Alcom-FT Technical Report Series
ALCOMFT-TR-01-22

Visualizing Algorithms over the Web with the Publication-driven Approach*

Camil Demetrescu†, Irene Finocchi‡, and Giuseppe Liotta§

March 15, 2001

Abstract

We propose a new approach to algorithm visualization over the Web, called *publication-driven approach*. According to the publication-driven approach, algorithms run on a developer's remote server and their data structures are published on blackboards held by the clients. Animations are specified by attaching visualization handlers to the data structures published on the client's blackboard: modifications to these structures, due to the remote algorithm execution, trigger the running of the corresponding handlers on the client's side. The publication-driven approach has been used for designing an algorithm visualization facility over the Web, called *WAVE*. A first prototype of *WAVE* is available at the URL `http://www.dis.uniroma1.it/~wave`.

1 Introduction

Algorithm visualization concerns the high-level graphical display of both the control flow and the data flow of running programs [16]. The visualization of an algorithm and in particular the pictorial representation of the evolution of its data structures has several applications both in the algorithm engineering and in the educational field [2, 9]. Indeed, a complex algorithm can be described not only by presenting a commented piece of code, but also by associating geometric shapes with the algorithm's data and by showing how these shapes move and change as the algorithm implementation is executed. The latter capability is not only an effective tool for algorithms researchers who want to share and disseminate their ideas, but also provides a valuable help for discovering degeneracies, i.e., special cases for which the algorithm may not produce a correct output. Moreover, teachers may benefit from the support of algorithm visualization to describe the behaviour of an algorithm with a more attractive and communicative medium than the traditional blackboard.

*Work partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the project "Algorithms for Large Data Sets: Science and Engineering" of the Italian Ministry of University and Scientific and Technological Research (MURST 40%) and by the project "Geometria Computazionale Robusta con Applicazioni alla Grafica e al CAD" of the Italian Research Council.

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy, Tel. +39-06-4991-8442, demetres@dis.uniroma1.it

‡Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy, Tel. +39-06-4991-8430, finocchi@dsi.uniroma1.it

§Dipartimento di Ingegneria Elettronica e dell'Informazione, Università di Perugia, Via G. Duranti 93, 06125 Perugia, Italy, Tel. +39-075-5852685, liotta@diei.unipg.it

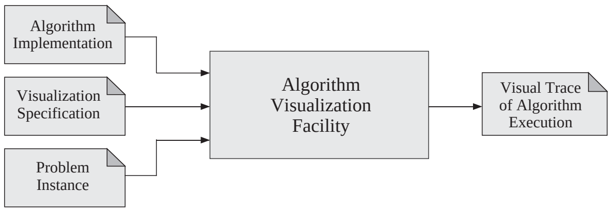

Figure 1: Algorithm visualization facility seen as a “black box”.

A lot of effort has been put in recent years to designing and developing *algorithm visualization facilities* (see, e.g., the systems reviewed in [11, 16] and [2, 4, 5, 6, 7, 8, 10, 17]). An algorithm visualization facility, seen as a “black box”, is shown in Figure 1: it receives as input the implementation of an algorithm, an instance of the input of the algorithm, and a specification of the visualization of the algorithm, that is a mapping from the set of algorithms data to a set of geometric shapes that will be displayed; the output is a *visual trace* of the algorithm execution, i.e., a sequence of moving images that describe the evolution of the data at-run-time.

There is a general consensus that algorithm visualization facilities can strongly benefit from the potentialities offered by the World Wide Web. Indeed, the use of the Web for easy communication, education, and distance learning can be naturally considered a valid support for improving the cooperation between students and instructors, and between algorithm engineers . Valuable features of an effective Web-based algorithm visualization facility are:

Task Decoupling: The algorithm visualization facility should help with separating the roles of *Algorithm Designer*, *Algorithm Developer*, *Algorithm Visualizer*, and *End-user* [16], and with making their interaction easier and faster.

Language Independence: Algorithm developers usually want to implement algorithms in their favourite programming language. Therefore, an effective algorithm visualization facility should be able to accept programs written in different languages and to allow the algorithm visualizer to design a visualization independently of the algorithm implementation language.

Customizability: Algorithm visualizers should be given the possibility of customizing the visualization and to design different graphical interpretations of the same data structures without having neither a detailed knowledge of the source code nor ability with computer graphics concepts and libraries.

Code protection: Since developers may want to keep privacy and protection on their code, visualizations should be designed without downloading it.

User-friendliness: It should be easy to learn how to use the algorithm visualization facility for all the different types of users. The interaction with the visualization facility should be supported by a friendly environment.

The simplest approach to algorithm animation over the WEB consists of exploiting directly the graphic capabilities offered by the Java language without making use of any algorithm animation facility. In particular, it is very common to find out Java applets over the WEB consisting of the implementation of a specific algorithm annotated with hooks to Java visualization methods. Several examples can be found at the URL `http://www.cs.hope.edu/~alganim/ccaa /site.html`. Such animations usually portray the execution of the algorithm on predefined test sets or on problem instances specified by the end user that interacts with the applet. Unfortunately, although some of these animations are of very good quality due to the high level of customizability offered by Java, this approach has several disadvantages with respect to the above features: it is very time consuming, requires algorithms to be written in Java, guarantees no protection of the code, and assumes the algorithm developer to be the same person as the algorithm visualizer. Even those systems explicitly designed to support visualization over the Web are either targeted for particular classes of algorithms, or do not completely satisfy the aforementioned requirements. In the following we give a few examples.

JEliot [10] aims at making automatic the task of producing visualizations of Java programs. It parses the Java code and allows the user to choose the cast of variables to visualize on the stage according to built-in graphical interpretations. Changes to the variables are automatically detected as the program runs and the visualization is coherently updated. JEliot relieves the user from writing any visualization code and so it is very easy to use, but does not support a high level of customizability. JDSL [2] is a library of data structures in Java which features a visualizer for animating operations on abstract data types such as AVL Trees, Heaps, and Red-Black Trees. It is well suited for educational purposes as students are allowed to write and test their own classes provided they implement specific JDSL Java interfaces. In particular, constructors and selectors of the data types must be implemented as methods having predefined signatures. This allows JDSL visualizer to invoke them as the student interacts with the system for testing her implementation. The visualization of supported data types is embedded into the library. VEGA [12] is a C++ client/server visualization environment especially targeted for geometric algorithms. The algorithm is executed on the server; the client runs on any Java Virtual Machine and a small bandwidth communication interface guarantees good performance even in slow networks. The end-user can interactively draw, load, save, and modify graphical scenes, can visualize algorithms on-line or show saved runs off-line, and can customized the visualization by specifying a suitable set of view attributes.

In this paper we propose a new approach to algorithm visualization on the Web, called *publication-driven approach*. According to the publication-driven approach, algorithms run on a developer's remote server, while their visualization can be realized and viewed through any Java-enabled Web-browser. The visualization is specified with respect to an intermediate set of *public* data structures that are a coherent copy of a subset of the program's data structures. In the following we call this intermediate set of data structures *public blackboard*. With the publication-driven approach, an algorithm visualization is realized by performing two steps: (1) a set of program's variables, containing all the information to be conveyed into the visualization, is chosen for publication on the public blackboard; (2) visualization handlers are attached to the variables of the public blackboard and are executed whenever the variables themselves change. These handlers may be chosen from a library of predefined ones, yet experienced users are allowed to write new handlers in a suitable visualization language. Modifications to the variables in the public blackboard, due to the remote algorithm's execution, trigger the running of the corresponding handlers on the client's side. We remark that a concept similar to that of public blackboard has been used in the field of computational steering where, in the tradi-

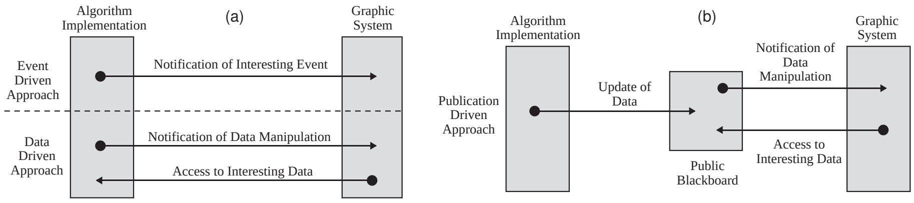

Figure 2: (a) Event-driven and data-driven approaches to visualization; (b) publication-driven approach.

tional cycle of a simulation (i.e., preparing input, executing a simulation, and visualizing the results as a post-processing step), researchers change parameters of their simulations on the fly and immediately receive feedback on the effect [13].

The publication-driven approach has been used to design the architecture of an algorithm visualization facility called *WAVE* (Web Algorithm Visualization Environment). A first prototype of the system is currently being implemented and experimented. The interested reader can find further information at the URL `http://www.dis.uniroma1.it/~wave`.

2 The Publication-driven Approach

In this section we describe the main ideas that underlie the publication-driven approach and compare it with other existing approaches. As shown in Figure 2(a), the approaches to algorithm visualization described in the literature can be roughly classified into two main groups, i.e., *event-driven* and *data-driven* approaches, according to the way the visualization events are triggered and the information to be visualized are gathered. Furthermore, systems that allow users to customize the visualization by means of a suitable language can be classified according to the paradigm of the visualization language, i.e., imperative, declarative, constraint-based, or object-oriented.

The *event-driven approach* was pioneered by BALSA [5] and has been subsequently followed by many other systems: visualizations are realized by identifying *interesting events* in the source code that correspond to relevant steps of the underlying algorithm, and by annotating the code with calls to visualization routines of the graphic system (these routines are usually written in imperative or object-oriented languages). The event-driven approach allows the algorithm visualizer to easily customize the visualization according to her needs and appears simple to understand; however, even the visualization of moderately easy and short programs may require several lines of additional code.

In the *data-driven approach* visualization events are triggered by modifications to the content of the data structures performed by the program during its execution. The main motivation behind this approach is that observing how the data change during the execution of an algorithm is usually enough to understand the algorithm behavior. The data-driven approach relieves the algorithm visualizer from the task of identifying interesting events and makes it possible to concentrate only on the data structures of the algorithm, thus achieving high levels of abstraction and automation. In spite of its flexibility, only few algorithm visualization facilities have explored its potentialities. Existing systems are either automatic [10] or based on a declarative [15] or logic-based [7] specification of the visualization.

WAVE is a data-driven imperative algorithm visualization facility. We call the approach that underlies WAVE *publication-driven approach*. As shown in Figure 2(b), the publication-driven approach introduces an software layer, called *public blackboard*, between the algorithm implementation and the graphic system. The public blackboard maintains a copy of those data structures that are the subject of the visualization. Each time a data is modified by the program, the copy of this data in the public blackboard is coherently updated. Also, the public blackboard notifies the update to the graphic system, which accesses the modified data in the public blackboard itself and updates the visualization accordingly.

A data structure that has a copy on the public blackboard is said to be *published* on the blackboard. Data structures are published by means of an *algorithm publication mechanism*, which consists of annotating the source code with meaningful events (i.e., variable declarations, changes, and going out of scope) and of mirroring such events on the public data. The annotation of the source code is the task of the *algorithm publisher*, that thus decouples the role of the algorithm visualizer from that of the algorithm developer. The *algorithm visualization mechanism* consists of attaching visualization handlers to the public data structures. The *algorithm visualizer* accomplishes the task of designing these visualization handlers and of associating them with the public data structures. In the sequel we provide some more details on these mechanisms and give simple examples. A more elaborate example is discussed in Section 4.

The Algorithm Publication Mechanism The publication mechanism exploits a suitable publication language whose main goal is to encode *execution traces* instead of the program itself: execution traces are plain descriptions of the published data structures and of their manipulations when the program is executed on a given input. Such traces are the only information about the program that are delivered over the Web, which guarantees a high level of code protection. In a trace, *declarations* of strongly typed public data structures alternate with *assignment* statements and with *un-declarations*. Declarations are introduced by the keyword "`declare`" followed by the name of the variable, by its type, and by its address in main memory. This last information makes it possible to handle dynamic variables and pointers arithmetic. Assignment statements use the operator ":=" and follow a Pascal-like syntax. Un-declarations consist of the name of the variable prefixed by the keyword "`undeclare`" and allow the simulation of the scope rules of the published data structures. Execution traces are generated by capturing the standard output stream of the program, that is sent over the Web for visualization. The link between a program variable and its copy on the public blackboard is created by interpreting these printing statements.

As an example, let us consider a simple C program that fills in an array of pairs of numbers and the same program annotated for publication as shown in Figure 3. In the annotated code, we declare two public variables, named `k` and `a`, associated with the control variable `i` and with the array `v` of the original code, respectively. Observe that only the integer field of array `v` is published, showing that the publication step can be useful for omitting details irrelevant for the comprehension of the algorithm. The values of `k` and `a` are updated each time the values of `i` and `v` change; `k` and `a` are finally un-declared when `i` and `v` go out of scope.

The Algorithm Visualization Mechanism As for the publication mechanism also the visualization mechanism is based on a suitable language, whose syntax is very similar

Original C Code	Annotated C Code

<pre>int i;
struct {int x; float y;} v[100];
 	
void main() {
   for (i=0; i<100; i++)
      v[i].x=v[i].y=i*2;
}</pre>
	<pre>int i;
	struct {int x; float y;} v[100];
	
	void main() {
	   printf("declare k:integer;");
	   printf("declare a:array[100] of integer;");
	   for (i=0; i<100; i++) {
	      printf("i:=%d;",i);
	      v[i].x=v[i].y=i*2;
	      printf("a[%d]:=%d;",i,v[i].x);
	   }
	   printf("undeclare a;");
	   printf("undeclare k;");
	}</pre>

Figure 3: Original and annotated fragment of C code.

to that of the publication language. The algorithm visualizer designs a visualization of a program by performing the following two steps: (1) A set of *visualization handlers* is written in the visualization language. A visualization handler is a routine implemented in the visualization language that can access the variables on the public blackboard and invoke 2-D graphical primitives that draw basic geometric objects, such as disks, lines, points, and so on. (2) The visualization handlers are associated with the events of declaration, assignment, and un-declaration of public variables. This is done by defining a set of *associations variables-handlers*. When a remote manipulation of a variable of the public blackboard takes place, the corresponding visualization handler is automatically executed by the system.

Example of Handlers and Associations Variables/Handlers

```
function ResizeCircle()
begin
   call "size Circle 1 "+a[0]+" 1"
   call "update";
end;

function CreateCircle()
begin
   call "create Circle 1 1";
   call "locate Circle 1 100 100 1";
   call ResizeCircle();
end;
```

```
function DeleteCircle()
begin
   call "delete Circle 1 1";
end;

declare a:   call CreateCircle();
change a:    call ResizeCircle();
undeclare a: call DeleteCircle();
```

Figure 4: Example of handlers and associations variables/handlers for visualizing the program of Figure 3.

As an example, consider the public variable `k` of Figure 3 and suppose that the algorithm visualizer designs a visualization where there is a disk whose radius is equal to the value of `k`. Figure 4 shows the three visualization handlers (i.e., `CreateCircle`, `ResizeCircle`, and `DeleteCircle`) and the three associations variables-handlers that are needed for this task.

3 The Architecture of WAVE

The architecture of WAVE is shown in Figure 5. The interaction between clients and the system takes place through any Java-enabled Web browser and is based on the so called Mocha model [1, 3]: the user interacts with a hypertextual interface which is transferred over the Web to the user's machine together with the code for the interface itself. In this model the visualization is displayed by an applet that runs locally, while the algorithm is

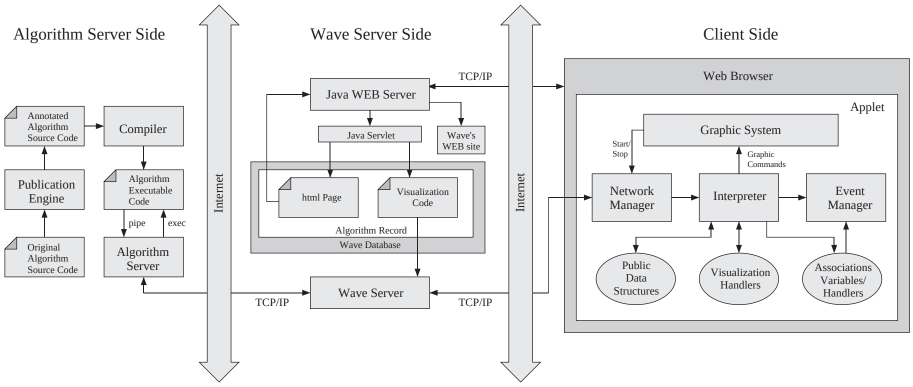

Figure 5: The architecture of WAVE.

executed on a remote server, thus guaranteeing good code protection for the source code and reducing the load on the network. As Figure 5 shows, WAVE works on three sides:

- `Algorithm Developer Side.` Algorithm implementation codes are annotated through a *Publication Engine* (further described in Section 4) and are compiled on the developer's local machine. An *Algorithm Server*, started by the local system administrator, runs the resulting executable codes on client's demand. The standard output of executed programs consists of scripts in the publication language that declare, manipulate, and un-declare variables of the public blackboard, and is sent by the algorithm server to the remote client.

- `WAVE Server Side.` It contains the *WAVE Web site* that is the Web front-end of WAVE Server Side. It also contains the *WAVE Database* that is a collection of *Algorithm Records*. Each Algorithm Record is associated with an algorithm whose visualization is available. The Algorithm Record stores a description of the algorithm together with the set of visualization handlers and the set of associations variables-handlers that are needed for its animation. Finally, the *WAVE Server* acts as a bridge between the Algorithm Server and the Client Side and provides both the visualization handlers and the associations variables-handlers for a specific algorithm chosen from WAVE Database.

- `Client Side`: Users interact with WAVE by running a general-purpose applet, called *WAVE Applet* embedded into ad-hoc Web pages dynamically generated by WAVE Web Server. The main modules of *WAVE applet* are: a *Graphic System*, which allows end-users to stop, restart, pause, step, and continue a visualization and provides 2D graphic capabilities; a *Network Manager*, which allows WAVE Applet to make connections over the Internet; an *Interpreter*, which parses and executes the statements in the publication and visualization languages, accesses information in the public blackboard, and issues graphic commands to the GUI; and finally an *Event Manager*, which checks if execution traces coming from the Algorithm Server imply the execution of visualization handlers according to the associations variables-handlers.

The interaction between WAVE and the users changes depending on the different types of users. When an algorithm developer wants to make a new algorithm implementation

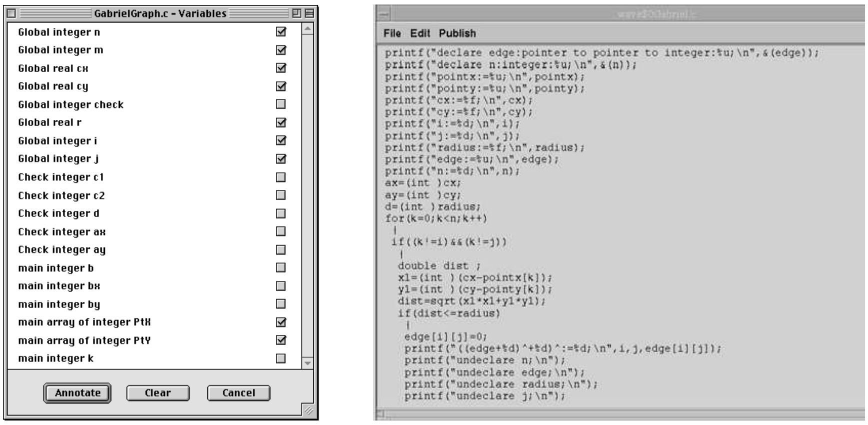

Figure 6: (a) The Publication Dialog; (b) an excerpt of annotated code.

available for visualization, she downloads from WAVE's web site the Publication Engine, which parses the code and asks the developer for those data that have to be published. Also, the Publication Engine automatically generates an annotated version of the source code, that is then compiled by the developer and made available on the Algorithm Server, which runs on the local Server of the developer. The developer is also requested to provide information about the new algorithm, such as the name of the algorithm, the Internet address/port number of the Algorithm Server, and the algorithm executable file pathname. All these information are stored in an algorithm record of WAVE Database associated with the new algorithm. The design of a visualization for the new algorithm is now the task of the algorithm visualizer, who adds visualization handlers and associations variables-handlers in the Algorithm Record. The visualization handlers can be either chosen among those of a Visualization Library, or can be new handlers designed ad-hoc for the specific animation. There can be more than one animation for a given algorithm. Finally, end-users can access the list of algorithms that are available for visualization in the WAVE Database and select one of them. end-users are given the possibility of choosing among the different possible visualizations that have been designed for an algorithm and also can customize the chosen animation by changing some parameters of the visualization handlers. For example, they can change color, thickness, or shape for a geometric object specified in a visualization handler. The next section illustrates an example of interaction between Wave and its users.

4 Interacting with WAVE

Based on the architecture described in Section 3 we have developed a prototype of WAVE, currently available at the URL `http://www.dis.uniroma1.it/~wave`. The example of interaction with the prototype of WAVE that we are going to show concerns the visualization of an algorithm for computing the *Gabriel graph* of a set of point in the plane [14]. Given a set S of distinct points in the plane, the Gabriel graph of S is a geometric graph whose vertices are the elements of S and such that there is an arc between two vertices P_1 and P_2 if and only if the disk whose diameter is the segment P_1P_2 does not contain any other element of S. In our example, we refer to a brute-force algorithm for computing the Gabriel graph of S that checks all pairs of vertices to define the edges. A possible

Handlers

```
function CreateCircle(c:integer)
begin
   call "create Circle "+c+" 1";
   call "locate Circle "+c+" "+cx+" "+cy+" 1";
   call "size Circle "  +c+" "+r+" 1";
   call "color Circle " +c+" 1 0 0 1";
   call "layer Circle " +c+" 0 1";
   call "update";
end;

function EraseCircle(c:integer)
begin
   call "delete Circle "+c+" 1";
end;

function CreatePoints(nodes:integer)
begin
   if(nodes>1) then call CreatePoints(nodes-1);
   call "create Point_ "+nodes-1+" 1";
end;
```

```
function LocatePoint(node:integer)
begin
   call "locate Point_ "+node+" "+
                          PtX[node]+" "+
                          PtY[node]+" 1";
   call "layer Point_ " +node+" 1 1";
   call "update";
end;

function AddArc(arc:integer; start:integer; end:integer)
begin
   call "create Line "+arc+" 1";
   call "locate Line "+arc+" "+PtX[start]+" "+
                               PtY[start]+" 1";
   call "size Line "  +arc+" "+PtX[end]+" "+
                               PtY[end]+" 1";
   call "layer Line "+arc+" 2 1";
   if(arc>0) then call EraseCircle(arc-1);
   call CreateCircle(arc);
   call "update";
end;
```

Associations Variables/Handlers

```
declare PtY:      call CreatePoints(n);
change PtY:       call LocatePoint(i);
undeclare PtY:    call EraseCircle(s);
change m:         call AddArc(s,i,j);
```

Figure 7: Visualization code related to the public variables of Figure 6.

visualization consists of showing for each edge found by the algorithm the corresponding disk. We assume in this example that the developer encodes the algorithm in the C language. When the developer submits the code to the Publication Engine, the latter returns the Publication Dialog shown in Figure 6(a) with the list of variables used in the program. The developer can now choose which variables she wants to be published and described in the program's execution trace. In our example, we assume that the developer selects, among others, the following variables in the Publication Dialog: the number of points `n`; the number of arcs `m` (the value of `m` is originally equal to 0, but is progressively incremented as the algorithm runs); the two arrays of size `n`, `PtX` and `PtY` containing the x-coordinates and the y-coordinates of the points, respectively; the indices `i` and `j` that refer to the pairs of points that are progressively considered by the algorithm (the value of these indices range from 0 to $n-1$); the coordinates `cx` and `cy` and the radius `r` of the circle considered by the algorithm at each step. After variables have been selected in the publication dialog, the Publication Engine parses again the code and automatically annotates it with messages in the publication language. Figure 6(b) shows an excerpt of annotated code.

The task of the algorithm visualizer is now to design the visualization of the program by defining visualization handlers and associations variables-handlers for the variables that have been selected in the Publication Dialog. An example of Visualization handlers and Associations variables-handlers is shown in Figure 7. The visualization handlers in Figure 7 are as follows:

- `CreatePoints` creates a set of points with cardinality equal to the value of the input parameter *nodes*. These points are numbered from 0 to $nodes-1$.
- `LocatePoint` locates in $(PtX[node], PtY[node])$ the point identified by the number *node*. The point is also assigned with a layer that controls its visibility with respect to other graphical objects covering the same position.
- `AddArc` creates a new line. The line is identified by number *arc* and goes from point *start* (located in $(PtX[start], PtY[start])$) to point *end* (located in $(PtX[end], PtY[end])$). This function also erases the circle drawn at the previous step of the program's execution (if any), and draws the new circle.

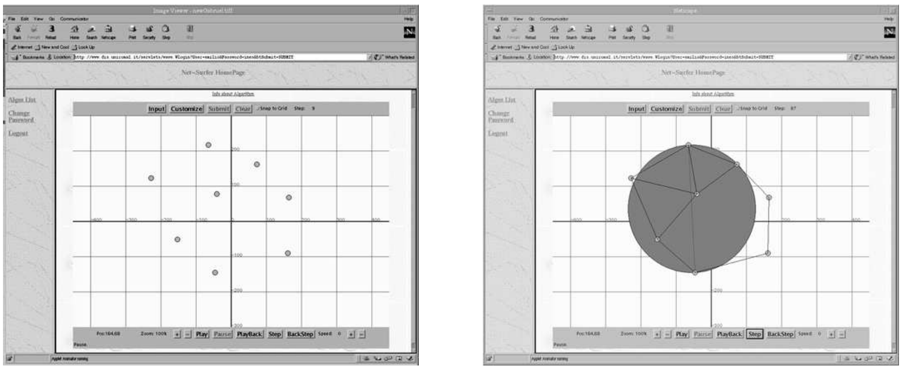

Figure 8: (a) A WAVE Applet for editing the input of a program that computes a Gabriel Graph. (b) Running the visualization of the Gabriel graph program.

- `DrawCircle` creates a new circle centered in (cx, cy) and having radius equal to r. The circle is colored red.

- `EraseCircle` erases the circle identified by the number c.

The visualization is now ready to be run by an end-user. Figure 8 describes the interaction between the end-user and WAVE. Figure 8(a) shows a WAVE Applet with a canvas that allows the User to graphically define a set of points. The applet provides basic features for editing the input, such as snapping the points to the grid, moving them on the canvas, zooming in and out, and saving the input on a file. Figure 8(b) shows the running visualization: at any pair of points considered by the program there corresponds the visualization of a disk on the canvas. If the disk does not contain any other points, then it is colored green and the segment connecting the points is chosen to be an edge of the Gabriel graph; otherwise, the disk is colored red and the the segment connecting the points is not an edge of the Gabriel graph. The end-user can play back and forth the visualization at different speeds and can also review the visualization proceeding in a step-by-step mode. Also, the end-user is given the capability of customizing the visualization by clicking on the "customize" button and editing the visualization handlers; a possible customization for this example is changing the color of the disks.

5 Conclusions and Future Work

In this paper we have presented a new approach to algorithm visualization over the Web, called publication driven approach. We have also described the architecture and the prototype of WAVE, an algorithm visualization facility based on the publication-driven approach. The current prototype of WAVE has supported our research showing the feasibility of the publication-driven approach.

There are still several features that we would like to implement in the system and some design issues that we would like to explore. In particular, we would like to extend the publication engine, which currently supports C, with the possibility of supporting C++ and Java in a near future. Within the object-oriented paradigm, it could be also interesting to explore other ways of implementing the publication mechanism, e.g., encapsulating publication statements into classes. We finally planned to study the visualization of special families of algorithms, such as graph drawing algorithms.

Acknowledgements We would like to thank E. Di Giacomo who has been cooperating with us in the development of the current prototype of WAVE.

References

[1] J.E. Baker, I. Cruz, G. Liotta, and R. Tamassia. Visualizing geometric algorithms over the web. *Computational Geometry: Theory and Applications*, 12, 1999.

[2] R.S. Baker, M. Boilen, M.T. Goodrich, R. Tamassia, and B. Stibel. Testers and visualizers for teaching data structures. *ACM SIGCSE Bulletin*, 31, 1999.

[3] G. Barequet, S. Bridgeman, C. Duncan, M.T. Goodrich, and R. Tamassia. GeomNet: Geometric computing over the Internet. *IEEE Internet Computing*, 1999.

[4] M.H. Brown and M. Najork. Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 12th IEEE Symposium on Visual Languages (VL'96)*, pages 266–275, 1996.

[5] M.H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, 1984.

[6] G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998.

[7] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2), 2000. Leonardo is available at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[8] C. Demetrescu and I. Finocchi. Smooth Animation of Algorithms in a Declarative Framework. In *Proceedings of the 15th IEEE Symposium on Visual Languages (VL'99)*, pages 280–287, 1999.

[9] M.T. Goodrich and R. Tamassia. Teaching the analysis of algorithms with visual proofs. In *ACM SIGCSE '98*, 1998.

[10] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of User Algorithms on the Web. In *Proceedings of the 13th IEEE Symposium on Visual Languages (VL'97)*, pages 360–367, 1997.

[11] A. Hausner and D. Dobkin. Making geometry visible: an introduction to the animation of geometric algorithms. In *Handbook for Computational Geometry*. Sack and Urrutia.

[12] Ch.A. Hipke and S. Schuierer. VEGA: A user centered approach to the distributed visualization of geometric algorithms. In *Proceedings of the 7-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)*, pages 110–117, 1999.

[13] J. Mulder, J. van Wijk, and R. van Liere. A Survey for Computational Steering Environments. *Future Generation Computer Systems*, 15:2, 1999.

[14] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, 1985.

[15] G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

[16] J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA, 1997.

[17] A. Tal and D. Dobkin. Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):194–204, 1995.