Visualizing Gometric Algorithms Over the Web

J.E. Baker, I.F. Cruz, G. Liotta,
and R. Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-97-10
July 1997

Visualizing Geometric Algorithms over the Web*

James E. Baker[†] *Isabel F. Cruz*[‡] *Giuseppe Liotta*[§]
Roberto Tamassia[†]

July 3, 1997

Abstract

The visual nature of geometry applications makes it 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 WWW browser supporting Java. The Mocha prototype has been widely accessed over the WWW, 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, WWW, client-server architecture, visual interface, Java, multimedia.

---

*Research supported in part by the National Science Foundation under grant CCR–9423847 and CAREER Award IRI–9625105, by the U.S. Army Research Office under grant DAAH04–96–1–0013, by the N.A.T.O.-C.N.R. Advanced Fellowships Programme, and by EC ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244.

[†]Center for Geometric Computing, Department of Computer Science, Brown University, Providence, RI 02912–1910, USA. {jib,rt}@cs.brown.edu

[‡]Computer Science Department, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609–2280, USA, ifc@cs.wpi.edu. Work by this author was performed in part at Brown University.

[§]Dipartimento di Informatica e Sistemistica, Universitá di Roma "La Sapienza", Via Salaria 113, Roma 00198, Italy, liotta@dis.uniroma1.it. Work by this author was performed in part at Brown University.

1 Introduction

Algorithm visualization and animation appeals 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. See, e.g., the survey by Myers [34] and [7, 8, 9, 10, 13, 17, 20, 23, 39, 44, 45]. Algorithm animation is also a powerful tool to demonstrate new algorithms to others in a intuitive, often appealing fashion. Therefore, there is a strong pedagogical interest associated with interactive algorithm visualization, which can be used by students individually or in class demonstrations [4, 42, 50, 12]. Visualization can also be used as a debugging tool for large software applications [38].

The visual nature of geometry applications makes it 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., [28, 47, 32, 14, 26] and the collections of videos accompanying the proceedings of the ACM Symposium on Computational Geometry since 1992) [46, 47]. An extensive survey by Hausner and Dobkin on the visualization of geometric algorithms is also available [22]

There is every reason to believe that work on the visualizations of geometric algorithms will continue and even accelerate in the future [48]. 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* [43], 3D visualization [11, 40, 37], and automatic graph layout [15]. Another challenge, which extends beyond algorithm animation, is the visualization of large sets of data [33]. As pointed out in [22], a major issue in the visualization of geometric algorithms is the realization of interactive animations, where the user has direct control over the algorithm's input (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 to travel forward and backward in time).

A new challenge relates to making animation available to a wide 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 [41] provides a basic client-server mechanism for algorithm animation over the Internet, which we shall call *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 [21] and its environment opens the possibility of embedding interactive applications in HTML documents, which are executed through the WWW 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 WWW. It is object-oriented, and therefore easily extensible, while providing a wide-range of authoring capabilities given its interface with the WWW. 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 over the WWW 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, which we use in our implementation) 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* [3], 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 WWW 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 [8] and the conceptual framework pioneered by Stasko [45, 43, 44], 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 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 between the algorithm, GUI, and animator is illustrated in Figure 1.

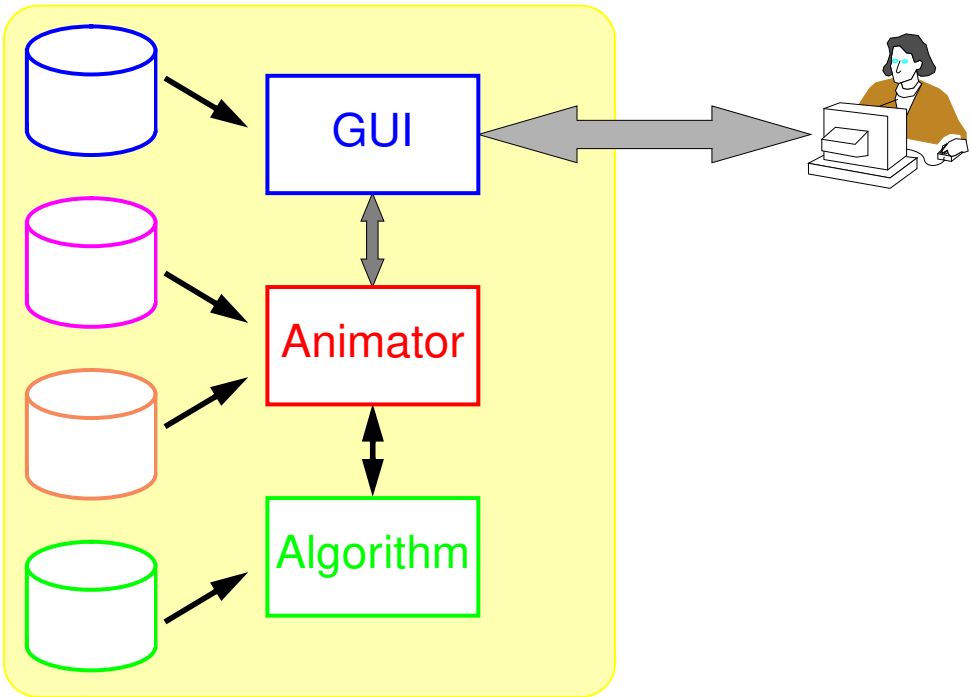


Figure 1: Components of an interactive system for visualizing and animating algorithms.

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 [41] provides a basic client-server mechanism for algorithm visualization over the Internet, which we shall call *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 (display events). For example, this mechanism is used to provide on-line demonstrations of the *XTango* visualization system [45], see Figure 2, where the visualization program can be activated by a `cgi-bin` script launched in a Web browsing session.

The X model of algorithm visualization is schematically illustrated in Figure 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 to display complex visualizations on not very sophisticated computers.

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 access to the visualization program with an `xhost +` command (or `xauth` authentication procedure), and hence allows a "Trojan horse" visualization

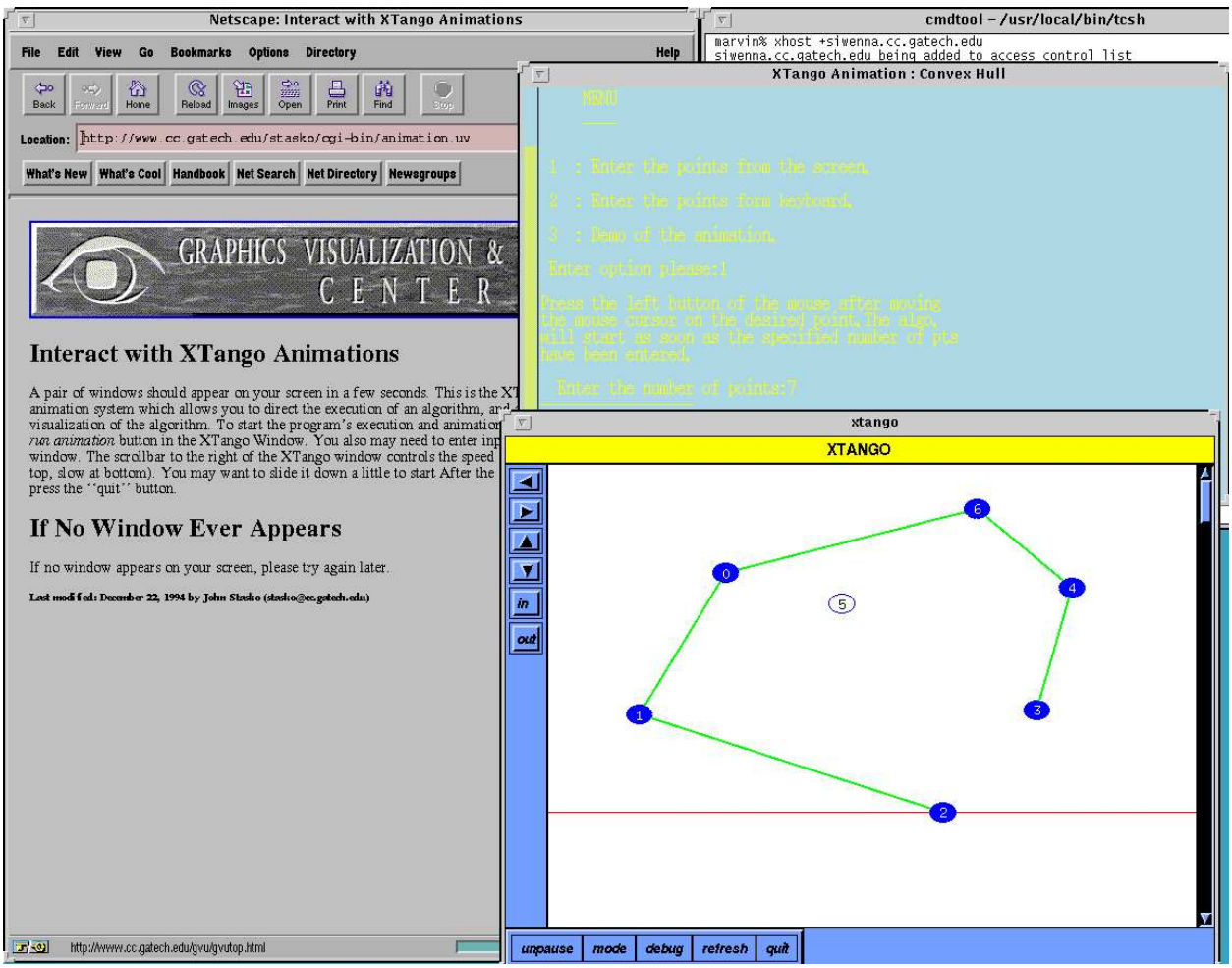


Figure 2: An XTango convex hull visualization in the X-model.

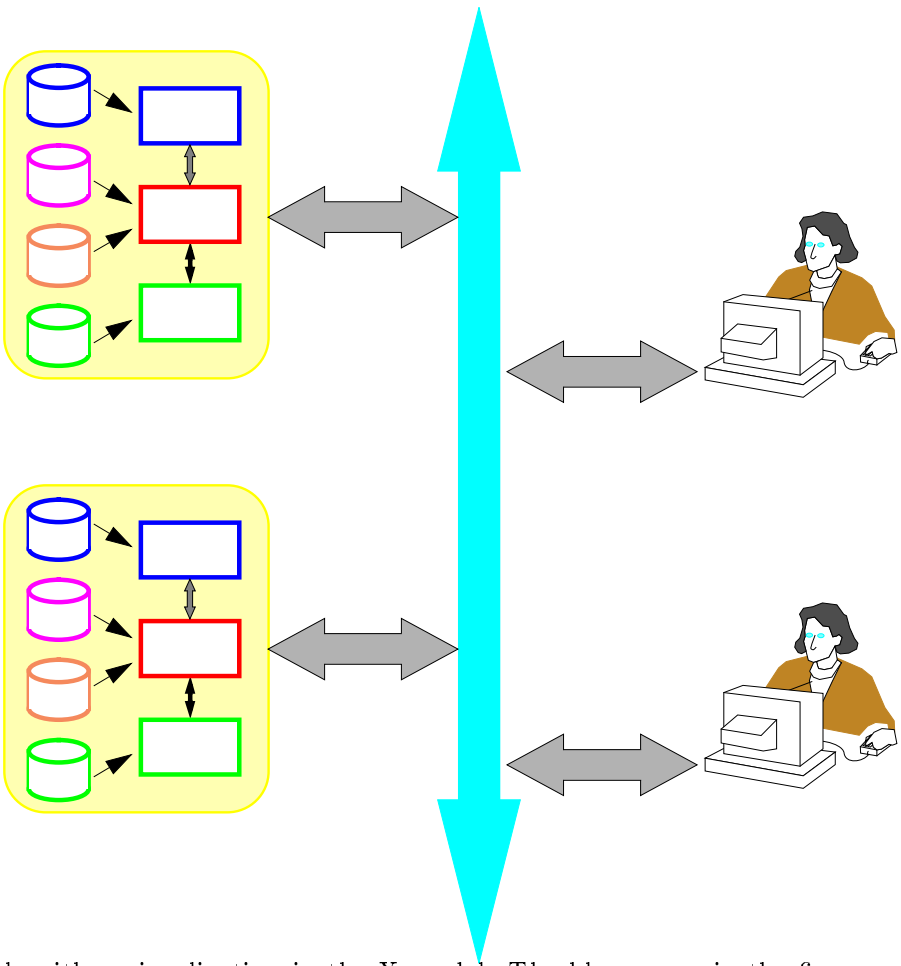


Figure 3: Algorithm visualization in the X model. The blue arrow in the figure represents the Internet; the grey double arrows describe the communications data exchanged between the end-user and the remote X server. The thickness of the grey arrows is comparable with the one of the blue arrow, to describe that the communication of the X-model is high. The icon representing the local computer is small to describe that no powerful computers are needed under this model.

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. Hence, visualizations with complex dynamic scenes need to transport large amounts of data through the Internet, which makes poor use of the Internet bandwidth and slows down the interaction.

The thickness of the grey arrows of Figure 3 is comparable with the one of the blue arrow, to describe that the communication load of the X-model is high. On the other hand, the icon representing the local computer is small to describe that no powerful local computers are needed.

2.4 The Java Model

Java [1] 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 (JVM) 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 [30]. In the Web environment, these classes are generally transported to the Web browser through the HTTP protocol for execution on the user's machine as an applet contained on a Web page, although other mechanisms exist; see for example `http://www.marimba.com`.

Java is an emerging standard for Internet programming. Being an interactive part of the Web enables greater authoring capabilities through hypertext narratives; it also enables the use of the Web for distribution of this interactive content in terms of Java programs to a large audience.

Java incorporates safety and security into its design:

- no pointer arithmetic;
- checked array bounds;
- automatic garbage collection;
- code authentication of loaded modules;
- a byte code running on a virtual machine;
- 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

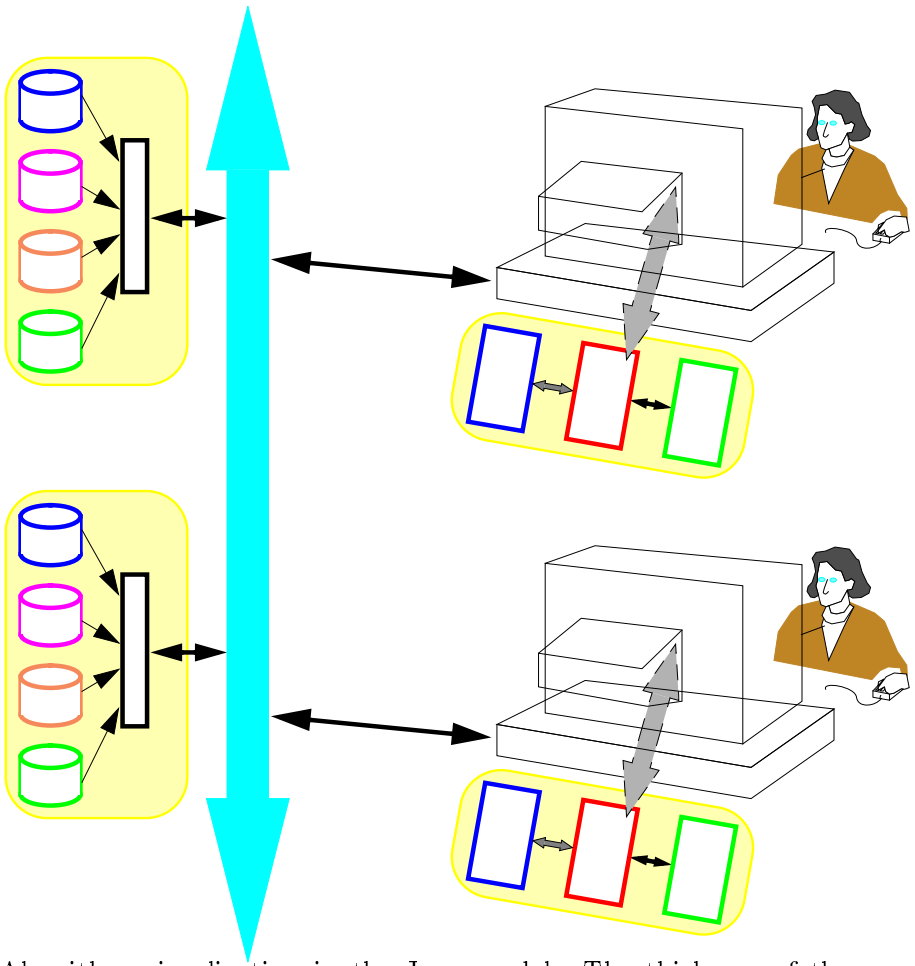


Figure 4: Algorithm visualization in the Java model. The thickness of the grey arrows is very thin , to describe that the communication load of the Java model is low. On the other hand, the icon representing the local computer is large to describe that powerful local computers may be needed. Also, possibly expensive local computations that involve locally running visualization code, GUI code, and annotated algorithm code are highlighted in the figure.

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.

Finally, one interesting capability of Java is its provision for the automatic generation of hypertext documentation (in HTML) from tagged source code. We have provided this documentation in the web version of this document.

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 Figure 4. The three main functional components of the visualization system (GUI, animator, 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 [8]) that uses the Java model is shown in Figure 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, animator, and algorithm. The latter can be particularly disadvantageous since the provider may want to use directly existing algorithm libraries (e.g., LEDA [31]).

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

The thickness of the grey arrows of Figure 4 is very thin, to describe that the communication load of the Java model is low. On the other hand, the icon representing the local computer is larger than the one of Figure 3 to describe that powerful local computers may be needed. Also, possibly expensive local computations that involve locally running visualization code, GUI code, and annotated algorithm code are highlighted in the figure.

2.5 The Mocha Model

The online `webster` dictionary 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

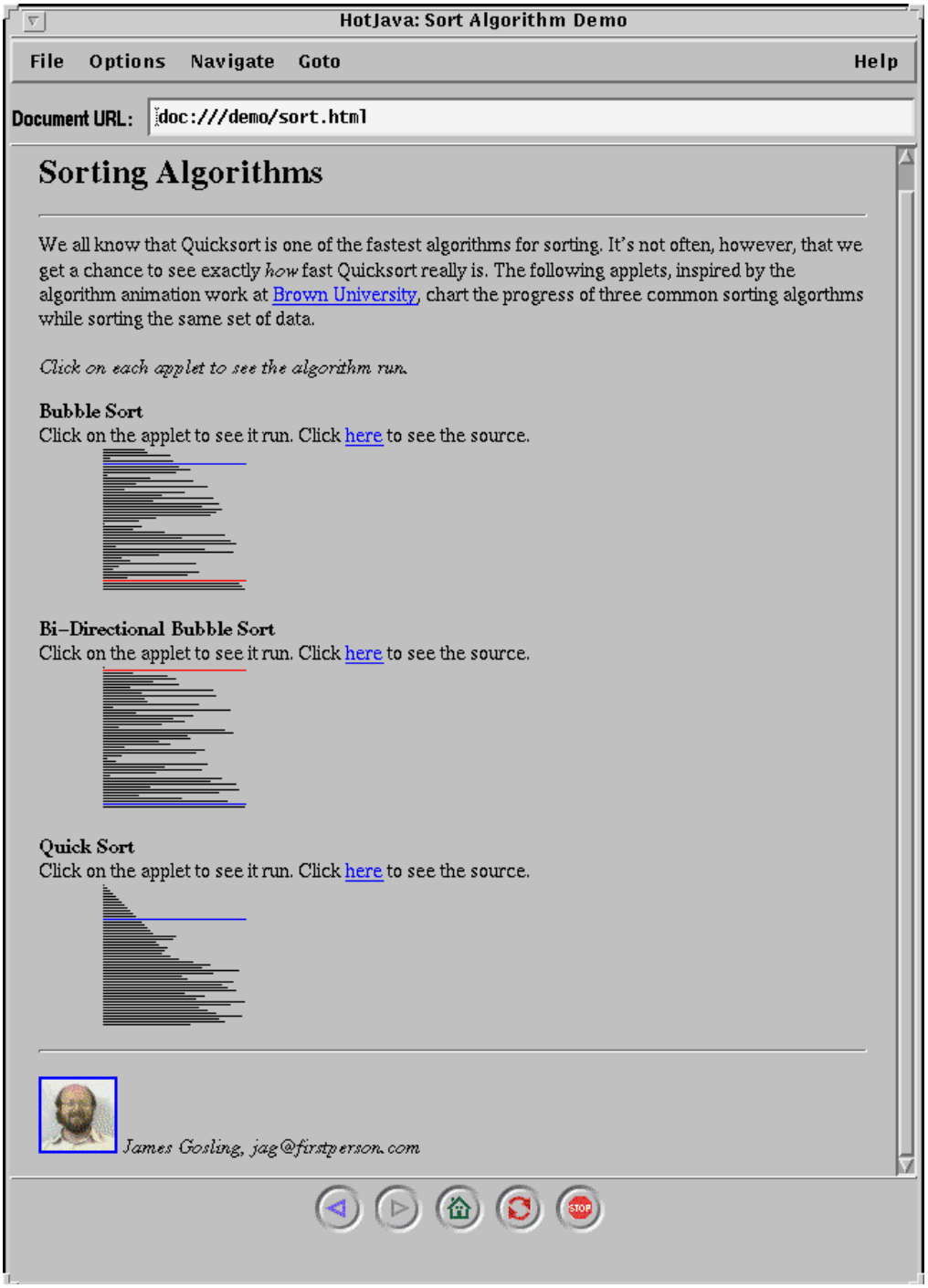


Figure 5: A sorting visualization in the Java model.

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 (to taste) 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). Yet Mocha is not simply a pastiche or the latest novelty blend. In particular, our efforts to define 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, is what distinguishes Mocha from a naive blend.

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, e.g., Figures 8 and 13. Details of the design, architecture and implementation of Mocha are provided in Section 4.

The Mocha model, which is schematically illustrated in Figure 6, is a distributed architecture for algorithm visualization, where the algorithm is executed on the provider's machine(s), while the animator 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 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 code and multimedia objects to the GUI on the users's machine by means of the HTTP protocol. The provider also employs several other servers to execute algorithms and exchange data (inputs, operations, and control) with the animator and GUI on the users'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 animator 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.

Observe that the thickiness of the arrows describing the communication load in Figure 6 is intermediate between the corresponding ones shown in Figure 3 and 4. Also, the size of the icon representing the local computer in Figure 6 is small, since no powerful local machines are needed. The local computation, that involves locally running visualization code and GUI code, is suitably depicted in the figure.

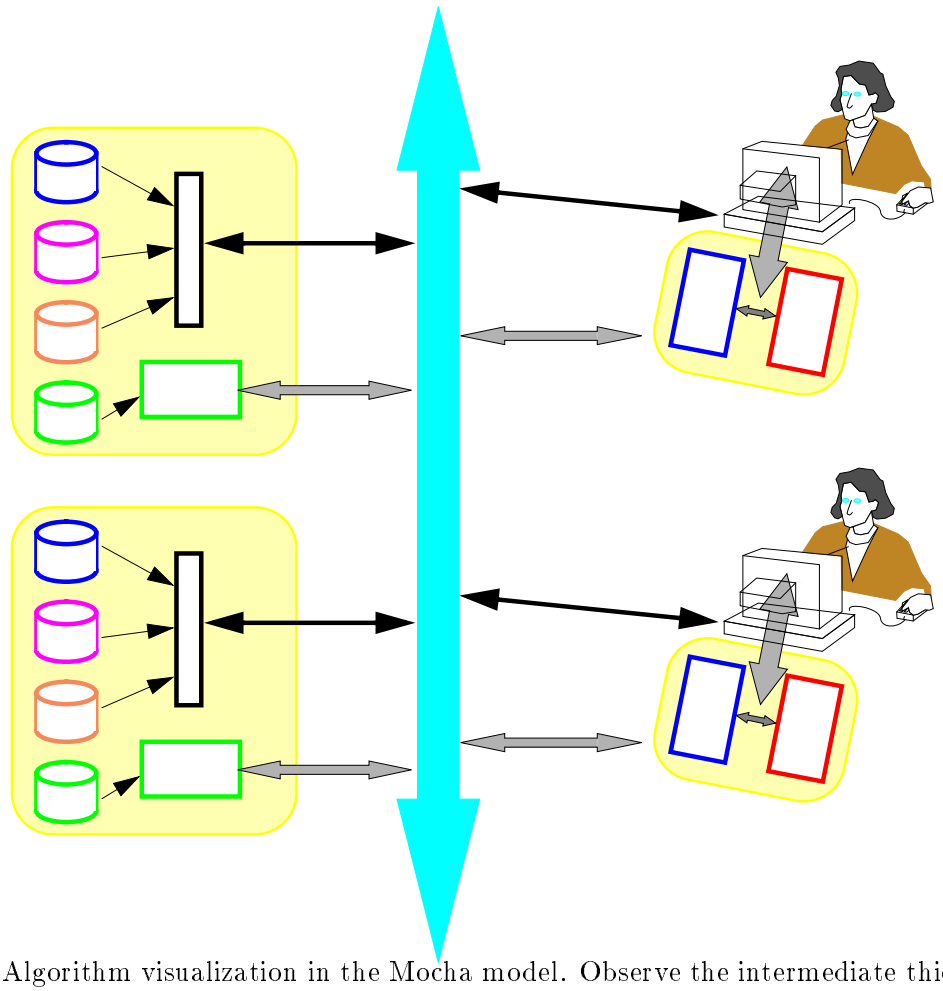


Figure 6: Algorithm visualization in the Mocha model. Observe the intermediate thickiness of the grey arrows representing a moderate communication load. Also, the size of the icon representing the local computer is small, since no powerful local machines are needed. Some local computation is requested, involving locally running visualization code and GUI code.

	Security	Code Protection	Authoring	Communication Complexity	Accessibility
X model	−−	++	+	−−	+
Java model	++	−	+	++	−−
Mocha model	++	+	++	+	+

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

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 qualitatve comparison. An entry with a "++" 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 Section 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 runs 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 animation stream between the Java applet and algorithm animation server, we can determine the additional cost of using X.

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

- XScope, a utility provided in the standard X distribution, monitors the X stream
- Instrumented algorithm server supporting the Delaunay triangulation in the Mocha prototype

The instrumentation data of the experiment consists 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.) Fig. 7 is a

histogram cumulated at a two-second resolution for a typical user interaction involving inserting, deleting, and moving up to 25 points in a point set. 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 with this interaction.

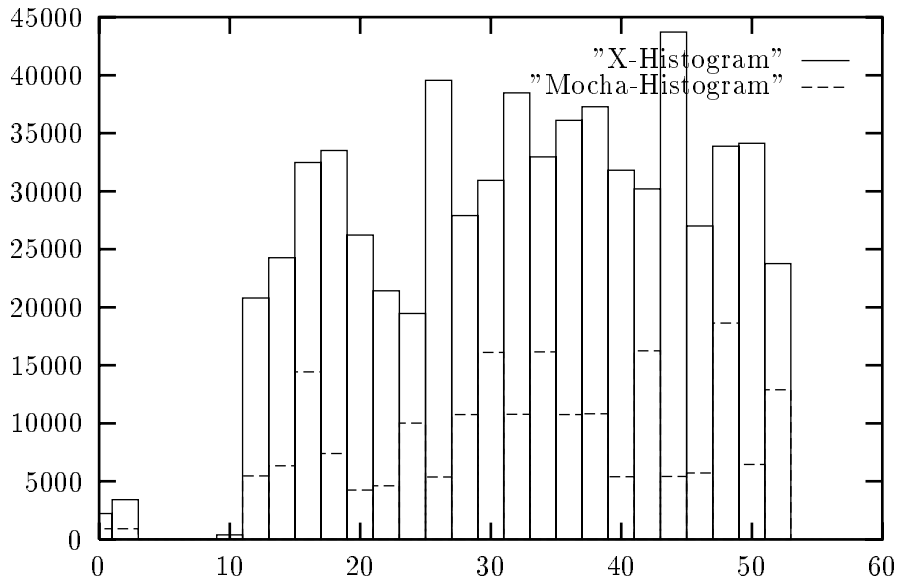


Figure 7: Experimental comparison of the communication load in the X and Mocha models.

3 Interactive Visualization of Geometric Algorithms with Mocha

In this section we describe a prototype that shows the feasibility of the architecture described in Subsection 2.5.

The visualized algorithms solve problems of computational geometry and graph drawing. For an overview of these two fields the reader is referred to [36] and to [15]. In what follows, we will call our system for interactive visualization of geometric algorithms *Geometry Server* or *GS* for brevity.

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 GS are *friendliness* and *interactivity*. Friendliness is achieved by means of an hypertextual interface in which the

visualization results are displayed on canvases embodied in the hypertext. Interactivity is obtained by providing real-time responses to operations such as insertion, deletion, or displacement of the geometric objects of interest in the application. The two subsections that follow aim at describing these aspects within two different application contexts.

3.1 The LEDA Visualization Service

LEDA [31] is a library of data types and computational geometry algorithms written in C and C++. Among its main facilities, it offers effective and robust algorithms for computing the *convex hull*, the *Voronoi diagram*, and, by suitable manipulation, the *Delaunay triangulation* of a finite set of distinct points in the plane. The *LEDA Server* of GS consists of an Algorithms Operations Server and of an Algorithm Library with the algorithms of LEDA that compute such geometric structures.

The user of GS interacts with the hypertextual interface of the LEDA Server. Such interface, written in the HTML markup language, is stored locally on the user's computer, as a result of the interaction (via the http protocol) between the MM Documents/Code Server, the GUI and the Executor. 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, called *Section Convex Hull*, *Section Delaunay Triangulation* and *Section Voronoi Diagram*, respectively. Each section is provided with a canvas where the user is supported by standard user-interface facilities, such as zoom-in and zoom-out, addition, deletion and displacement of objects.

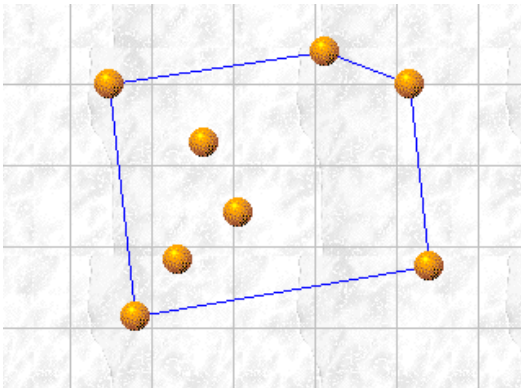
Suppose the user wants to execute the algorithm of LEDA Server that computes the convex hull[36] of a finite set S of distinct points of the plane. The client, using the Visualization Protocol, interacts with the LEDA Server. The Executor receives the algorithm operations code that are needed to construct the convex hull; the GUI replies to the LEDA Server by sending back messages about possible changes of the input, such as deletion, addition, or displacing of elements of S. To any of such messages there corresponds a new sequence of algorithm operations that the executor will perform on S, recomputing the convex hull and displaying it through the GUI.

The visualization is interactive, since the end-user can perceive in real-time how the convex hull changes when a point is inserted, deleted or moved around in the canvas (see Figure 8).

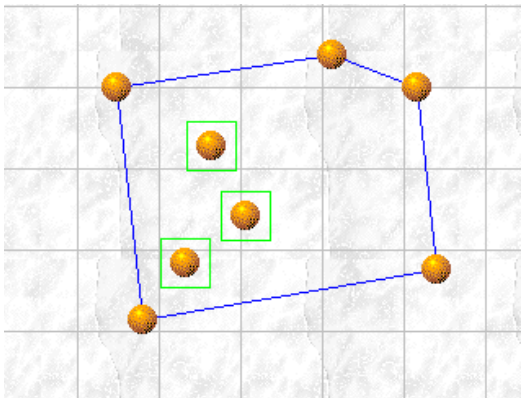
Similarly, the algorithms to compute the Voronoi diagram and the Delaunay triangulation of S are visualized (see Figs. 9–11).

Mocha has also revealed as a powerful tool for experimenting efficiency and robustness of existing implementations of geometric algorithms. For example, it is a useful tool to visualize degenerate sets of points for which Voronoi diagram and Delaunay triangulation algorithms are typically affected by numerical errors that may give rise to topologically inconsistent outputs. An example of a degenerate configuration with three collinear points is shown in Figure 11. The visualization reveals that the Voronoi diagram was incorrectly computed by the algorithm.

(a)



(b)



(c)

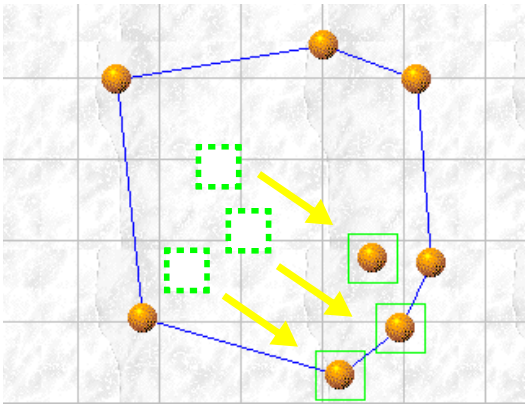
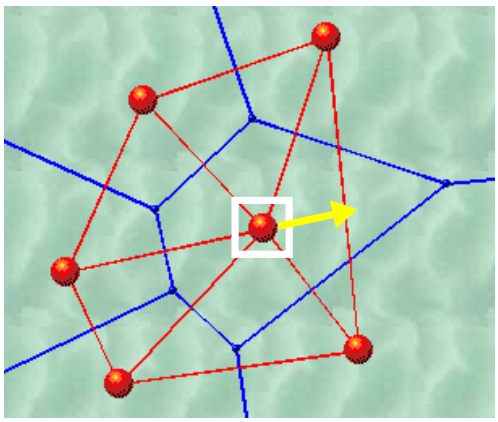
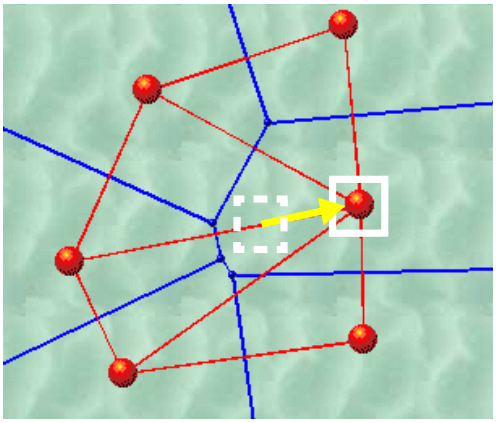


Figure 8: Convex hull visualization in Mocha: (a) initial view; (b) selection of three points; (c) dragging the selected point causes the convex hull to be interactively updated.

(a)



(b)



(c)

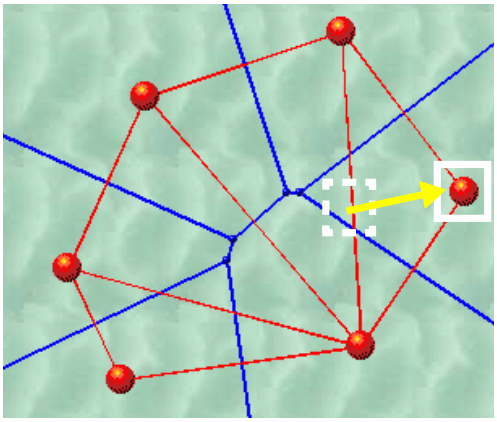
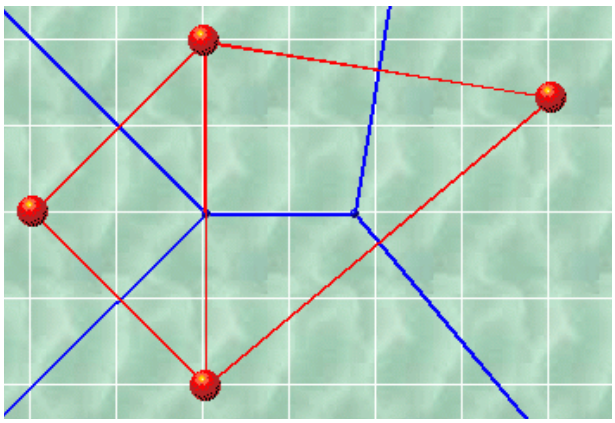
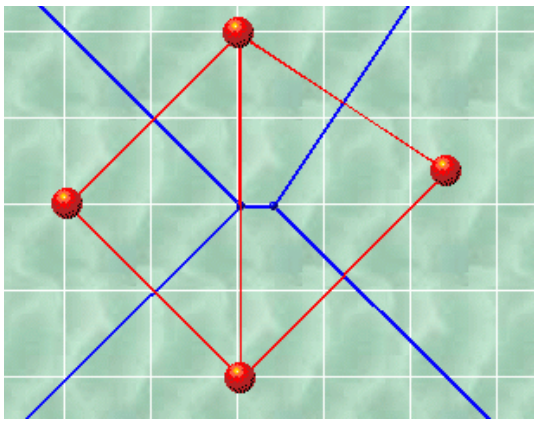


Figure 9: Voronoi diagram and Delaunay triangulation visualization in Mocha: dragging the selected point causes the Voronoi diagram and Delaunay triangulation to be interactively updated.

(a)



(b)



(c)

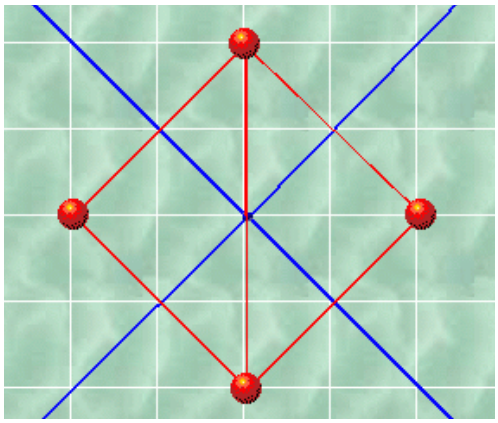


Figure 10: Voronoi diagram and Delaunay triangulation visualization in Mocha: dragging a point to create a degenerate configuration with four cocircular points where one of the vertices of the Voronoi diagram has degree 4.

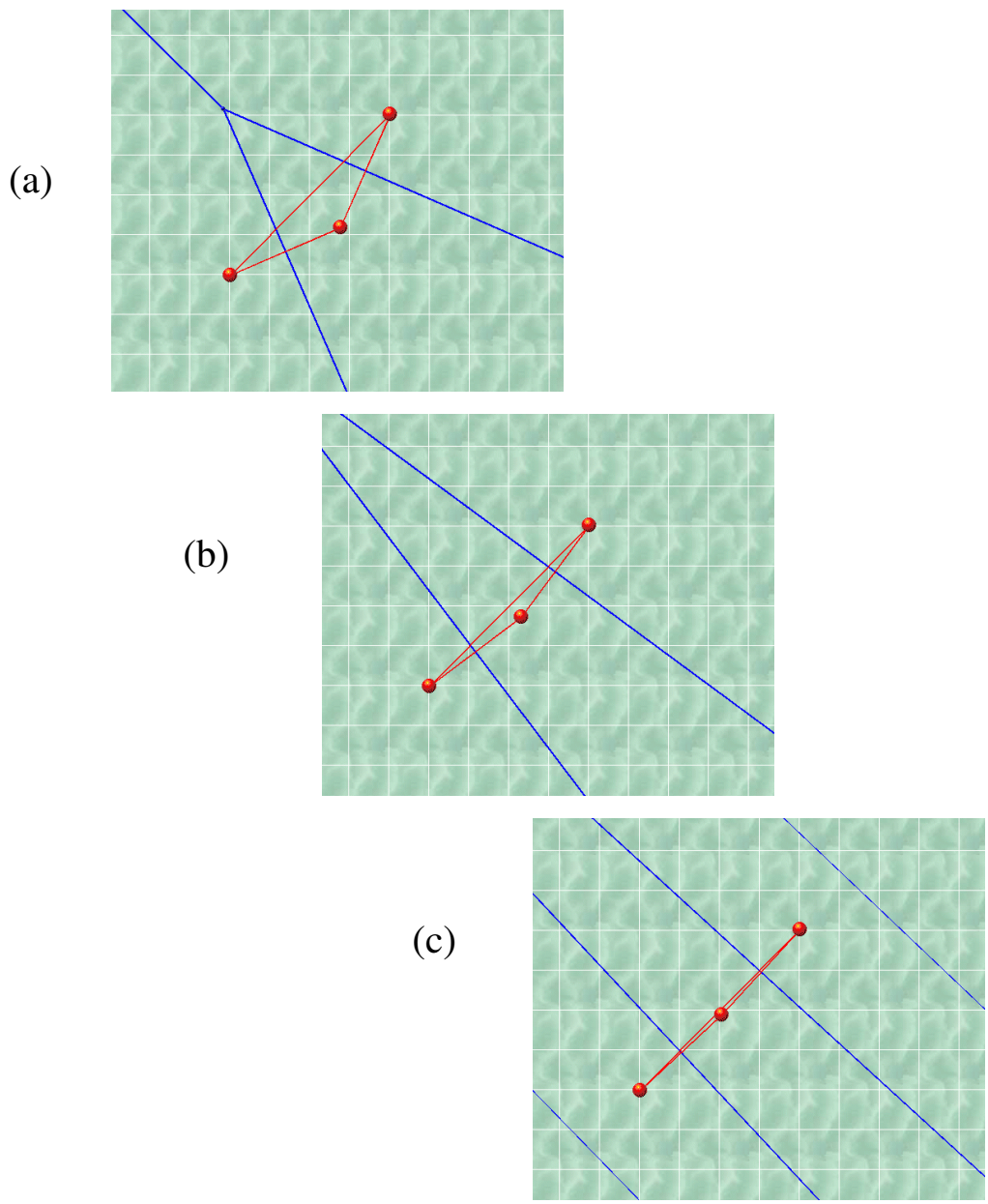


Figure 11: Voronoi diagram and Delaunay triangulation visualization in Mocha: the visualization is useful to detect that an error in the algorithm occurs when dragging a point creates a degenerate configuration with three collinear points.

The prototype described in this section contains just three of the many geometric algorithms that are available in LEDA. The geometry server can be easily extended to embody new geometric algorithms, as illustrated in the following code excerpt.

3.2 The Proximity Visualization Service

A *proximity graph* exhibits a relation between points in a point set by connecting pairs of points with edges that are deemed *close* by some proximity measure. A classical way to measure the closeness of two points p and q in a set S is to use a region of the plane, called the *proximity region* of p and q. The points p and q are close if their proximity region is *empty*, i.e. it does not contain any other point of S. It is the shape of the proximity region that determines the type of graph that results. For example, the *Gabriel region* [18] 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* [49]. See the examples in Figure 12. For a survey on proximity graphs and on their applications to graph drawing see [24] and [16].

The *Proximity Server* of GS can generate infinitely many proximity graphs on a given set S. The user, through the canvas of the hypertextual interface, specifies both the set S and a nonnegative parameter β. This parameter unambiguously defines the shape of the proximity region, called the *β-region*. For a formal definition of β-regions see [25]. We provide here an intuitive description of them. See Figure 12 (d). For $\beta = 1$ the 1-region of p and q is their Gabriel region. 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 region decreases in its size until, for $\beta = 0$, it becomes the straight line with endpoints p and q.

It is easy to see that the β-graph of S will be very dense (i.e. will have very many edges) as the β-region of p and q decreases (if $\beta = 0$ the 0-region will be always empty and all elements of S are adjacent), while it becomes very sparse (i.e. it contains very few edges) as β approaches ∞.

Two proximity graphs produced by GS on the same set of points are shown in Figure 13 and Figure 14. In the first figure the parameter β was chosen to be 1; in the second figure we have $\beta = 2$, that corresponds to the lune underlying the relative neighborhood graph. Observe that the graph of the second figure is a subgraph of the one in the first figure. An advanced interaction technique 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, 5]), are explicitly highlighted below the ruler and cause the cursor to "snap" when dragged near them: $\beta = \frac{\sqrt{3}}{2}$, $\beta = \frac{1}{1-\cos\left(\frac{2\pi}{5}\right)}$, and $\beta = \frac{1}{\cos\left(\frac{2\pi}{5}\right)}$).

```
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 callbacks from the geometry editing
      provided by GeometryClient into this child applet. */
   ...
```

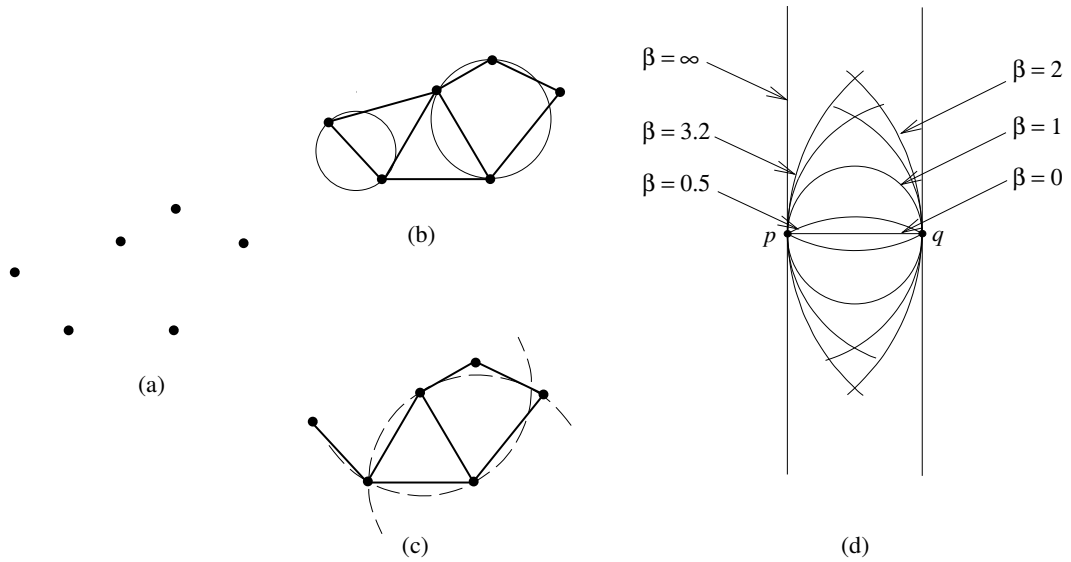


Figure 12: (a) A point set S. (b) The Gabriel graph of S. (c) The relative neighborhood graph of S. (d) The set of β-regions.

3.3 Prototype Usage Statistics

The effectiveness of the Mocha model is confirmed by the access statistics to our running prototype. For the three-month period February 4, 1997 to April 4, 1997, we have analyzed the usage of the Mocha prototype. The pie chart of Fig. 15 illustrates the distribution of users.

Mocha has been successfully used by other developers of algorithm animations. At Brown, with minimal interaction with the developers of the Mocha prototype, 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 architecture; see
`http://www.cs.brown.edu/courses/cs252/range_search/home.html`. At the Max-Planck-Institut für Informatik, the Mocha architecture and the "look and feel" of its graphical user interface have been adopted by the developers of the LEDA library to make available over the Web demonstrations of graph algorithms; see
`http://batman.ag1.mpi-sb.mpg.de:22222/LEDAdemo/`.

4 Technical Details

In this section we re-visit the architecure of the 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.

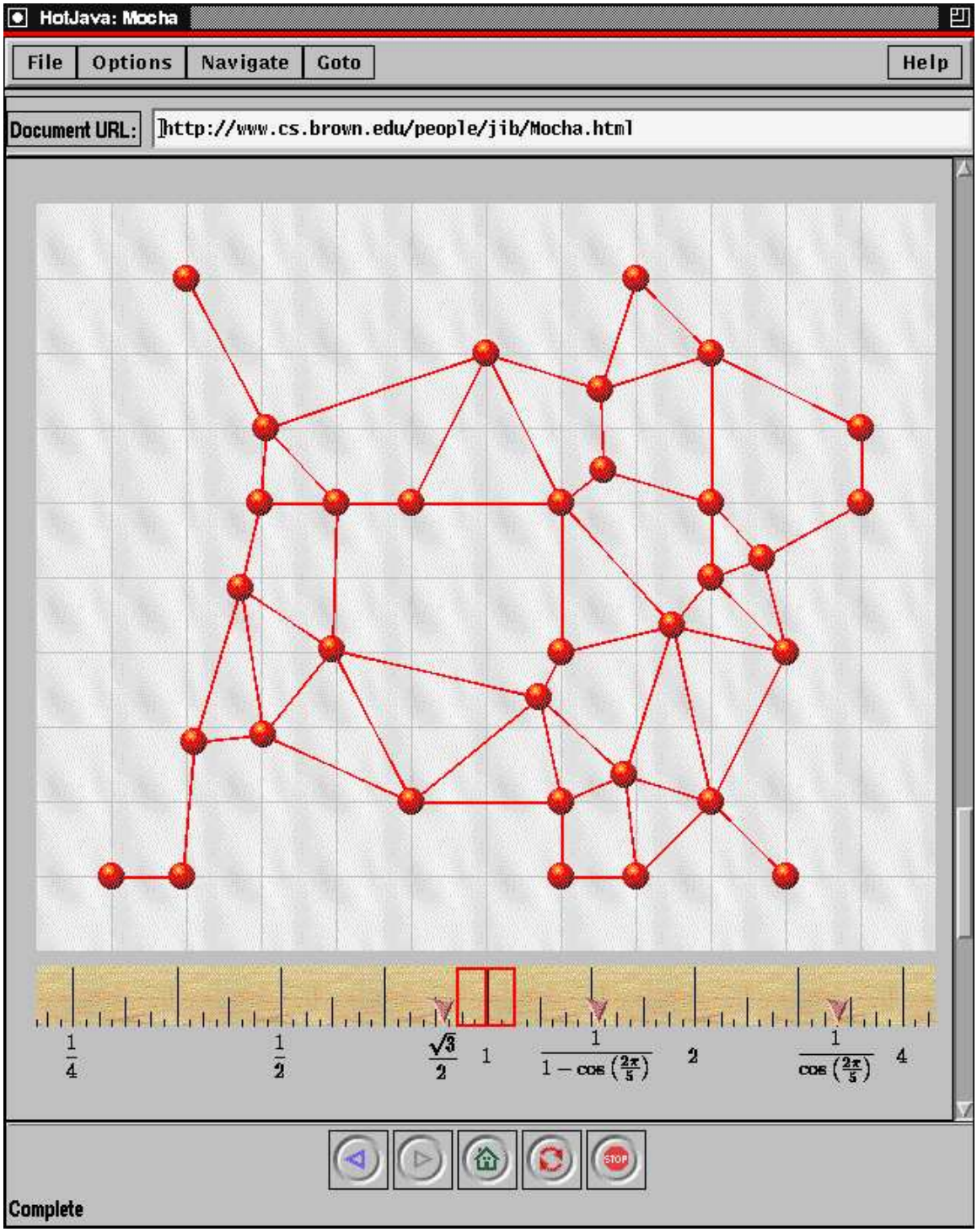


Figure 13: Proximity graph of the a point set with $\beta = 1$.

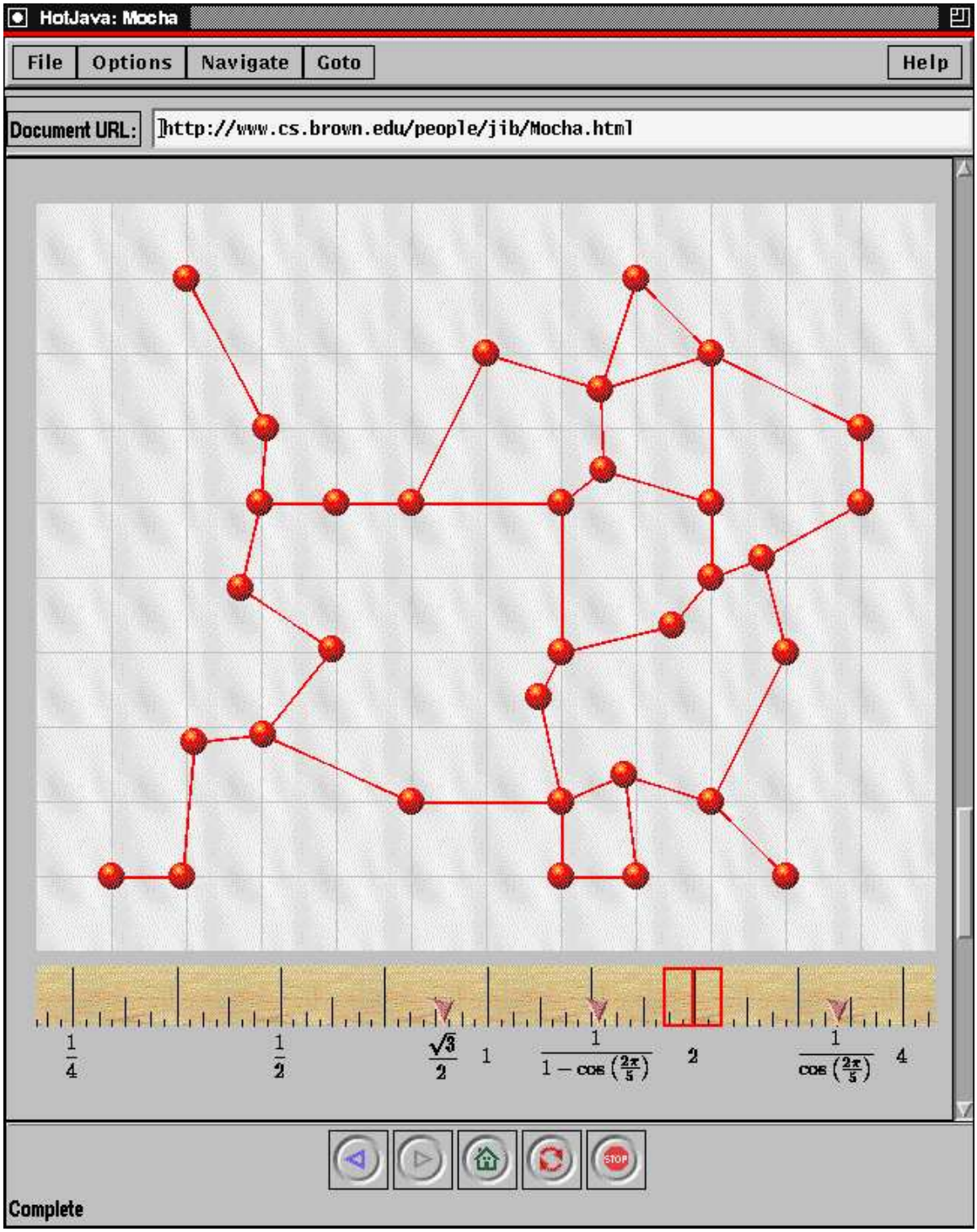


Figure 14: Proximity graph of the a point set with $\beta = 2$.

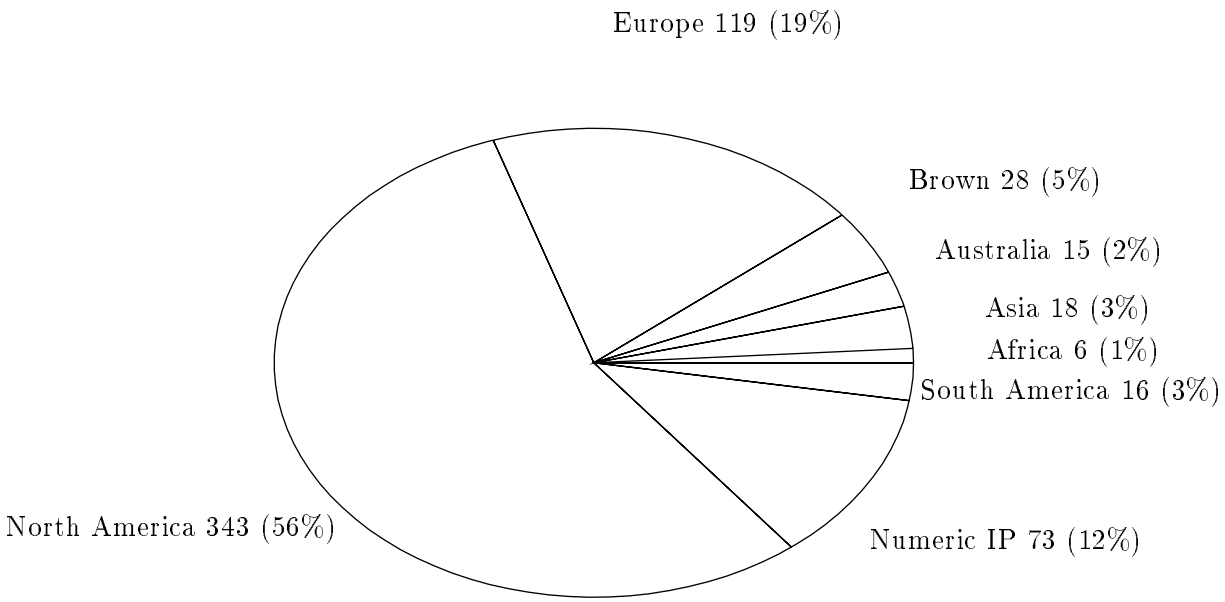


Figure 15: Usage statistics for the Mocha prototype, where we count distinct hits to the main demo page linked to `http://www.cs.brown.edu/cgc/demos/Mocha.html`. Some IP addresses do not resolve to symbolic domain names in our distributed name service database (DNS).

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 simply place the desired visualization applet as simply another component of an HTML file, comparable to an image, for example. The use of Java also enables the simple use of CGI scripts from the applet itself, image files (GIF, JPEG), audio streams, etc. Yet at the same time, the Java clients can take full advantage of existing or new services, written in a variety of languages, such as C++ or LEDA, as long as they can be written to use the visualization protocol or a wrapper is written to enable their 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.

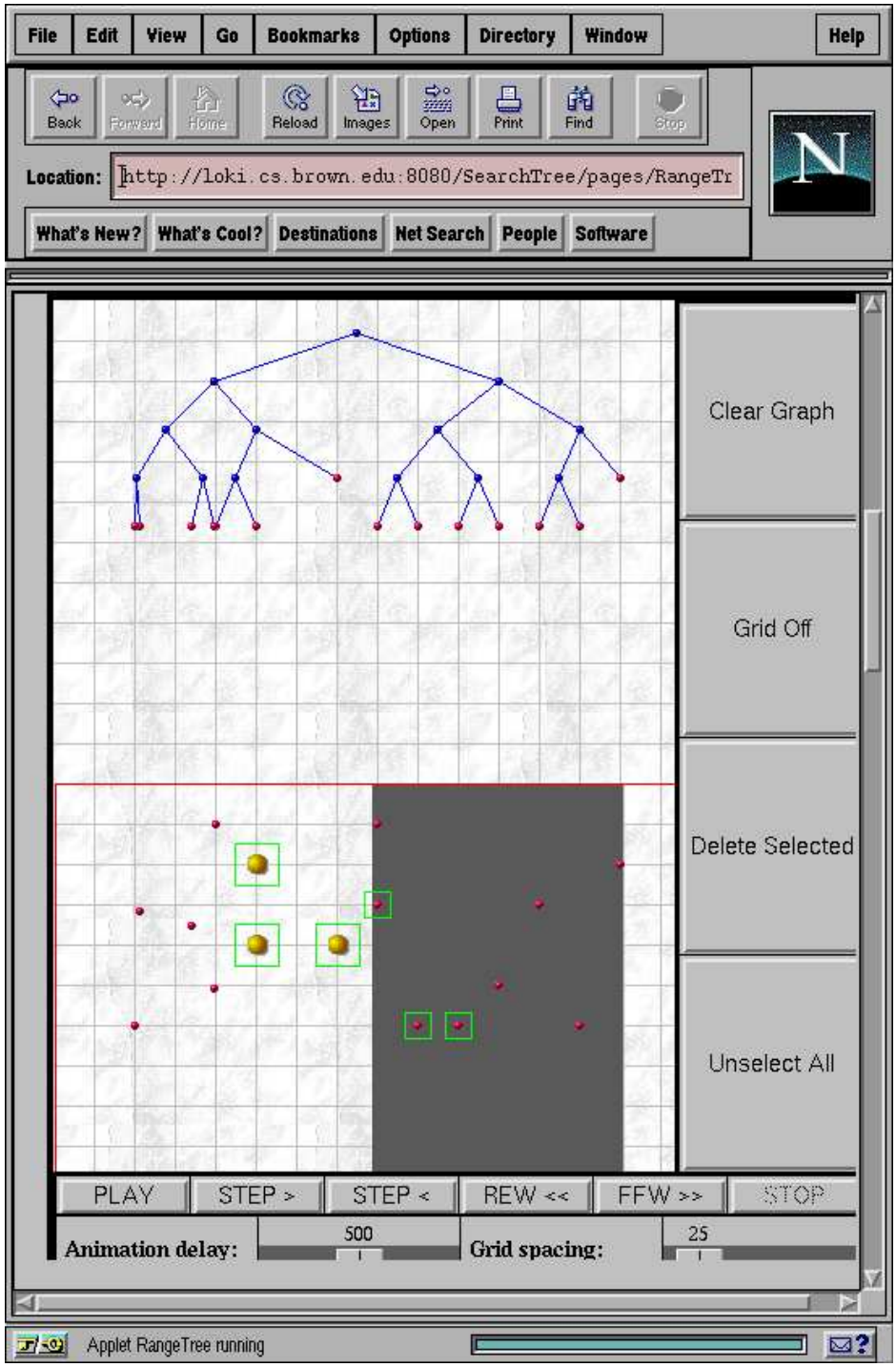


Figure 16: Mike Walczak's algorithm animation of a 2-D range tree with corresponding input point set. The user has placed points and is now stepping through an algorithm animation corresponding to the query rectangle.

Responsive feedback. Maintaining high responsiveness to the user's interaction is especially important in the case of client-server environment where there is a possibility of network latency; yet it is also important from the standpoint of accessibility, where we allow users to have access to potentially very expensive computations.

From the user's standpoint, interaction should provide responsive feedback. Mocha's support of interactive algorithm visualization provides for multiple levels of feedback, ranging from instantaneous to longer range. Display pointer correspondence to the user's mouse, or other input device, should be instantaneous, of course; ideally, any drag-and-drop or other direct manipulation should also be apparently instantaneous. Additional threads, conveniently part of the Java language, provide for other feedback which may not be instantaneous, such as servicing the communication of a lengthy geometric computation on the server. A third layer of feedback might be provided by a monitoring thread that observes the user and suggests further interactional or instructional possibilities. One simple example of such a thread is an audio narrative that instructs the user on the use of the visualization if the user has simply been reading the surrounding text without attempting to interact with the visualization.

Attractiveness. Although this subjective, we consider the prototypes to be attractive and of interest to a user seeking to better understand these algorithms. Here, 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 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 [19, 35] 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" [35].

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 or a client-server architecture. In fact, client-server architectures exist outside of Java since they introduce non-Java components, as well as the interfaces and protocol that connect these components. Mocha is thus both a implementation framework — in terms of support for MVC by visualization clients and common mediator code — as well as a design framework for integrating algorithm services.

4.3 The Model-View-Controller Paradigm

The *Model-View-Controller* paradigm [27] separates the task of modeling from that of displaying the model (view) and of interacting with the model (controller). A conventional implementation of algorithm visualization with MVC would then separate the geometric structures, say of a Voronoi diagram as a planar graph with the nodes marking the Voronoi sites, versus the display which may render the nodes as shaded balls. The controller provides facilities for interacting with the display, such as drag and drop, which then is 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 for algorithm visualization was introduced by BALSA [8].

Mocha extends this conventional use of MVC by partitioning both the model and the controller between the client and the server. Both the client and the server model the geometric structures used in the algorithm visualization, although the client will typically employ implementations of these 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. Some interesting results occur when this correspondence is not fixed instantaneously, as with a transactional protocol, but is instead allowed to lag or to be incremental, to account for the effects of network latency or lengthy computations.

Because the visualization protocol is a messaging protocol, the servers can also provide control. This is not the same as a peer model because clients always make the initial connection to the service, and not vice versa, but once initiated, the server can asynchronously introduce visualization events.

4.4 Mediators and Protocol Support

A client-server architecture is the result of a decomposition, or partitioning, of the system that crosses all of the gross descriptions of the design of the system. Partitioning is not

arbitrary, but rather chosen to localize functionality or responsibility. This may be to provide for better performance, increase security, or enhance reusability, or some other reason. For example, an application for maintaining a warehouse inventory might have a GUI client for interacting with the user and a back-end database. This can increase performance by reducing network traffic, performing display operations only on the client; security, by limiting the database to a more secure machine; and reuse by enabling other component to be replaced, perhaps dynamically, as long as the interfaces remain compatible, such as through a published open application interface (API).

However, naive application partitioning can result in high maintenance costs and even 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 potential relationships is of course $n \times m$. *Mediators* [51] are a well-known mechanism for reducing this interoperability problem. Mediators are used in the commonly-used (and mentioned) three-tiered architecture model in business applications of presentation logic, business rules, and database backend; this could be realized through a windows GUI, a transaction monitor, and a database server for the example of maintaining a warehouse inventory. The transaction monitor would ensure that all participating databases were consistent. The mediator isolates the commonality between the interaction of the client and the server; it may be running on another machine — to enable fault tolerance or security, for example — 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 of the following components to create a coherent framework for easily creating new interactive algorithm visualization:

Java-enabled WWW browser. HotJava from Sun and Netscape both provide support for Java; others 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. This 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 implement our framework for the Java clients on top of the existing applet/panel GUI framework. An alternative choice, to be a content handler for a novel protocol, has limited flexibility at this time because interaction is entirely of the request-reply class.

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 application domain instead of mouse clicks or redraws because the window of the interactive visualization is now exposed. Although this is not always realizable, the Mocha framework is structured so that only events in terms of geometry and visualization are dispatched to derived visualization clients. (Overrides of the framework provide for rare cases where it is necessary for the client to know the current position of the mouse, for example.)

Our support for point sets illustrates the capabilities of our framework. 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.

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). Furthermore, through Java's provision for interthread communication, the interaction thread can simply signal a modeling thread that there is new input, while also requesting a redraw to simulate direct interaction by changing part of the input model. When the computation has finished, perhaps after a lengthy server call, this can trigger again the display thread to make the consistent again. An example of this for Delaunay triangulation is to enable the user to input and edit a point set without latency, while the triangulation is performed in the background and redrawn as available.

4.6 Server Implementation

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

Session manager. his element supports the creation of context or 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 support two services at this time, based on the libraries that they were built on: proximity and LEDA. Other services will become useful in future versions of this architecture. 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 this architecture.

Simple services are easy to construct with existing libraries or filters through the wrapping with a thin socket dispatcher and model translator. We anticipate quickly adding a large library of existing geometric algorithm filters to Mocha .

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 [2]. 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 would be able to 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 [29].

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 serve 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 will significantly simplify the development of palette-oriented white-boards, enabling the user has the choice of working with and even installing new geometric objects and algorithms.

References

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

[2] C. L. Bajaj. Collaborative multimedia & scientific toolkits. Course Notes, Dept. Comp. Sci., Purdue Univ., West Lafayette, Indiana, 1996.

[3] G. Barequet, S. S. Bridgeman, C. A. Duncan, M. T. Goodrich, and R. Tamassia. Classic computational geometry with GeomNet. In *Proc. Annu. ACM Sympos. Comput. Geom.*, 1997.

[4] J. Bazik, R. Tamassia, S. P. Reiss, and A. van Dam. Software visualization in teaching at Brown University. In Brown, J. Domingue, B. Price, and J. Stasko, editors, *Software Visualization: Programming as a Multi-Media Experience*. MIT Press, 1996.

[5] P. Bose, G. Di Battista, W. Lenhart, and G. Liotta. Proximity constraints and representable trees. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 340–351. Springer-Verlag, 1995.

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

[7] M. Brown, J. Domingue, B. Price, and J. Stasko. Software visualization. In *Proc. ACM Conf. on Human Factors in Computing Systems*, volume 2, page 463, 1994.

[8] M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, 1988.

[9] M. H. Brown. ZEUS: A System for Algorithm Animation and Multi-View Editing. In *IEEE Symposium on Visual Languages (VL '91)*, pages 4–9, 1992. Also available from `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-075.html`.

[10] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *IEEE Computer*, 25(12):52–63, 1992.

[11] M. H. Brown and M. A. Najork. Algorithm animation using 3D interactive graphics. In *Proc. ACM Symp. on User Interface Software and Technology*, pages 93–100, 1993.

[12] M. H. Brown and M. A. Najork. Collaborative active textbooks. In *IEEE Symp. on Visual Languages*, 1996.

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

[14] P. J. de Rezende and W. R. Jacometti. Animation of geometric algorithms using GeoLab. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 401–402, 1993.

[15] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.

[16] G. Di Battista, W. Lenhart, and G. Liotta. Proximity drawability: a survey. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 328–339. Springer-Verlag, 1995.

[17] J. Domingue, B. A. Price, and M. Eisenstadt. A framework for describing and implementing software visualization systems. In *Proceedings of Graphics Interface '92*, pages 53–60, May 1992.

[18] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[20] S. C. Glassman. A Turbo Environment for Producing Algorithm Animation. In *IEEE Symposium on Visual Languages (VL '93)*, pages 32–36, 1993.

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

[22] A. Hausner and D. Dobkin. Making geometry visible: An introduction to the animation of geometric algorithms. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*, pages ??–?? North-Holland, 1997. To appear.

[23] S. E. Hudson and J. T. Stasko. Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions. In *Proc. UIST*, pages 57–67, 1993.

[24] J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proc. IEEE*, 80(9):1502–1517, Sept. 1992.

[25] D. G. Kirkpatrick and J. D. Radke. A framework for computational morphology. In G. T. Toussaint, editor, *Computational Geometry*, pages 217–248. North-Holland, Amsterdam, Netherlands, 1985.

[26] A. Knight, J. May, M. McAffer, T. Nguyen, and J.-R. Sack. A computational geometry workbench. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, page 370, 1990.

[27] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug. 1988.

[28] D. T. Lee. Geometric algorithm visualization revisited. In *Workshop on Geometric Computing*, 1997.

[29] L. D. Lejter. A collaborative environment for algorithm animation on the WWW. `http://loki.cs.brown.edu:8080/CollaborativeMocha/paper_html/index.html`.

[30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, 1997.

[31] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[32] T. Munzner, S. Levy, and M. Philips. Geomview: A system for geometric visualization. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C12–C13, 1995.

[33] J. Muthukumarasamy and J. T. Stasko. Visualizing Program Executions on Large Data Sets Using Semantic Zooming. Technical Report GIT-GVU-95-02, Georgia Institute of Technology, 1995.

[34] B. Myers. Taxonomies of visual programming and program visualization. *J. of Visual Languages and Computating*, 1(1):97–123, March 1990.

[35] O. Nierstraz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2), 1995.

[36] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[37] S. P. Reiss. A Framework for Abstract 3D Visualizations. In *IEEE Symposium on Visual Languages (VL '93)*, pages 100–107, 1993.

[38] 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).

[39] S. P. Reiss and I. F. Cruz. Practical Software Visualization. In *CHI '94 Workshop on Software Visualization*, 1994.

[40] G. G. Robertson, S. K. Card, and J. D. Mackinlay. Information visualization using 3D interactive visualization. *Comm. ACM*, 36(4):56–71, April 1993.

[41] R. W. Scheifler and J. Gettys. The X window system. *ACM Trans. Graph.*, 5(2):79–109, 1986.

[42] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *Proc. ACM Conf. on Human Factors in Computing Systems*, pages 61–66, 1993.

[43] J. T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *J. Visual Lang. Comput.*, 1(3):213–236, 1990.

[44] J. T. Stasko. Simplifying algorithm animation with tango. In *Proc. IEEE Workshop on Visual Languages*, pages 1–6, 1990.

[45] J. T. Stasko. Tango: a framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

[46] A. Tal and D. Dobkin. Gasp: A system to facilitate animating geometric algorithms. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 388–389, 1994.

[47] A. Tal and D. P. Dobkin. Visualization of geometric algorithms. *IEEE Trans. Visualization and Computer Graphics*, 1, 1995.

[48] 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`.

[49] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recogn.*, 12:261–268, 1980.

[50] A. van Dam. The electronic classroom: Workstations for teaching. *Int. J. of Man-Machine Studies*, 21(4):353–363, 1984.

[51] G. Wiederhold. Mediation in information systems. *ACM Computing Surveys*, 27(2), 1995.