)

Fig. 11. Experimenting with the LEDA implementation of Voronoi diagrams and Delaunay triangulations in Mocha: the visualization seems to indicate that an error in the algorithm occurs when dragging a point creates an "almost degenerate" configuration with three points that are almost collinear. Indeed, lines that are not part of the Voronoi diagram appear. The error lies instead in the interpretation of the output data structure, which includes dummy vertices and edges.

and q in a set S is to use a region of the plane associated with p and q, called the *proximity region* of p and q. The points p and q are considered to be close if their proximity region is *empty*, i.e., it does not contain any other point of S. The shape of the proximity region determines the type of graph that results. For example, the *Gabriel region* [20] of p and q is the disk having p and q as antipodal points; the corresponding proximity graph is called *Gabriel graph*. Another example is the *lune* of p and q, defined as the intersection of two (open) disks whose radius is the distance from p to q, with one disk centered at p and the other at q; the corresponding graph is called the *relative neighborhood graph* [51]. See the examples in Fig. 13. For a survey on proximity graphs and on their applications to graph drawing see [18,27].

The *Proximity Server* is a component of the Mocha Geometry Server that provides visualizations of proximity graphs. The user, through the canvas of the hypertextual interface, specifies both the point set S

```
public class Voronoi extends GeometryClient {
   /* Instance data -- set later */
   PointSet ptsetInput;
   Graph graphVoronoi;
   Color colVoronoi;

   /** Create an empty input point set and  Voronoi diagram. */
   public void init() {
      super.init();
      /* init geometry */
      colVoronoi = attribute ("voronoi_color", Color.red);
      ptsetInput = new PointSet();
      ptsetInput.setLimits(0, 0, myGeometryPanel.size().width,
                                 myGeometryPanel.size().height);
      graphVoronoi = new Graph();
   }

   /** Asynchronously runs in another thread, recomputing the Voronoi
       diagram.  This method extends the method in the super class
       GeometryClient, which automatically causes a repaint upon
       recalculation. */
   public void asyncRecalculate () {
      if (ptsetInput.size() > 1)
      {
        /* The super class GeometryClient ties an input stream and an
           output stream to the socket of the server.  Requests for
           computing the Voronoi diagram on the server can be sent on
           pstrmServerOut.  The LEDA implementation requires a value
           for infinity. */
         pstrmServerOut.print ("alg_voronoi 10000 ");
         /* Send the point set */
         ptsetInput.print(pstrmServerOut);

         /* Receive the computed Voronoi diagram when it becomes
            available from the server. */
         graphVoronoi = new Graph (stknServerIn);
      }
      else /* Empty graph in case the point set has shrunk */
         graphVoronoi=new Graph();
      super.asyncRecalculate();
   }

   /** Draws the applet, the input point set, and the Voronoi diagram. */
   public void draw(Graphics g) {
      super.draw (g);
      ptsetInput.draw (g, imRedDot, this);
      graphVoronoi.draw (g, imBlueDot, colVoronoi, this);
   }

   /* We omit glue code for call backs from the geometry editing
      provided by GeometryClient into this child applet. */
   ...
```

Fig. 12. A Java class in the implementation of the LEDA server.

and a nonnegative parameter β that unambiguously defines the shape of the proximity region, called the *β-region*. For a formal definition of β-regions see [28]. We provide here an intuitive description of them (see Fig. 13(d)):

- for $\beta = 1$ the 1-region of p and q is their Gabriel region;

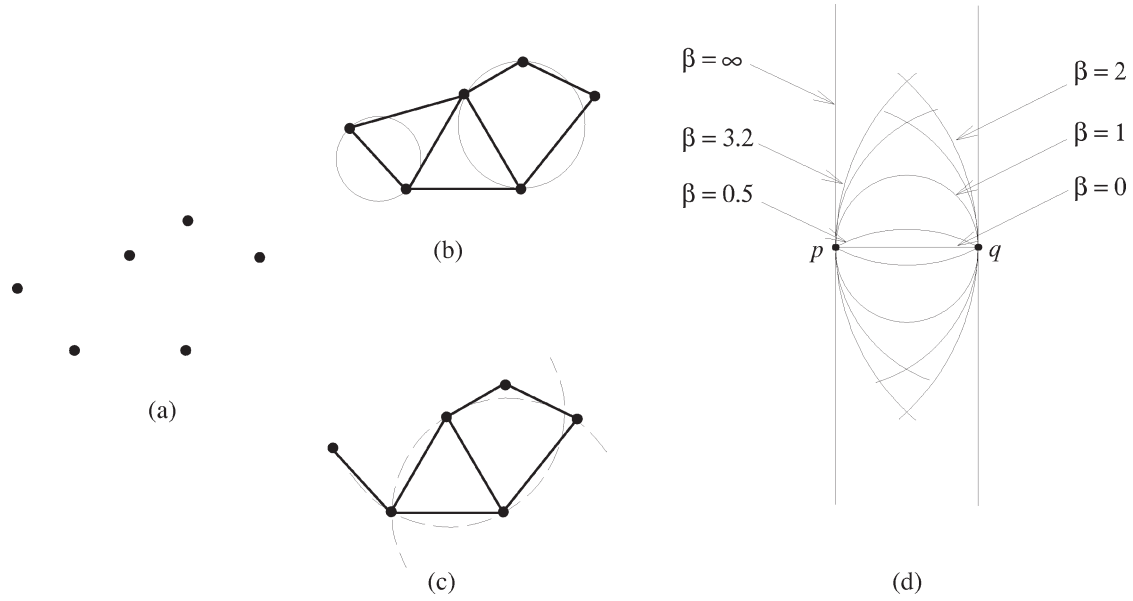


Fig. 13. (a) A point set S. (b) The Gabriel graph of S. (c) The relative neighborhood graph of S. (d) β-regions for two points p and q.

- as β increases the β-region "stretches out" until, for $\beta = \infty$, it becomes the infinite strip orthogonal to the line-segments with p and q as endpoints;
- for $\beta < 1$, the β-region decreases in its size "contracts" until, for $\beta = 0$, it becomes the straight line with endpoints p and q.

The *β-graph* of a point set S is the proximity graph induced by the empty β-regions of all the pairs of points of S. It is easy to see that the β-graph of S becomes dense as the β decreases (if $\beta = 0$ and no three points are collinear, the β-graph is the complete graph), while it becomes sparse as β approaches ∞.

Two proximity graphs visualized by the Proximity Server on the same set of points are shown in Figs. 14 and 15. In the first figure, we have $\beta = 1$ (Gabriel graph); in the second figure we have $\beta = 2$ (relative neighborhood graph). Observe that the proximity graph of the second figure is a subgraph of the one in the first figure.

An advanced interaction technique of the Proximity Server allows the user to define the value of β by sliding a cursor along a logarithmic-scale ruler. The following values of β, which are of special interest in the theory of proximity graphs (e.g., see [6,7]), are explicitly highlighted below the ruler and cause the cursor to "snap" when dragged near them:

$$\beta = \frac{\sqrt{3}}{2}, \quad \beta = \frac{1}{1 - \cos(2\pi/5)}, \quad \text{and} \quad \beta = \frac{1}{\cos(2\pi/5)}.$$

3.3. Prototype usage statistics

For the period January 1997 to May 1998, we have analyzed the usage of the Mocha Geometry Server prototype by counting the number of distinct hits to the main demo page linked to `http://www.cs.brown.edu/cgc/demos/Mocha.html`.

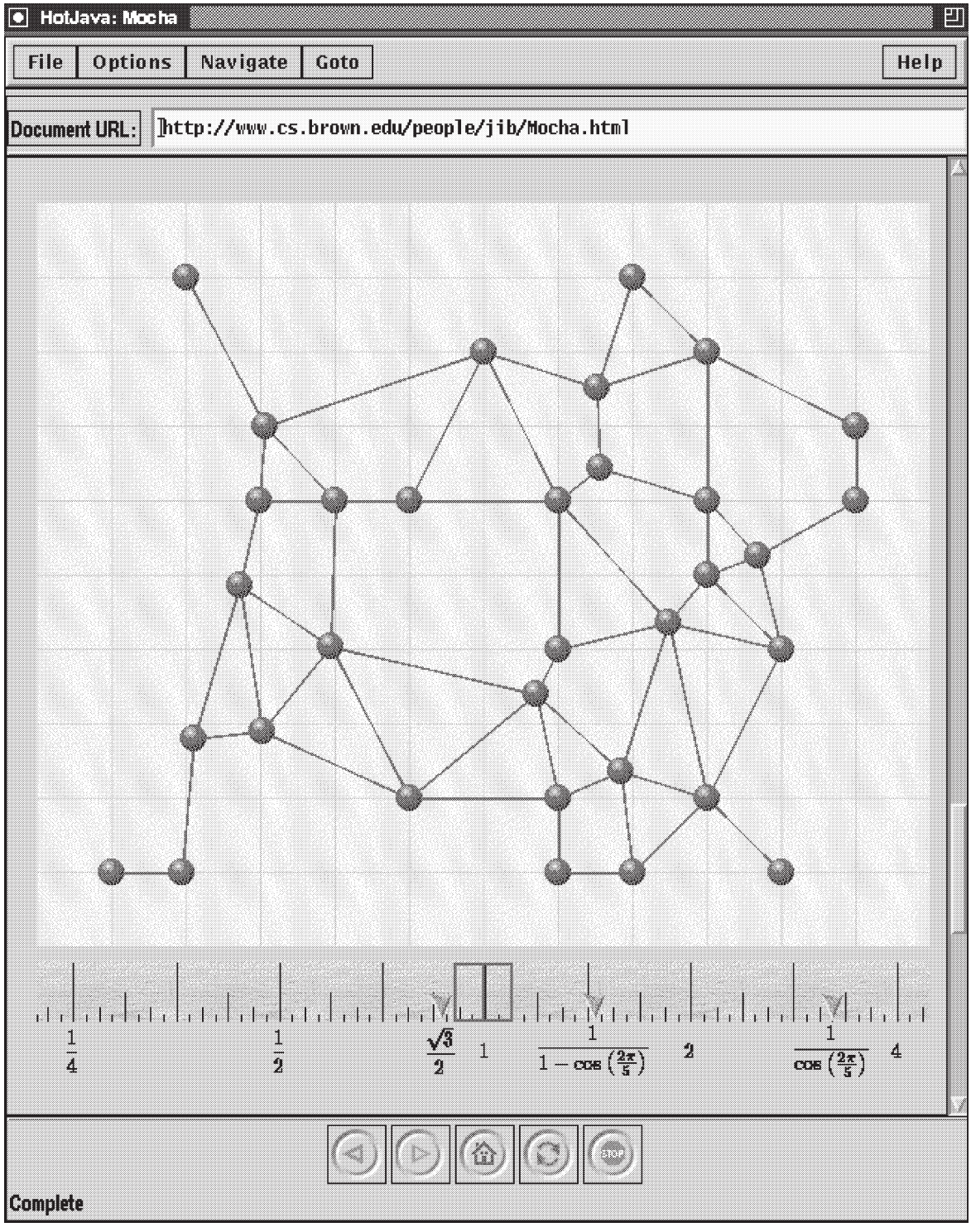


Fig. 14. Visualization of the Gabriel graph of a point set obtained by setting $\beta = 1$.

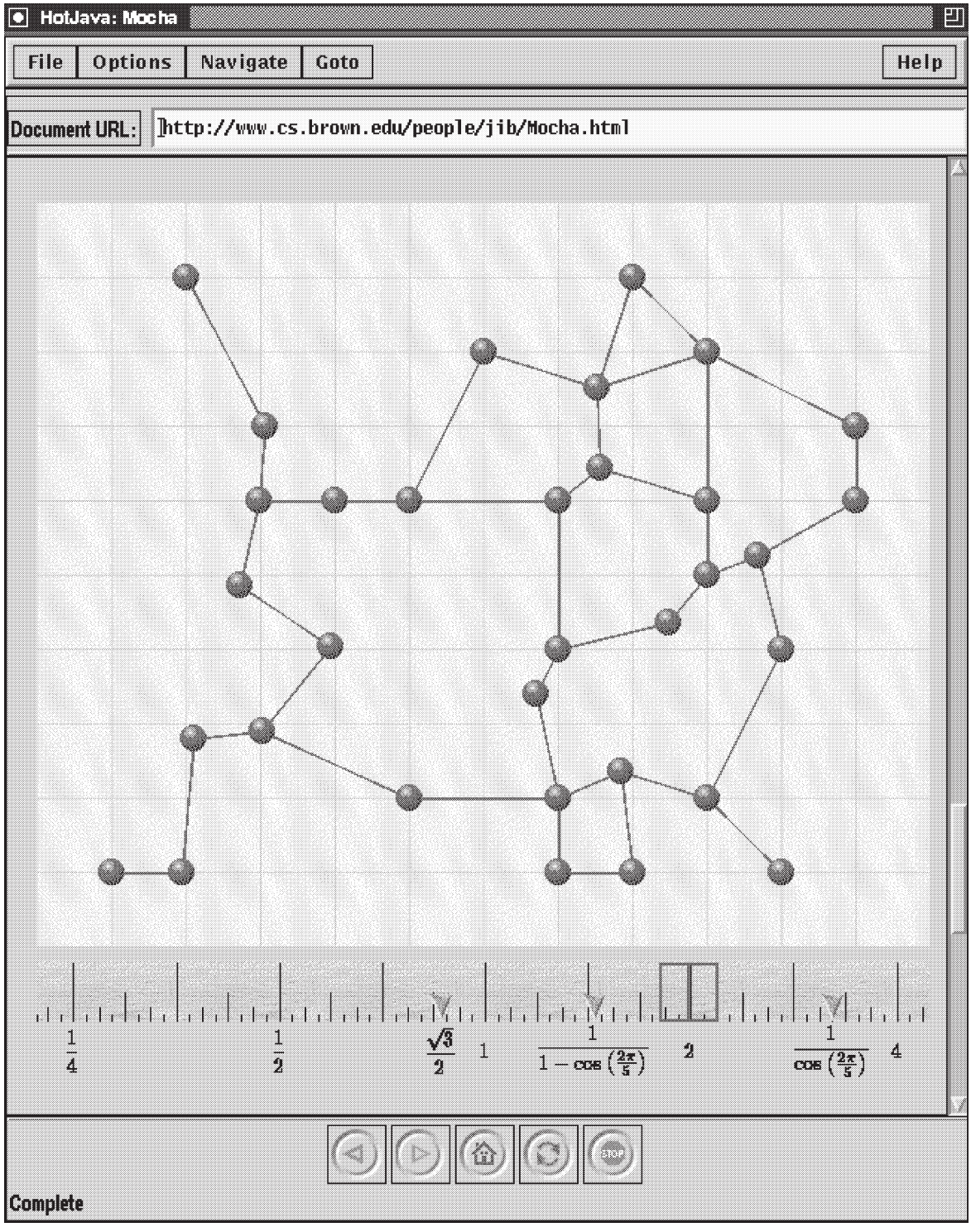


Fig. 15. Visualization of the relative neighborhood graph of a point set obtained by setting $\beta = 2$.

Table 2
Details of the monthly usage of the Mocha Geometry Server prototype

Month	Africa	Asia	Austral	Brown	Europe	N Amer	Num IP	S Amer	Hits
Jan-97		11	2	2	82	78	20	3	198
Feb-97		5	1	8	43	143	17	14	231
Mar-97	2	12	13	18	52	184	49	2	332
Apr-97	4	12	10	35	129	114	33	2	339
May-97		5	2	27	70	99	26		229
Jun-97		7	18	36	84	101	21		267
Jul-97	1	4	2	1	93	90	31	4	226
Aug-97				2	10	26	11		49
Sep-97		13	1	4	70	84	32	6	210
Oct-97		8	5	8	84	95	48	6	254
Jan-98		3	1	7	29	44	8		92
Feb-98		2	11	4	63	64	19		163
Mar-98		8	3	5	40	70	11		137
Apr-98		16	1	8	67	137	30	3	262
May-98		12	2	2	66	102	18	6	208
Total	7	118	72	167	982	1431	374	46	3197

Table 2 shows the details of the monthly usage by continent. We report separately accesses from Brown University and from IP addresses that do not resolve to symbolic domain names in our distributed name service database (DNS). The bar chart of Fig. 16 shows the total monthly number of hits. The pie chart of Fig. 17 shows the overall geographic distribution.

3.4. Other visualization systems based on the Mocha model

The Mocha model has been successfully used by other developers of algorithm animations:

- At Brown, with minimal interaction with the authors of this paper, Mike Walczak created algorithm animations of 1-D and 2-D range trees and segment trees as part of a course project using the Mocha model as depicted in Fig. 18.
- At the Max-Planck-Institut für Informatik, the Mocha model and the "look and feel" of the graphical user interface of the Mocha Geometry Server have been adopted by the developers of the LEDA library to make available over the Web demonstrations of graph and geometric algorithms; see `http://batman.ag1.mpi-sb.mpg.de:22222/LEDA-in-the-Web/`.

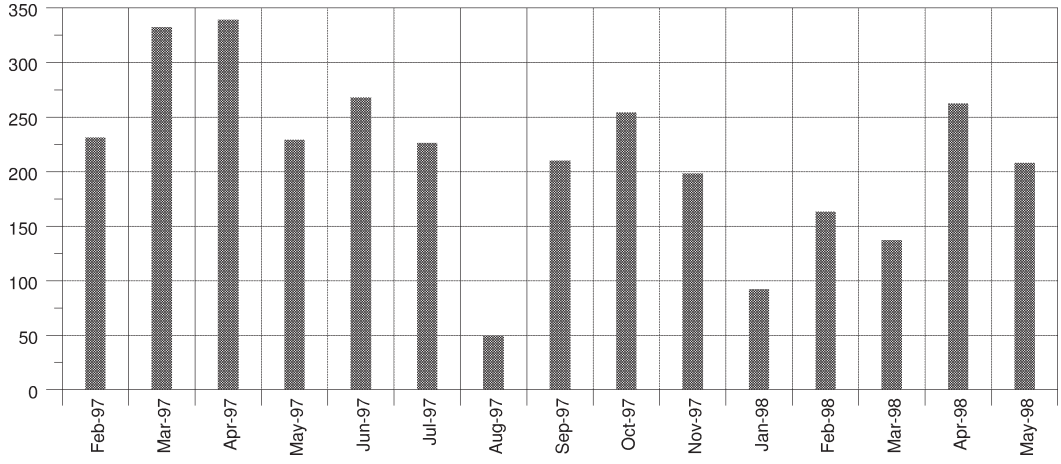


Fig. 16. Monthly usage of the Mocha Geometry Server prototype.

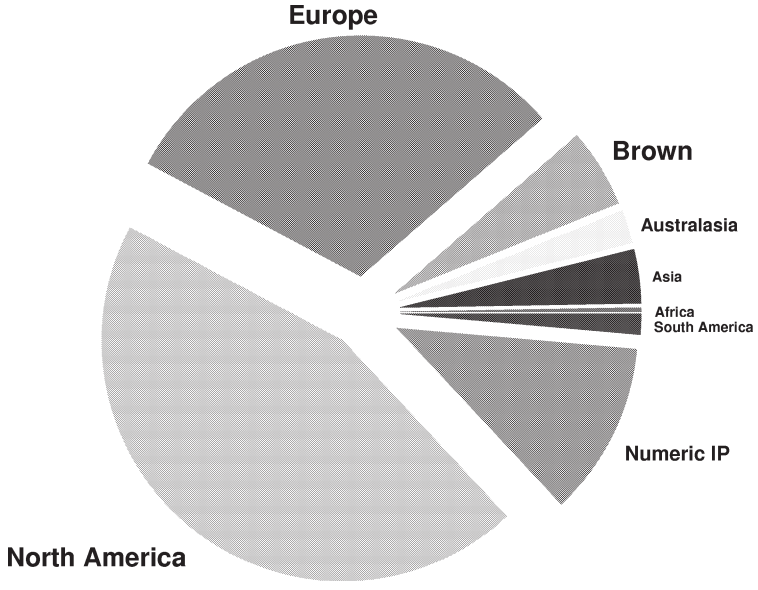


Fig. 17. Geographic distribution of accesses to the Mocha Geometry Server prototype.

4. Technical details

In this section we re-visit the architecture of Mocha already introduced in the previous sections and give a detailed description of selected technical issues that have been addressed in the implementation of our prototype.

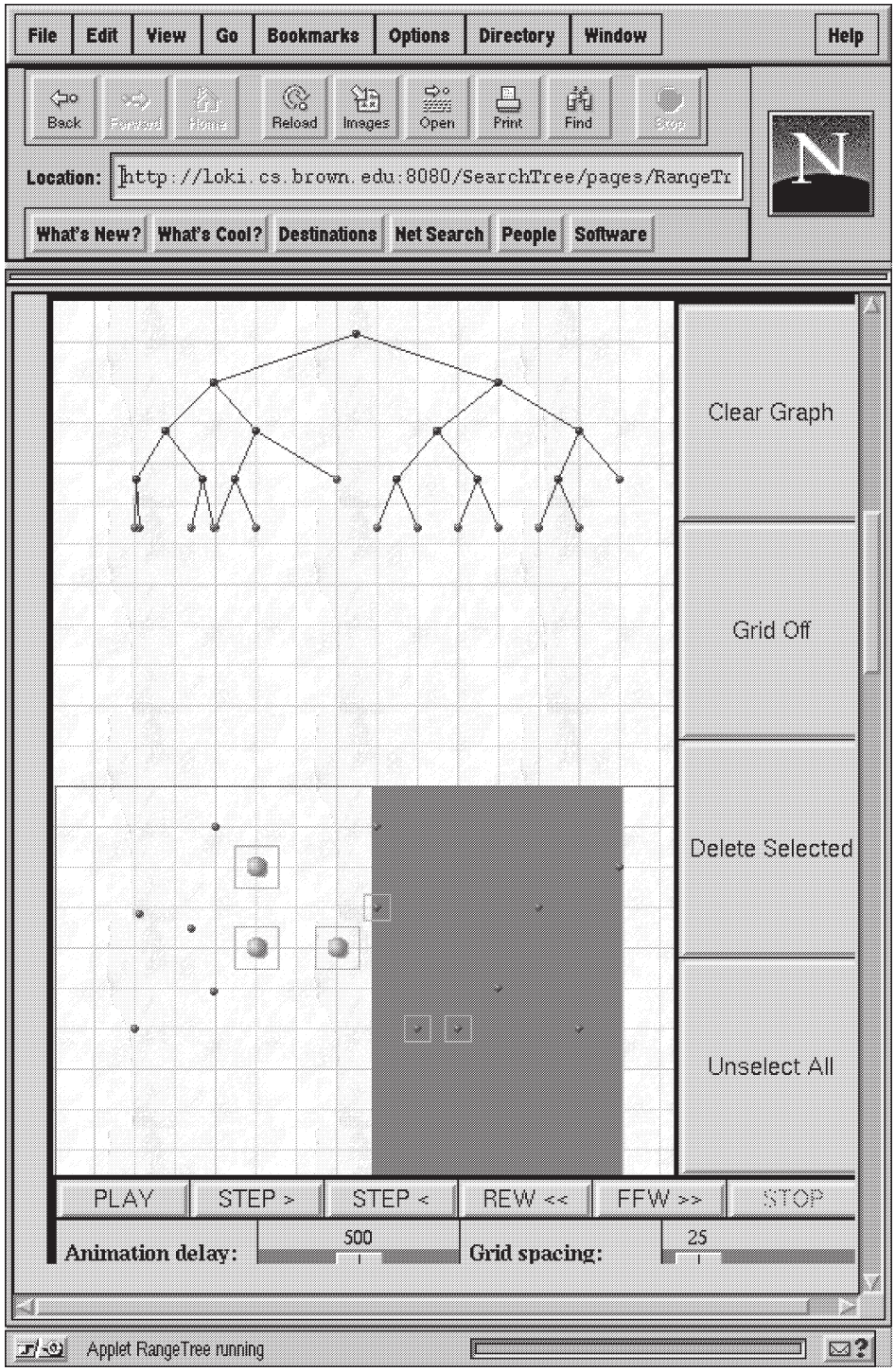


Fig. 18. Mike Walczak's visualization of a 2-D range tree. The user has placed the points and the query rectangle, and is now stepping through the visualization of the query algorithm.

4.1. Design goals

Many of our design goals are derived from the comparison criteria that distinguish Mocha from other models, see Section 2.2.

Security. Java provides support for security on the user side. On the provider side, security is guaranteed both by Java and the design of the algorithm servers.

Authoring. Mocha provides full support of the World Wide Web by being embedded in a Java-compatible browser. Authors and users of algorithm visualizations can place the desired visualization applet as simply another component of an HTML file, comparable to an image, for example. Java also enables the use of CGI scripts from the applet itself, image files (GIF, JPEG), audio streams, etc. Also, the Mocha clients can take full advantage of existing or new services, written in a variety of languages, such as C or C++, as long as such services are designed to use the Mocha visualization protocol or an ad-hoc wrapper is designed to enable its use.

Communication complexity, accessibility, code protection. Using the client-server paradigm is a well-known means of localizing functionality, code, and computation so that these goals can be achieved.

Responsive feedback. Maintaining high responsiveness to the user's interaction is especially important in the case of client-server environments, where there is a possibility of network latency; yet, it is also important from the standpoint of accessibility, since users are allowed to access potentially very expensive computations. Mocha's support of interactive algorithm visualization provides for multiple levels of responsive feedback, ranging from instantaneous to longer range. Display pointer correspondence to the user's mouse, or other input device, should be instantaneous, as well as any drag-and-drop or other direct manipulation of geometric objects. Additional threads, conveniently part of the Java language, provide for other feedback which may not be instantaneous, such as servicing the communication of an expensive geometric computation on the server.

Attractiveness. Although this is subjective, we consider the prototype to be attractive and of interest to a user seeking to better understand the geometric algorithms that have been implemented. The ease of authoring, especially from the standpoint of using resources available on the Internet for creating attractive Web pages, may be the more objective criterion.

Support of multiple views. Mocha employs a model-view-controller paradigm that simplifies the support for multiple views.

Advanced interaction. Geometric structures and their algorithms have many parameters and attributes. Additionally, robust interaction is often required. For example, the support of proximity graphs using the parameter β requires the input of β at exact points to visualize the transitions between special points of interest, as detailed in Section 3.2. A simple scaling would not produce these numbers; instead the client applet provides for "gravity" near these special points of interest. Other possible input mechanisms would be special grids (hexagonal, octagonal) and coordinate systems (polar) to precisely enter input to observe interesting algorithmic behavior.

4.2. Frameworks

Architectural frameworks [21,37] provide for the reusability of the design and implementation of a set of cooperating classes over a given application domain. The advantage that frameworks provide over a monolithic API is that they define the interactions, collaborations, and responsibilities of the components, including the novel parts, in the framework. Frameworks thus provide for "generic software architectures" [37].

Java provides a GUI framework in terms of its `java.awt` package and especially the applet class. Users of this GUI framework are constrained to how the framework dispatches events, such as mouse events or repaints. This simplifies the programming of the component written in Java, as well as its integration on a Web page, potentially with other Java components.

However, the Java framework does not address such issues as the use of the Model-View-Controller (MVC) paradigm [30] or a client-server architecture. In fact, client-server architectures exist outside of Java since they introduce non-Java components such as the interfaces and protocol that connect these components. Mocha can be seen both as an implementation framework and as a design framework for integrating algorithm services.

4.3. The model-view-controller paradigm

The *Model-View-Controller (MVC)* paradigm [30] separates the task of modeling from that of displaying and interacting with the model. For example, an implementation with MVC of an algorithm that visualizes a Voronoi diagram separates the task of managing the geometric structure (i.e., a planar map with points associated with its vertices and regions), from its display which may render the points as shaded spheres. The controller provides facilities for interacting with the display, such as drag and drop, which is then updated in the model.

Note that as the attributes of interest in the visualized geometric objects increase, or as we distinguish the abstract characterization of a geometric object from its implementation as a data structure, it is possible to derive several interesting views. By using MVC we can ensure the correspondence of each view to the model without increasing the complexity of the design (at least beyond the design's initial incorporation of MVC). The importance of this approach for algorithm visualization was introduced by BALSA [9].

Mocha extends the conventional use of MVC by partitioning both the model and the controller between the client and the server. The client will typically employ implementations of the structures optimized for rendering and user control, whereas the server maintains structures for efficient use by the supported algorithms. The visualization protocol supports the maintenance of the correspondence between these models.

MVC on the client supports a high degree of parallelism, which can be exploited through the use of threads on the client. Additional parallelism is through the client-server partitioning. We exploit MVC's parallelism by allocating one or more threads to each task: modeling (interacting with the server), viewing (rendering the display), and interaction (controller).

4.4. Mediators and protocol support

A client-server architecture is the result of a partitioning of the system, that aims at localizing functionalities and/or responsibilities in order to provide better performances, increase security, and

enhance reusability. However, naive application partitioning can result in high maintenance costs and even in a lack of openness if each client-server pair has its own interface. As the number of clients n and the number of servers m expand, the number of relationships can become as large as course, $n \times m$. *Mediators* [53] are a well-known mechanism for reducing such interoperability problem.

Mediators isolate the commonality between the interaction of the client and the server. A mediator can be running on a dedicated machine—for example, to enable fault tolerance or security—but this is a result of the partitioning, not a necessary condition. Indeed, it would often be undesirable to make the mediator the hot spot of communication, but instead to provide it as part of the system design. Mocha provides a mediator as part of the framework for both the GUI clients and algorithms servers. This mediator then supports the visualization protocol.

4.5. Client implementation

Clients are composed by the following components:

Java-enabled WWW browser. Currently Sun HotJava, Netscape Navigator, and Microsoft Internet Explorer provide support for Java; other browsers will likely do so in the future because of the appeal of interactive content.

GUI. The GUI supports the view and controller of the MVC paradigm. It is written in terms of Java and its GUI framework.

Executor. The executor maintains the model in response to both the user and the annotated algorithms through the visualization protocol.

We have implemented our framework for the Java clients on top of the existing applet/panel GUI framework. The internal architecture of the Java framework is based on a container/component pattern that is becoming widely adopted, such as in OLE, OpenDoc, and other systems. Containers distribute events (`repaint`, `mouseDown`, `mouseDrag`) to their components through event handlers that can be further derived through inheritance. In the Mocha framework, we introduce the additional events and handlers corresponding to the specific geometric application domain. For example, we support the entry of point sets through `movePoint` and `addPoint` events. These events are then routed through the geometry manager (a mediator), which supports the visualization protocol between the client and the geometry services.

4.6. Server implementation

The simplest component of our architecture are the servers. Servers are created from the following:

Session manager. This element supports the creation of a context (state) through the use of processes. As new clients attach to the session manager through the sockets protocol, additional processes are forked to handle the desired service.

Protocol manager. Supports the visualization protocol.

Model manager. Model support of geometric objects. Typically this is a large component of the service, as it is with LEDA, which has rich support for robust geometric objects.

Service implementation. The actual annotated algorithms, such as Voronoi or β-proximity.

We currently support two services, based on the libraries that they were built on: proximity and LEDA. Other services will become useful in future versions of this architecture. Simple services are easy to construct with existing libraries or filters through the wrapping with a thin socket dispatcher and model translator. A database of interesting geometric objects that are created and viewed with these tools is an example of a service that could be readily accommodated in the architecture.

5. Extensions

Collaborative environments enable members of communities to interact with each other over the Internet with not only traditional tools (email, news, etc.) but also through specialized tools [3]. In particular, we envision the development of collaborative environments which provide access to geometric animations and visualizations in the context of Web browsers.

For example, members of the computational geometry research community could experience together interactive algorithm visualizations on a shared geometric white board, potentially while using other Internet collaboration tools developed by other parties, commercial and academic.

Introducing collaboration extends the possibilities of use of the Mocha environment, and some preliminary work has been done in the development of a collaborative version of our Mocha project [32].

Issues to be considered in collaborative environments is the consistency of views, the synchronization of these views, and the policy for updating the collaborative space. At Brown, we have developed an initial prototype of collaborative Mocha; see `http://loki.cs.brown.edu:8080/CollaborativeMocha/pages/CMocha.html`. The prototype supports users joining and leaving at different times in a common room. Users can manipulate the white board and observe the manipulations of other users as they occur. The floor policy is simultaneous updates on discrete entities—first come, first served on contention.

We are currently working on the following extensions of Mocha:

- Dynamic partitioning of the algorithm visualization system. The Java class loader loads and garbage collects classes (from the Web server, file system, or other sources) as they are used by instance objects. It is possible to use this mechanism to progressively increase the functionality of an applet as it is running or to move functionality and responsibility from the server to the client applet, assuming a common Java code base.
- Providing a Java beans implementation of Mocha. The Java beans component model was developed to support both visual programming tools as well as the arbitrary composition of Java beans, which extends the existing embedding capabilities of the Java applet model in documents. Implementing Mocha within the Java beans framework would significantly simplify the development of palette-oriented white-boards.

Acknowledgements

We would like to thank Maurizio Pizzonia for help and insight in experimenting with the Mocha Geometry Server prototype.

References

[1] N. Amenta, S. Levy, T. Munzner, M. Philips, Geomview: A system for geometric visualization, in: Proc. 11th Annu. ACM Sympos. Comput. Geom., 1995, pp. C12–C13.

[2] K. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, Reading, MA, 1996.

[3] C.L. Bajaj, Collaborative multimedia and scientific toolkits, Course Notes, Department of Computer Science, Purdue University, West Lafayette, IN, 1996.

[4] G. Barequet, S.S. Bridgeman, C.A. Duncan, M.T. Goodrich, R. Tamassia, Classical computational geometry in GeomNet, in: Proc. 13th Annu. ACM Sympos. Comput. Geom., 1997, pp. 412–414.

[5] J. Bazik, R. Tamassia, S.P. Reiss, A. van Dam, Software visualization in teaching at Brown University, in: J. Stasko, J. Domingue, M.H. Brown, B.A. Price (Eds.), Software Visualization: Programming as a Multimedia Experience, MIT Press, Cambridge, MA, 1998, pp. 383–398.

[6] P. Bose, G. Di Battista, W. Lenhart, G. Liotta, Proximity constraints and representable trees, in: R. Tamassia, I.G. Tollis (Eds.), Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science, Vol. 894, Springer, 1995, pp. 340–351.

[7] P. Bose, W. Lenhart, G. Liotta, Characterizing proximity trees, Algorithmica 16 (1996) 83–110. (Special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia.)

[8] M. Brown, J. Domingue, B. Price, J. Stasko, Software visualization, in: Proc. ACM Conf. on Human Factors in Computing Systems, Vol. 2, 1994, p. 463.

[9] M.H. Brown, Algorithm Animation, MIT Press, Cambridge, MA, 1988.

[10] M.H. Brown, ZEUS: A system for algorithm animation and multi-view editing, in: IEEE Symposium on Visual Languages (VL '91), 1992, pp. 4–9. Also available from `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-075.html`.

[11] M.H. Brown, J. Hershberger, Color and sound in algorithms animation, IEEE Comput. 25 (12) (1992) 52–63.

[12] M.H. Brown, M.A. Najork, Algorithm animation using 3D interactive graphics, in: Proc. ACM Symp. on User Interface Software and Technology, 1993, pp. 93–100.

[13] M.H. Brown, M.A. Najork, Collaborative active textbooks, in: IEEE Symp. on Visual Languages, 1996.

[14] M.H. Brown, R. Sedgewick, Techniques for algorithm animation, IEEE Software 2 (1) (1985) 28–39.

[15] P.J. de Rezende, W.R. Jacometti, Animation of geometric algorithms using GeoLab, in: Proc. 9th Annu. ACM Sympos. Comput. Geom., 1993, pp. 401–402.

[16] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, Algorithms for drawing graphs: an annotated bibliography, Computational Geometry 4 (1994) 235–282.

[17] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, Graph Drawing, Prentice-Hall, Englewood Cliffs, NJ, 1998.

[18] G. Di Battista, W. Lenhart, G. Liotta, Proximity drawability: a survey, in: R. Tamassia, I.G. Tollis (Eds.), Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science, Vol. 894, Springer, 1995, pp. 328–339.

[19] J. Domingue, B.A. Price, M. Eisenstadt, A framework for describing and implementing software visualization systems, in: Proceedings of Graphics Interface '92, May 1992, pp. 53–60.

[20] K.R. Gabriel, R.R. Sokal, A new statistical approach to geographic variation analysis, Systematic Zoology 18 (1969) 259–278.

[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, Reading, MA, 1995.

[22] S.C. Glassman, A turbo environment for producing algorithm animation, in: IEEE Symposium on Visual Languages (VL '93), 1993, pp. 32–36.

[23] J. Gosling, H. McGilton, The Java language environment: a white paper, 1995. `http://www.javasoft.com/whitePaper/java-whitepaper-1.html`.

[24] M. Hall, Core Web Programming: HTML, Java, CGI, & JavaScript, Prentice-Hall, Englewood Cliffs, NJ, 1997.

[25] A. Hausner, D. Dobkin, Making geometry visible: An introduction to the animation of geometric algorithms, in: J.-R. Sack, J. Urrutia (Eds.), Handbook on Computational Geometry, North-Holland, 1997.

[26] S.E. Hudson, J.T. Stasko, Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions, in: Proc. UIST, 1993, pp. 57–67.
[27] J.W. Jaromczyk, G.T. Toussaint, Relative neighborhood graphs and their relatives, Proc. IEEE 80 (9) (1992) 1502–1517.
[28] D.G. Kirkpatrick, J.D. Radke, A framework for computational morphology, in: G.T. Toussaint (Ed.), Computational Geometry, North-Holland, Amsterdam, 1985, pp. 217–248.
[29] A. Knight, J. May, M. McAffer, T. Nguyen, J.-R. Sack, A workbench for computational geometry (wocg), in: Proc. 6th Annu. ACM Sympos. Comput. Geom., 1990, p. 370.
[30] G.E. Krasner, S.T. Pope, A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, J. Object-Oriented Programming 1 (3) (1988) 26–49.
[31] D.T. Lee, Geometric algorithm visualization revisited, in: Workshop on Geometric Computing, 1997.
[32] L.D. Lejter, A collaborative environment for algorithm animation on the WWW, `http://loki.cs.brown.edu:8080/CollaborativeMocha/paper_html/index.html`.
[33] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, Reading, MA, 1997.
[34] K. Mehlhorn, S. Näher, LEDA: a platform for combinatorial and geometric computing, Commun. ACM 38 (1995) 96–102.
[35] J. Muthukumarasamy, J.T. Stasko, Visualizing program executions on large data sets using semantic zooming, Technical Report GIT-GVU-95-02, Georgia Institute of Technology, 1995.
[36] B. Myers, Taxonomies of visual programming and program visualization, J. Visual Lang. Comput. 1 (1) (1990) 97–123.
[37] O. Nierstraz, T.D. Meijler, Research directions in software composition, ACM Comput. Surv. 27 (2) 1995.
[38] S.P. Reiss, A framework for abstract 3D visualizations, in: IEEE Symposium on Visual Languages (VL '93), 1993, pp. 100–107.
[39] S.P. Reiss, An engine for the 3D visualization of program information, J. Visual Lang. Comput. 6 (3) (1995). (Special issue on Graph Visualization, edited by I.F. Cruz and P. Eades.)
[40] S.P. Reiss, I.F. Cruz, Practical software visualization, in: CHI '94 Workshop on Software Visualization, 1994.
[41] G.G. Robertson, S.K. Card, J.D. Mackinlay, Information visualization using 3D interactive visualization, Comm. ACM 36 (4) (1993) 56–71.
[42] R.W. Scheifler, J. Gettys, The X window system, ACM Trans. Graph. 5 (2) (1986) 79–109.
[43] J. Stasko, A. Badre, C. Lewis, Do algorithm animations assist learning? An empirical study and analysis, in: Proc. ACM Conf. on Human Factors in Computing Systems, 1993, pp. 61–66.
[44] J.T. Stasko, The path-transition paradigm: a practical methodology for adding animation to program interfaces, J. Visual Lang. Comput. 1 (3) (1990) 213–236.
[45] J.T. Stasko, Simplifying algorithm animation with tango, in: Proc. IEEE Workshop on Visual Languages, 1990, pp. 1–6.
[46] J.T. Stasko, Tango: a framework and system for algorithm animation, IEEE Computer 23 (9) (1990) 27–39.
[47] C. Stoll, The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage, Doubleday, New York, 1989.
[48] A. Tal, D. Dobkin, Gasp: A system to facilitate animating geometric algorithms, in: Proc. 10th Annu. ACM Sympos. Comput. Geom., 1994, pp. 388–389.
[49] A. Tal, D.P. Dobkin, Visualization of geometric algorithms, IEEE Trans. Visualization Computer Graphics 1 (1995).
[50] R. Tamassia et al., Strategic directions in computational geometry, ACM Comput. Surv. 28 (4) (1996). `http://www.cs.brown.edu/people/rt/sdcr/report.html`.
[51] G.T. Toussaint, The relative neighbourhood graph of a finite planar set, Pattern Recogn. 12 (1980) 261–268.

[52] A. van Dam, The electronic classroom: Workstations for teaching, Internat. J. Man-Machine Studies 21 (4) (1984) 353–363.
[53] G. Wiederhold, Mediation in information systems, ACM Comput. Surv. 27 (2) (1995).