The Mocha Algorithm Animation System*

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

Abstract We describe the implementation of a new system, called *Mocha*, for providing algorithm animation over the World Wide Web. Mocha is a distributed system with a client-server architecture that optimally partitions the software components of a typical algorithm animation system, and leverages the power of the Java language, an emerging standard for distributing interactive platform-independent applications across the Web.

Mocha We have implemented a prototype of an animation system called Mocha that can be accessed by any user with a WWW browser supporting Java (currently Netscape 2.0 and HotJava) at URL `http://www.cs.brown.edu/people/jib/Mocha.html`.

In this paper, we discuss in detail the implementation of Mocha. A companion paper [1] describes the model underlying the architecture and design of Mocha, provides a comparison between this model and previous ones, and presents an application to the animation of geometric algorithms.

Design Goals Our design goals are derived from the comparison criteria that distinguish Mocha from other models, see [1].

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 animations can simply place the desired animation 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 animation 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.

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 algorithm animation 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 animation if the user has simply been reading the surrounding text without attempting to interact with the animation.

Attractiveness. Although this is 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 mul-

*Research supported in part by the National Science Foundation under grant CCR–9423847, by the U.S. Army Research Office under grants DAAH04–93–0134 and DAAH04–96–1–0013, and by the N.A.T.O.-C.N.R. Advanced Fellowships Programme.

†Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912–1910, USA. {jib,gl,rt}@cs.brown.edu

‡Department of Electrical Engineering and Computer Science Tufts University, 161 College Avenue, Medford, MA 02155, USA. isabel@cs.tufts.edu

AVI '96, Gubbio Italy
© 1996 ACM 0-89791-834-7/96/05..$3.50

tiple views.

Frameworks *Architectural frameworks* [3, 5] 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" [5].

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 animation clients and common mediator code — as well as a design framework for integrating algorithm services.

The Model-View-Controller Paradigm The *Model-View-Controller* paradigm [4] separates the task of modeling from that of displaying the model (view) and of interacting with the model (controller). A conventional implementation of algorithm animation 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 animated 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 animation was introduced by BALSA [2].

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 animation, 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 animation 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 animation 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 animation events.

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* [6] 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

animation protocol.

Client Implementation Clients are composed of the following components to create a coherent framework for easily creating new interactive algorithm animations:

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.

Animator. The animator maintains the model in response to both the user and the annotated algorithms (through the animation 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 animation is now exposed. Although this is not always realizable, the Mocha framework is structured so that only events in terms of geometry and animation are dispatched to derived animation 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 animation 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.

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 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 animation 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 .

References

[1] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. Algorithm animation over the World Wide Web. In *Proc. Int. Workshop on Advanced Visual Interfaces.* ACM Press, 1996.

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

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

[4] 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.

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

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