Vega—A user-centered approach to the distributed visualization of geometric algorithms

Christoph A. Hipke* Sven Schuierer*

Abstract

We present a new approach to the distributed visualization of geometric algorithms that emphasizes the position of the end user. Concepts are introduced that enable a more flexible usage of visualized geometric algorithms, while keeping the task of adapting existing algorithms to the new scheme as simple as possible.

A main proposition is that interactivity should not be built into the visualized *algorithms*, but into the visualizing *system*. With this in mind, we devise a visualization model for geometric algorithms that incorporates strong algorithm execution control, flexible manipulation of geometric input/output data and adjustable view attributes. The new visualization model is implemented in the *Vega*[1] system. Vega offers distributed visualization of geometric algorithms based on source code annotation and supports the standard libraries LEDA and CGAL.

1 Introduction

A picture is worth a thousand words—this well-known proverb illustrates why visualization is so important in creating, presenting and understanding algorithms. Human perception is faster in processing direct visual input than in reading ordinary text. Since algorithms are abstract objects, visualization plays a key role in improving their perception and unterstanding.

- *Researchers* use visualization during the design and debugging phase of an algorithm and also when presenting the algorithm to colleagues.
- *Teachers* can present algorithms and explain their execution steps in detail through the aid of visualization.
- *Students* are able to experiment with existing algorithms and view their results on selected input data.

We use the term *algorithm visualization* for a graphical display of an algorithm execution that is based on the basic structure of the algorithm, not its specific implemention. In showing *abstract* views of an algorithm, algorithm visualization conveys a high *intention content* [11] and thus differs from other visualization models such as software visualization, visual debugging, program-state visualization and data structure visualization. The drawback is that algorithm visualization cannot be easily automated, finding the crucial data structures in an algorithm and visualizing them in an appropriate way has to be done by a human familiar with the algorithm.

An important property of an algorithm visualization system is its *scope*. There are special-purpose systems adjusted to the needs of a specific project, domain-focused systems and general systems for visualizing arbitrary algorithms. Examples for general visualization systems are Balsa [2], Zeus [3] and XTango [10]. A trade-off exists between generality and ease-of-use; adaption of new algorithms to a domain-focused visualization system is usually easier. A domain particularly well suited for visualization is the domain of geometric algorithms. There is a strong need for the visual presentation of geometric objects instead of printing pure coordinates. Furthermore, there is usually a canonical visualization due to the inherent geometric nature of the objects. In this article we only address visualization systems focused on geometric algorithms. Existing systems include the following.

- **XYZ GeoBench** [9] provides an integrated environment for visualizing, testing, and evaluating geometric algorithms. It consists of a user interface including flexible manipulation tools for geometric objects and a library for adapting algorithms. XYZ GeoBench is restricted to the Macintosh platform.
- **Workbench** [4] is an approach resembling the GeoBench and also runs on Macintosh.
- **GeoMAMOS** [6] is a distributed visualization environment for geometric algorithms. It provides an object-oriented framework for adapting existing algorithms to visualization and defines a new protocol *GeoIPC* to con-

*Institut für Informatik, Universität Freiburg, Am Flughafen 17, 79110 Freiburg, Germany. E-Mail: {hipke,schuiere}@informatik.uni-freiburg.de. This research is supported by the DFG-Project "Diskrete Probleme", No. Ot 64/8-1.

[1] More information about Vega is available at http://www.informatik.uni-freiburg.de/~hipke/Vega/.

nect to visualizer clients based on *xfig*.

- **Mocha** [1] is a general model for distributed algorithm animation that has been used to visualize geometric algorithms.

The early systems [4, 9] include an integrated environment (workbench) that offers the user the possibility to use geometric data flexibly and switch between the included algorithms. However, the systems are confined to a specific platform and the addition of new algorithms is difficult. Modern distributed visualization systems overcome this disadvantage but often restrict the user in its interactions with the system. Often input data have to be entered while the algorithm is running, output data cannot be easily processed by other programs and visualization parameters cannot be changed.

In this paper we present concepts that allow the creation of a distributed framework for geometric algorithm visualization which increases usability and at the same time makes visualization of new algorithms simpler. Specifically, we suggest building interaction into the system and not into a single algorithm, thus letting the user take control. This leads to a simple algorithm model and is the basis of other concepts that increase the usability of implemented algorithms while not putting additional workload on the programmer. These concepts include hierarchically structured geometric scenes, views that are adjustable by attribute lists, increased execution control that addresses the granularity of the visualization, and a simple asynchronous distribution model. We present the prototype system Vega that implements our concepts. Vega offers a powerful visualization client based on Java, an algorithm server and a C++ library for the easy adaption of algorithms to the Vega system.

The rest of this paper is organized as follows. In Section 2 we describe our new concepts in detail. The system Vega which implements these concepts is presented in Section 3.

2 Concepts

We start the discussion of concepts with a note on interactivity and a description of our algorithm model.

2.1 Interactivity built into the system

Interactivity enables the user to change the input data of an algorithm and observe the effects on algorithm execution and output. Since this distinguishes algorithm visualization systems from simple movie demonstrations of an algorithm, interactivity is one of the most important aspects of algorithm visualization.

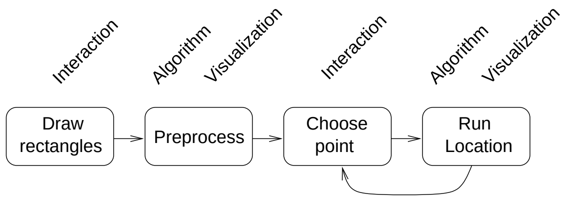

Figure 1: Sample algorithm visualization.

However, if interactivity is used improperly, it can actually limit the usability of a system. Consider an algorithm that performs point location in a set of rectangles. A visualized version of the algorithm could allow the user to first draw some rectangles and then specify points. After each entry of a point the location is performed and visualized, see Figure 1. This is of course a very basic kind of interaction, but typical for a class of algorithm visualizations where the user is prompted whenever the need for input data arises. The algorithm visualization is in control and the interaction is hard-coded into the system. If for instance the user wants to change a rectangle instead of specifying a new location for p in our example, he can not do so and is forced to restart the algorithm and enter all rectangles again. It is also not possible for the user to load the set of rectangles from a file or to write a test program that starts the algorithm with randomly generated input data.

To incorporate interactivity into an algorithm in this fashion creates a rigid algorithm visualization that imposes a fixed sequence of events onto the user. The problem is that algorithm, interaction, and visualization are interlocked. We state the following propositions, see Figure 2.

- *The algorithm and the visualization system should be strictly separated.*
- *Interactivity should be built into the visualization system, not into a single algorithm.*

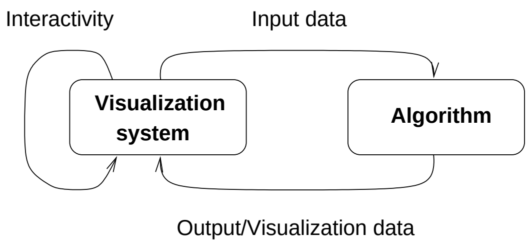

Figure 2: Separating algorithm and visualization.

This new visualization framework overcomes the shortcomings mentioned above. Consider the example algorithm again. In the new model the visualization system lets the user specify a set of rectangles and a point—in any order and either directly or by loading it—and start the algorithm. After the algorithm execution and output is visualized the user may change the input data in any way and start the algorithm again. By bypassing the visualization system completely an automated test program can be written for the algorithm. Furthermore, it is now easy to construct *algorithm pipelines*, the output of one algorithm is used as input of the next.

2.1.1 Algorithm model

Since the interactivity is built into the system, our algorithm model is the classical one—the algorithm is supplied with all its input data at the start, then does some calculation and finally responds with all its output. The algorithm must not directly interact with the user, e.g. prompt for input.

What has to be added for the visualization is an output stream of geometric objects that illustrate the states of the calculation. This stream is subdivided into *algorithm steps*. For each step the geometric objects constitute a picture of the calculation state at that moment. A textual description of the step may be included as well. The steps are structured hierarchically in a tree. The main steps of the algorithm are located below the root of the tree, and every step may have substeps that further show the details of the algorithm execution, see Figure 3. The output also becomes part of this tree as the last main step of the calculation.

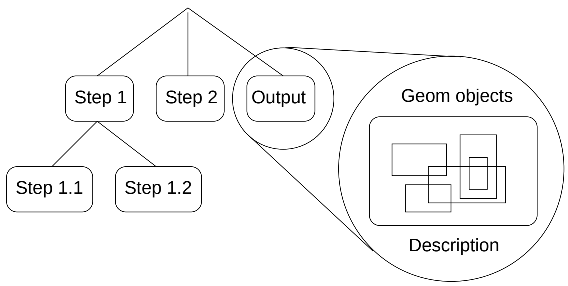

Figure 3: The algorithm execution tree.

In this way, for every input data set the algorithm responds with a structured sequence of pictures of the algorithm execution. That is, it produces a structured description of a movie of its execution. An important advantage of this simple algorithm model is that it makes the task of the programmer that adapts an algorithm to a visualization system even easier. He only has to subdivide the algorithm into steps and send the appropriate geometric objects in each step.

2.1.2 Interactivity options

Since there is no interactivity at the algorithm level, it is important that the visualization system supports as large a set of interactions as possible while retaining the simple algorithm model. Specifically, the following interactivity options should be incorporated.

- Flexible handling of input/output data
- Adjustable view attributes
- Strong algorithm execution control

We will describe these concepts in detail in the following sections.

Another interaction scheme often proposed is changing data while the algorithm is running. Though this seems intriguing, it almost always requires special adaption of the algorithm since internal data have to be adjusted. Though this may be carried out for single algorithms, it is not well suited to a general algorithm visualization framework, especially since it puts interaction back to the algorithm level.

2.2 Hierarchically structured geometric scenes

The key to flexible handling of geometric data and user-adjustable view attributes is a hierarchical naming scheme for geometric objects. Geometric objects can be organized in *groups*, and these groups can be combined further still, such that a *scene* of geometric objects resembles a tree where inner nodes are groups and leafs are geometric objects, see Figure 4.

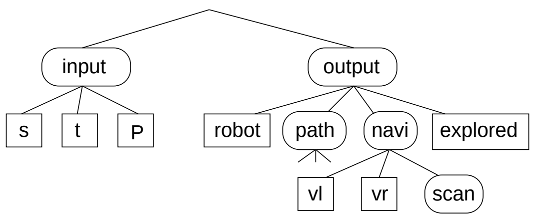

Figure 4: The hierarchical structure of the scene shown in Figure 5.

In addition to its arithmetic parameters, e.g. the coordinates of a point, each geometric object is assigned a *name*. Names of ordinary objects can be empty, these *anonymous* objects can only be referenced accessing their groups. Groups must have a name. The hierarchical structure of the scene is then reflected by the path or *full name* of each geometric object. This contains the names of the groups on its path in the tree in the form `group.subgroup.objectname`. In Figure 5 some objects are labeled with their full names. It shows a scene during the visualization of a search

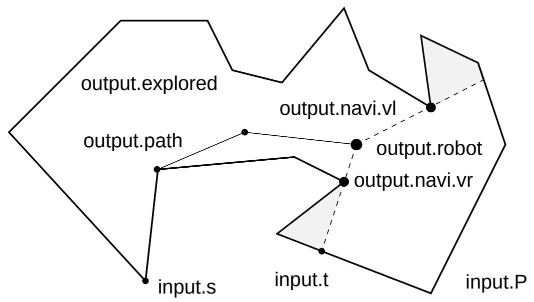

Figure 5: A sample geometric scene.

algorithm for a robot in a polygon. The input consists of the polygon P, a starting point s, and a target point t. The output contains the robot's position, its path, the explored region, and navigation data.

2.2.1 Flexible handling of input/output data

What are the benefits of the hierarchical naming scheme? The first advantage is that all input data can be specified before the algorithm is run. If the algorithm works on a number of objects such as a point list, they can be included as anonymous objects in a common group. For objects of the same type that have a different meaning, e.g. a starting and a target point, the objects are labeled accordingly and can thus be distinguished by the algorithm.

A second advantage is that all geometric data can be manipulated based on their meaning rather than just based on their coordinates. The user can choose to load or save a group of objects, can delete objects or copy them from a different scene. Thus, any part of an algorithm output scene may be selected and isolated. By renaming groups and objects other algorithms may be applied.

In a distributed environment the object naming scheme can also be used to solve the *visualization object consistency problem* [6]. If an object is updated by the algorithm and sent to the visualizer the name clearly identifies the old version of the object that has to be deleted.

2.2.2 Adjustable view attributes

To give the user flexible control over the presentation of a scene we introduce the concept of separating object *properties* and object *attributes.* The properties of a geometric object are its coordinates and its name. The attributes consist of any additional information that is used when actually displaying the object. This includes color, line width, line style, point size, fill color, labeling, and so on. Additional attributes may determine whether the object is visible at all and whether it is currently editable.

By using the hierarchical naming scheme attributes may be assigned to the geometric objects of a scene. Attributes specified for a group by default apply to all of its components, but can be overridden by giving different attribute values for a specific object. It is also possible to apply attributes to a *class* of objects, for example all points inside a group.

This concept has the advantage that attributes may be manipulated and also stored separately from the corresponding geometric objects. Furthermore, it clearly distinguishes *scenes* from *views.* A scene contains only the properties of geometric objects. A view displays the objects of a scene by applying an attribute list specific to this view. Multiple views can be used to highlight different aspects of the same scene by applying different attribute lists. A sample attribute list for a scene may look like this.

```
bgColor=white
lineWidth=2
input.color=red
input.Point.pointSize=7
output.additional.visible=false
```

`bgColor` is a special attribute that sets the background color for the whole view. `lineWidth` is an attribute of the nameless root group and applies to all objects as a default value. The color of objects in the input group is set to red, for all points in this group the size is set to 7, this is an example of a class attribute. All geometric objects in the additional output are not shown in this view.

Default attributes may be defined by the algorithm, which gives the algorithm the same possibilites as in traditional visualization systems where the display is controlled by the algorithm. The advantage of using view attributes is that they are adjustable interactively by the user, even while the algorithm is running. Furthermore, new attributes, e.g. transparency, may be added to the visualizer without affecting existing algorithms.

2.3 Strong algorithm execution control

The algorithm model depicted in Section 2.1.1 enables the visualizer to pass strong algorithm execution control to the user. To begin with, single step and continuous execution mode are supported. At any time the algorithm may be stopped, keeping all current objects in the scene for further use. Secondly, since each scene corresponding to a step of the algorithm is saved by the visualizer, the algorithm can be rewound or executed backwards.

Even finer control is gained by exploiting the hierarchical structure of the algorithm steps. The *granularity* of the execution may be specified by giving a maximum depth for the tree of executed steps. This maximum depth can be specified just for visualization—to get an overall picture of an algorithm—or also for the transmission of the visualization data—to reduce network traffic. The tree of executed steps may also be displayed in an explorer-like window, see Figure 6. This *execution browser* gives an immediate graphical representation of the algorithm's execution structure and is an excellent tool for navigating quickly to special points of interest inside the visualization.

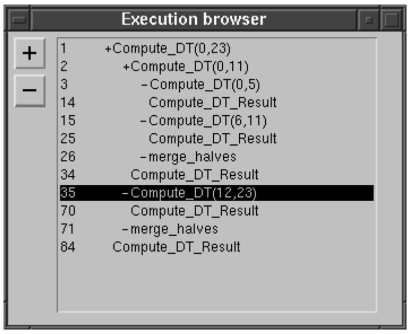

Figure 6: The Vega execution browser.

2.4 Asynchronous distribution model

Distributed visualization systems enable immediate access to algorithms stored at local and remote sites, even if the visualization client and the algorithm server do not share a common platform or operating system. In decoupling algorithm and visualizer, distribution also makes it possible to maintain and update the algorithm data base and the visualizing system independently. Furthermore, the open interface between client and server allows enhanced usage of the supplied algorithms for automated testing oder statistical analysis.

One of the benefits of our algorithm model is that it permits a narrow bandwidth asynchronous distribution model. The communication protocol is simple: The visualization client sends an algorithm name and some input data to the server, the server starts the algorithm and sends back the output data in one continuous stream. No synchronisation or interaction is needed. This is particularly advantageous when the transfer rate is high but latency is high as well.

Since geometric objects are transmitted via class identifier, coordinates and name, the required network bandwidth is small. In the Vega system we use a text based protocol which generates an average of 40 bytes for simple objects such as points or segments. Moderately complex scenes have a size of about 4K, for a run of 40 steps this leads to a typical size of 160K. This still can be significantly reduced by using a binary protocol and entropy coding.

3 The Vega system

We have implemented the concepts discussed in the previous section in a system called *Vega*—Visualization environment for geometric algorithms. Additional goals in the development of Vega have been:

- Platform-independence
- Modularity
- Intuitive interface for the viewing user
- Support for the standard libraries LEDA [7] and CGAL [8]

Currently, only 2D visualization is implemented, but the model can be applied to 3D visualization as well.

3.1 Architecture

The Vega system is divided into three main components, the server, the algorithm library and actual implemented algorithms, and the client, see Figure 7.

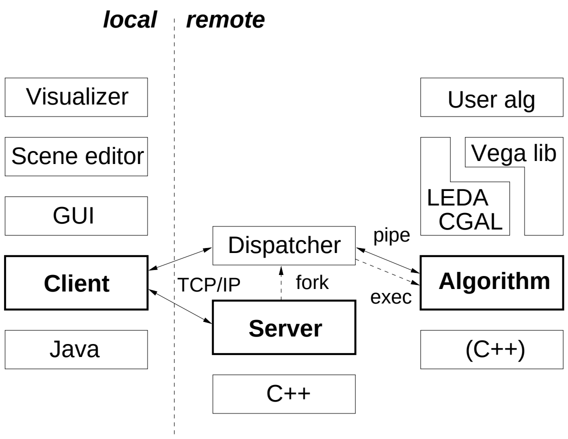

Figure 7: The Vega architecture.

We chose C++ as the language for the algorithm library, since most existing algorithm implementations are in C/C++. Adapting algorithms without the library is not difficult though, so other programming languages can be used as well. The server is written in C++ also for performance reasons. Library and server have been tested on Solaris and Linux. The client is written in Java and thus platform-independent.

In the next sections we describe the components of the Vega system in detail.

3.2 Server

The Vega server is the link between visualizer and algorithms. It offers its services at the TCP/IP port 11003. For every client connection a *dispatcher* process is started, such that one server can handle multiple clients simultaneously. The algorithms supplied by the server are maintained in a simple configuration file that can be changed while the server is running. The configuration also allows to group algorithms into categories and subcategories for easier selection.

The server offers a simple text based protocol that includes the following commands.

- `list`: Send list of supplied algorithms.
- `select`: Select an algorithm.
- `info`: Send algorithm information.
- `input`: Receive input data from the client.
- `run`: Run algorithm and send output data.

When the server runs an algorithm, a separate process is started. The dispatcher sends the input to the algorithm and reads its output. The output is forwarded immediately to the client. If the algorithm crashes or gets stuck in an infinite loop, the dispatcher detects a timeout, stops the algorithm process and notifies the client. Thus, the reliability of the visualization system is not compromised by a malfunctioning algorithm. Furthermore, the client can sent a break signal to the dispatcher, who in turn also terminates the algorithm process.

3.3 Algorithms

In the Vega system algorithms are separate executable programs that communicate through standard I/O. When started, a Vega algorithm prints an identifier string and enters a dispatch mode that recognizes three commands.

- `info`: Write the algorithm information text.
- `input`: Read algorithm input data.
- `run`: Run the algorithm and output data.

Typically, the algorithm is started by the Vega server through the system calls `fork` and `exec`, but it can also be operated interactively in a shell or by other programs.

Algorithms may be created using any programming language, but C++ is the most convenient way, because there is a library that supports the adaption of algorithms to Vega and supplies the following features.

- A Framework for a straight conversion of existing algorithms to Vega
- Routines for reading input and writing output data in the format used by Vega
- Special routines for reading and writing LEDA and CGAL objects directly

3.3.1 Example adaption

In the following we describe the process of adapting existing algorithms to Vega. As an example we choose an algorithm that computes the Delaunay triangulation of a point set using LEDA.

1. Main procedure. Vega header files are included. These define a class instance `vega` that supplies the library methods. The main procedure `main` is renamed and becomes `vegaRun`. Command line parameters have to be replaced by regular algorithm input data. A new `main` procedure is inserted, containing the commands `setAlgorithmName` (optional), `setAlgorithmInfo` and `start`.

```
vega.setAlgorithmName("Delaunay");
vega.setAlgorithmInfo(
  "Algorithm computes the Delaunay\n"
  "triangulation of a point set.\n"
  "INPUT\n"
  "Points sites: The sites.\n"
  "ATTRIBUTES\n"
  "output.delaunay.editable=false\n"
  "output.delaunay.color=red\n"
  "input.pointSize=7\n"
  "input.color=blue\n");
vega.start();
```

The first token in the `INPUT` specification is a geometric class identifier, e.g. Points. If this is in plural, the next token is the name of a group, otherwise the full name of a single object. After a colon a description for the user may be added.

2. Receiving input. Current input methods (stdio, file, interactive) are replaced with calls to `getPoint`, `getSegment`, and so on. These methods are overloaded and can thus be used to read the object's coordinates in `double` variables or create a LEDA or CGAL object. For anonymous objects input iterators can be used that enumerate objects of a given class in a group.

```
VegaInputIterator it=
  vega.input->begin("sites","Point");
while(it!=vega.input->end()) {
  vega.getPoint(p,*it++);
  sites.append(p); // add to internal list
}
```

3. Hierarchical step division. The algorithm is divided into steps by calling `step`. With `beginSubsection` and `endSubsection` the hierarchical structure of the steps can be defined.

```
vega.step("merge_halves");
vega.beginSubsection();
while (next_candidate) {
  vega.step("merging");
  ...
}
vega.endSubsection();
```

4. Sending output. Current output methods are replaced by calls to `sendPoint`, `sendSegment`, and so on. These methods are overloaded as well and can send various formats of geometric objects.

```
forall_edges(e,G) {
  node s=G.source(e);
  node t=G.target(e);
  vega.sendSegment(segment(G[s],G[t]),name);
}
```

Our experience shows that simple algorithms like the Delaunay triangulation can be adapted to Vega in a matter of one or two hours.

3.4 Client

The Vega client is the visualizing workbench of the Vega system. It can be used to

- Manipulate scenes with hierarchically named geometric objects—either in the view or in an object browser that displays the object tree, see Figure 8,
- View algorithm lists of Vega servers and display algorithm information,
- Apply algorithms to selected geometric data in a view, control the algorithm execution using a VCR-like panel or the execution browser, see Figure 6,
- Adjust view attributes directly or using the object browser, show several algorithms simultaneously in multiple scenes and open different views for a single scene, and
- Load and save single scenes, complete runs, and attribute lists, export scenes in `xfig` or `gif` format.

The client is available as Java application or applet. As an applet it may be included in any multimedia document based on HTML. Both versions support a number of parameters that can be used for the automatic execution of a selected algorithm on selected input data. Thus, self-contained demonstrations may be incorporated into other documents. Compared to traditional movies these have the advantage that the user can stop the pre-programmed execution and start experimenting on his own.

The interface of the client is incorporated into its *view windows*, see Figure 9. These windows are used for visualizing and manipulating geometric objects. When an algorithm is selected, its information data is being visualized in an information window, see Figure 10. The user may define input data by simply clicking on an object in the information window and then drawing the object in a view window.

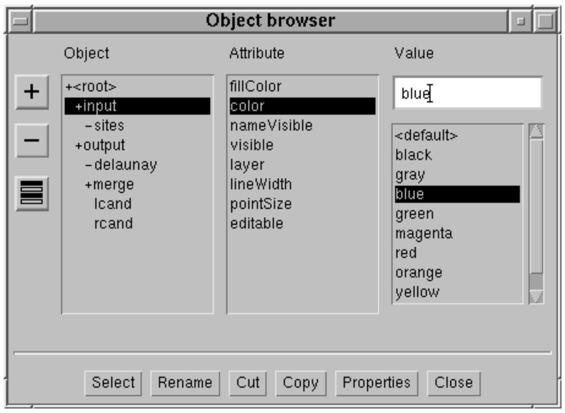

Figure 8: The object browser.

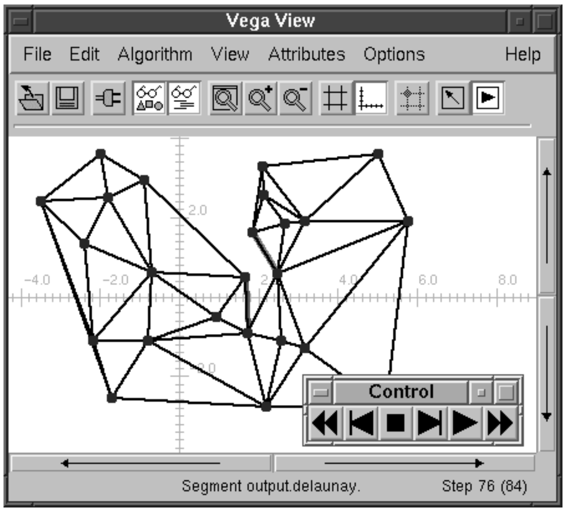

Figure 9: Vega client showing the delaunay triangulation algorithm while merging.

An important aspect of the implementation of the Vega client is that is uses *parallel threads* for receiving and visualizing the incoming data of the algorithm. Thus, the user can view the first steps of the execution immediately and does not have to wait until the transmission is complete.

As suggested by the Java language the client is implemented in an object-oriented fashion and makes use of *design patterns* [5]. The classes `Scene`, `View`, and `EditView` implement the *Model/View/Controller* pattern, separating an entity, its visualization and manipulation. The hierarchical structure of the scene is created with the *Composite* pattern, i.e. all geometric objects are derived from a common ancestor `GeomObj` and the group class that may contain `GeomObjs` is derived from `GeomObj` as well. For updating views and browsers the *Observer* pattern is used. Using design patterns significantly increases modularity and re-usability of software systems.

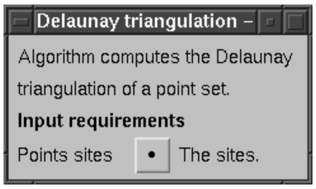

Figure 10: Algorithm information window.

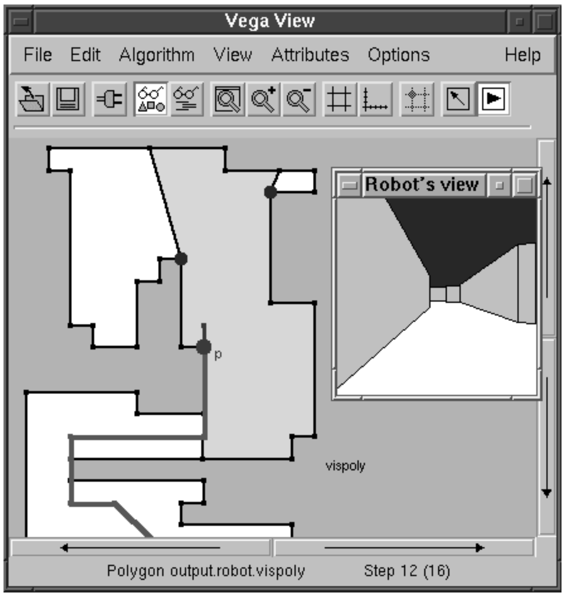

Figure 11: Vega client showing a search strategy of a robot inside a polygon. The window showing the view of the robot in 3D is a *plug-in*, a dynamic extension of the visualizer on the client side.

4 Conclusions

We have presented concepts for the visualization of geometric algorithms that enable interaction in a workbench-like distributed environment. The Vega system implements these concepts. Existing algorithms are visualized by source code annotation. The visualization is highly configurable by the end user.

Further research includes the extension of our concepts to the *animation* of geometric objects and the integration of Vega algorithms into multimedia authoring systems.

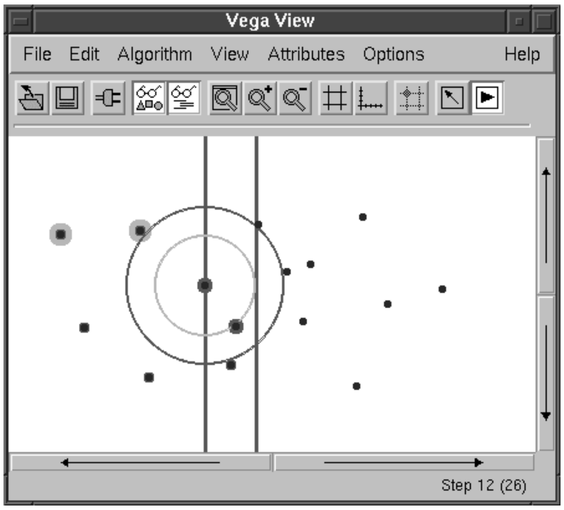

Figure 12: Vega client showing a divide and conquer strategy for finding the closest pair.

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*, pages 203–212, 1996.

[2] Marc H. Brown. *Algorithm animation.* MIT Press, 1988. ACM distinguished dissertations.

[3] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Visual Languages '91*, pages 4–9, October 1991.

[4] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11(4):404–428, 1994.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Elements of Reusable Object Oriented Software.* Addison-Wesley, 1st edition, 1995.

[6] D. T. Lee, C.-F. Shen, and D. S. Sheu. GeoSheet: A distributed visualization tool for geometric algorithms. *J. of Computational Geometry & Applications*, 8(2):119–155, April 1998.

[7] Kurt Mehlhorn and Stefan Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, January 1995.

[8] M. H. Overmars. Designing the computational geometry algorithms library CGAL. In *Workshop on Applied Computational Geometry, Philadelphia, Pennsylvania*, May 1996.

[9] P. Schorn. The XYZ GeoBench for the experimental evaluation of geometric algorithms. In *Computational Support for Discrete Mathematics: DIMACS Workshop*, pages 137–152, 1992.

[10] J. T. Stasko. Animating algorithms with XTango. *SIGACT News*, 23(2):67–71, 1992.

[11] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proc. IEEE Workshop on Visual Languages*, pages 3–10, 1992.