Algorithm Animation Systems for Constrained Domains

Ayellet Tal

Department of Electrical Engineering
Technion – Israel Institute of Technology
Haifa, Israel
ayellet@ee.technion.ac.il
http://www.ee.technion.ac.il/~ayellet

Abstract. This paper presents a conceptual model for designing an algorithm animation system for constrained domains. We define a hierarchy of users and a model for supporting each type of users. The hierarchy includes naive programmers, advance programmers, end users, and groups of end users. This paper also describes a few systems that realize the conceptual model within two domains: the domain of computational geometry and the domain of distributed algorithms.

1 Introduction

A major challenge in the area of algorithm animation is how to build systems which significantly facilitate the creation of algorithm visualizations regardless of the algorithms' complexity. One possible solution is to restrict the domain the algorithm animation system is designed for. In a constrained domain, knowledge regarding the type of objects and the type of operations prevalent in this domain, can be embedded into the system. As a result, the system can provide built in ways to visualize these objects and to animate the operations on them. Thus, large parts of the programmer's tasks can be automated.

This automation allows the programmer to be free from having to design and implement the visual aspects of the animation. Instead, the system makes decisions about how graphics is done. This is the major departure from general-purpose algorithm animation systems [3–6, 18, 19] where no assumptions are made regarding the building blocks that make up the animations. In this latter case, the programmer needs to specify the exact shape of each object and the exact transition each object goes through during the animation.

Freeing the programmer from implementing the visual aspects has a couple of benefits. The animation can be produced very fast, usually in a matter of days or even hours, since large parts of the process are left for the system to resolve. Moreover, this is done regardless of the complexity of the algorithm being visualized. This is an important point because complicated algorithms are those that gain the most from visualization.

As expected, the major disadvantage of automation is a limited flexibility. Not every algorithm can be animated and not every animation can be generated

for a given algorithm. Often, users want to have a say in how the animation should look. Choices made by the system might not necessarily be appealing to the user, who might find an automatic system too restrictive. We will show in the sequel that a certain amount of flexibly can be obtained nevertheless.

In our conceptual model we define a hierarchy of users. Naive programmers care only about the contents of the visualization and are not concerned with the presentation aspects. Advanced programmers want, in addition, to be able to easily modify and extend various visualization aspects. End users experiment with an algorithm to understand its functioning, and should be able to run the animation as an interactive experience. Finally, groups of end users should be able to collaborate with each other.

This conceptual model has been realized in three algorithm animation systems built for two constrained domains. GASP [21] and GASP-II [17] were built for the domain of computational geometry; VADE [12] was built for the domain of distributed algorithms. These two domains were chosen because they represent domains in which algorithms are difficult to comprehend.

In the geometric domain even the simple task of imagining in one's mind a three-dimensional geometric construction can be hard. In many cases the dynamics of the algorithm must be understood to grasp the algorithm and even a simple animation can assist the geometer. Degeneracies and robustness problems, which are common in geometry, add to the complexity of programming and debugging geometric code. The visual nature of geometry makes it one of the areas of computer science that can benefit greatly from visualization.

Distributed algorithms are difficult to understand due to the added complexity of the inter-process communication and synchronization. Many activities occur concurrently at various sites. Moreover, the activities depend on each other in many ways. Each state depends not only on the individual process, but also on the messages arriving from other processes. Visualization can give some insight to the way this inter-process communication takes place.

We will show below that though the systems differ, they follow the same conceptual model. In the next section we describe the general conceptual model. In Section 3 we present the realization of the model in the domain of computational geometry. In Section 4 we discuss the realization of the model in the domain of distributed algorithms. We conclude in Section 5.

2 Conceptual Model

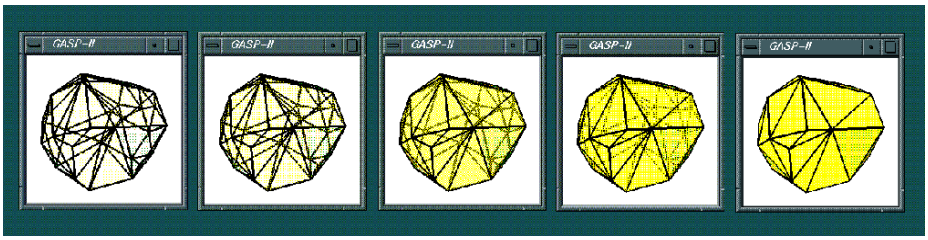
We define four sets of users of any algorithm animation system. They are naive programmers, advanced programmers, end users, and groups of end users. Naive programmers can choose to be isolated from any decisions of a graphical nature and concentrate solely on the contents of the animation. Advanced programmers want to be able to easily modify and extend various visual aspects of the animation in order to produce animations which better fit what the programmers think is most useful. End users experiment with an algorithm animation in order to understand the algorithm's functioning. They should be able to run

the animation as an interactive experience. Finally, groups of end users wish to collaborate.

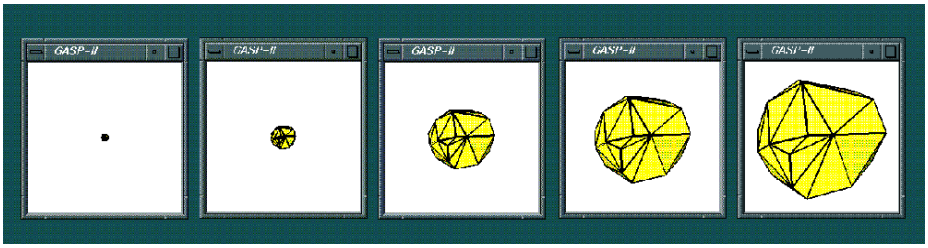
Accordingly, the algorithm animation system should provide an appropriate interface for each of these users. Domain-dependent libraries should be provided for naive programmer. An external graphical user interface should be given to the advanced programmers in order to facilitate the modification of the visual aspects. An interactive environment should let end users run the animation. Finally, tools for collaboration should be provided for groups.

In order to allow naive programmers to generate animations quickly, we distinguish between what is being animated and how it is animated. The naive programmer need only specify what is being animated and need not be concerned with how to make it happen on the screen. For example, the creation of an object is to be distinguished from the way it is made to appear through the animation. It can be created by fading into the scene, by traveling into its location, by scaling up from a point to its full size, etc.

The libraries provided by the algorithm animation system contain a set of building blocks relevant to the domain. The naive programmer should only write short snippets of code that contain calls to functions of the system's libraries. This code includes the contents of the animation and defines its structure. It does not contain the visual aspects of the animation. The algorithm animation system generates an appropriate animation from this code. For instance, only three lines of code are necessary to produce the animation of the creation of the polyhedron shown in Figure 1(a) (or the animation shown in Figure 1(b)).



(a)



(b)

Fig. 1. The creation of a polyhedron (two possible styles)

To enable that, the system's libraries support the objects prevalent in the domain and the common operations on them. In fact, several visualizations, or *styles*, are supported for each operation, one of which is the default chosen by the animation system. If the advanced programmer wishes to change any of the visual aspects of the animation, this programmer can do it through a graphical interface, called the *style panel*. There is no need to modify or write any code.

Note that the animation is still generated automatically by the animation system. But, a different animation is produced, to better reflect the programmer's taste or to better illustrate a specific algorithm. The ease of generating various animations allows advanced programmers to easily experiment with many visualizations for the same algorithm.

Figure 1 illustrates two of the possible styles for creating a polyhedron. In Figure 1(a) the polyhedron is created by fading into the screen, while in Figure 1(b) it is created by scaling up from a point to its full size. Changing one field in the style panel allows the creation of 1(b) instead of 1(a).

End users can experiment with an algorithm animation in an environment such as the one illustrated in Figure 2(a). The environment consists of a control panel through which the execution of the animation is controlled, animation windows where the algorithm runs, and a text window which contains explanations which accompany the animation. It is possible to run the animation forwards and backwards, to fast-forward through some parts of the animation and single-step through others.

Collaboration is useful both in the work place and in electronic classrooms where distance learning is gaining more popularity. In this environment the participants sit each in front of his or her computer and they collaboratively explore an algorithm while viewing its dynamic behavior.

In our model, two modes of distributed visualization are supported: independent visualization and collaborative visualization. In the first mode, each participant controls the animation running on his or her machine and no synchronization is applied. In collaborative visualization, the animations running on all the machines are synchronized with the animation of the instructor. In this case, the participants are more limited. They cannot fast-forward or rewind the animation whenever they wish to. However, they can still modify the style of the animation they view. This is illustrated in Figure 2 where three snapshots of screens taken at the same time are shown. The three participants, in this case an instructor and a couple of students, work collaboratively, yet each is viewing a different animation style. In particular, different colors and transparency values are chosen by the users. Moreover, the user at the bottom views a different unit of the algorithm, since this user rewound the algorithm to a previous atomic unit.

Any participant in the group can initiate an electronic discussion at any point during a collaborative session. An electronic discussion is more than a text exchange since the text is accompanied with an appropriate visualization. This is the way students can ask questions in electronic classrooms and programmers who work together can share ideas.

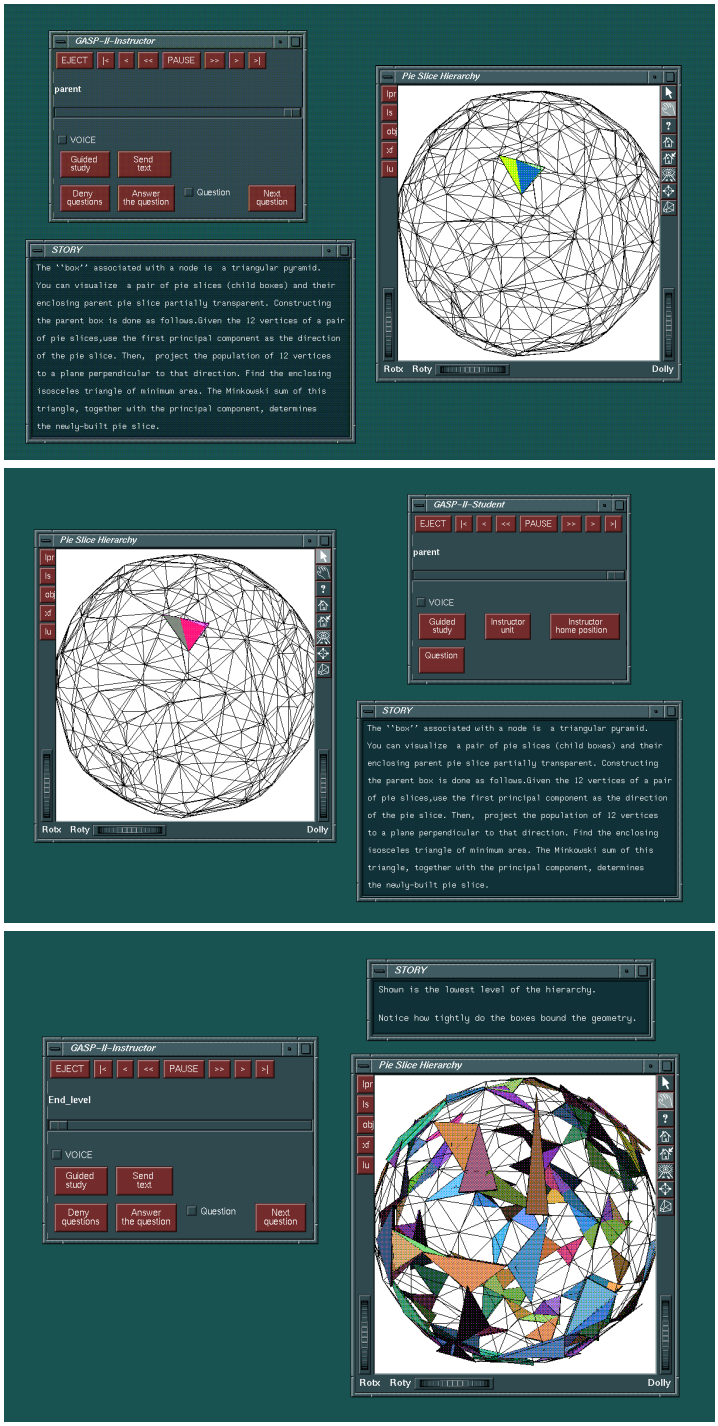


Fig. 2. An algorithm visualization in a distributed session

3 Realization of the Model in Computational Geometry

The last few years have seen a growing awareness to the need and to the importance of visualization in computational geometry [1, 7, 14, 17, 21]. In this domain, the scenes of interest are built out of geometric objects (e.g., polygons, polyhedra, spheres, cylinders, lines and points) and displays of data structures (such as lists and trees of various forms). A standard animation in the domain is built out of operations on these objects.

GASP [21] and GASP-II [17] realize the general model described above. GASP provides the interfaces for the naive programmer, the advanced programmer and the end user. GASP-II's main contributions are the support for groups collaborating over the network and the improvement of the interface provided to advanced programmers.

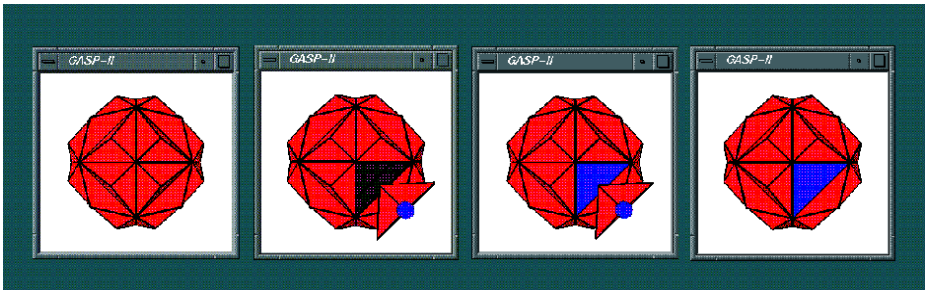
To realize the conceptual model for geometric computation, it is only required to supply appropriate libraries to be used by naive programmers. These libraries provide a rich set of animation tools for visualizing operations common in the domain. The objects contained in these libraries include three–dimensional geometric objects, two–dimensional geometric objects, combinatorial objects, views, text and titles. For example, for modifying polyhedra, the library provides functions for adding faces and vertices, removing faces and vertices, joining polyhedra, splitting polyhedra etc. The animation system can produce various visualizations for every operation, as discussed above.

Rather than going into detail describing the libraries, we will illustrate its use via a case study, which is a visualization of an algorithm for polyhedron realization for shape transformation [15]. This visualization [16] was selected as an example of an algorithm which is not basic. It is one of several animations created by GASP or GASP-II and presented at the video session of the annual symposium of computational geometry.

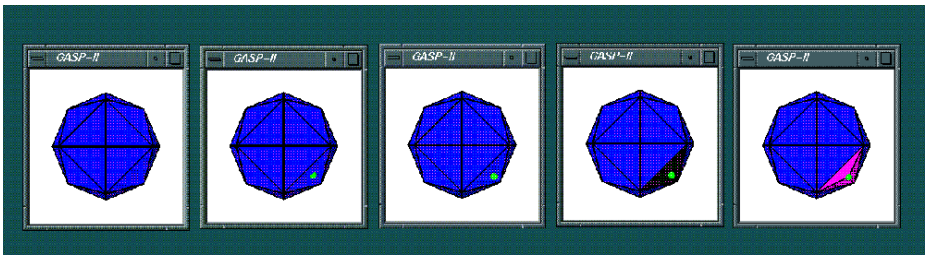
The realization algorithm proceeds in two steps, illustrated in Figures 3(a)-(b). First, each given polyhedron is iteratively simplified by removing a low–degree vertex from the 1−skeleton graph of the polyhedron, and re-triangulating the resulting hole, until a 4–clique graph (representing a tetrahedron) results. During the visualization, which is illustrated in Figure 3(a), the selected vertex blinks in blue. Its cone of faces smoothly drives away from the polyhedron, creating a black hole in it. Next, the hole created is re-triangulated, where the new faces fade-in in blue, to distinguish them from the old (red) faces. Finally, the highlighted vertex and its cone of faces fade out.

During the second step, the process is "reversed" and the vertices are re-attached, this time in a convex fashion. During the visualization, which is illustrated in Figure 3(b), a new vertex, shown in green, appears in a base position. It smoothly drives away from the base to its final position. Then, the blue faces which were previously created while the vertex was detached, fade out. Finally, new faces attaching the vertex to the polyhedron are added, such that convexity is maintained. The new faces are colored magenta to distinguish them from the blue / red faces.

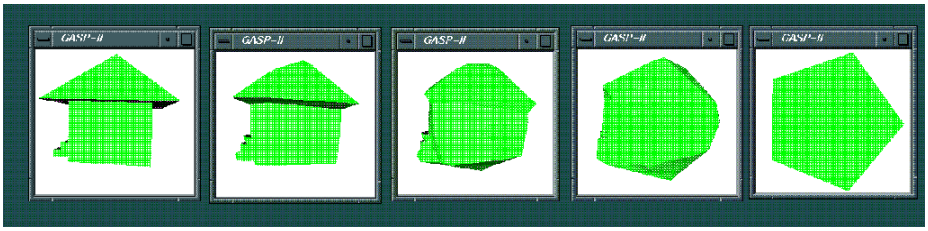
Finally, Figure 3(c) demonstrates a resulting shape transformation, where a house is transformed into an icosahedron. Each transformation is carried out by replacing the old polyhedra with a new one, until the final polyhedra is attained.



(a) The simplification phase



(b) The re-attachment phase



(c) The resulting shape transformation

Fig. 3. Animation of the realization algorithm

The major issue here is how fast it is to generate the animation. The code that generates the animation of the first step (Figure 3(a)) consists of ten lines of C code. It is shown in Figure 4. The code that generates the animation of the second step (Figure 3(b)) is twelve lines of C code. The code that generates the animation of the metamorphosis (Figure 3(c)) is four lines of C code.

The code in Figure 4 consists of calls to functions included in GASP–II's libraries. Note that no parameters of a graphical nature are required. The programmer need only specify the parameters which are related to the geometry (e.g., the vertices and the faces of the polyhedron). All the visual aspects of the visualization are left as empty slots that the algorithm animation system fills up.

```
/* Remove the selected vertex */
GASP_Begin_atomic(split_atomic_name);
  GASP_Split_mesh(new_mesh_names, previous_mesh_name,
                  remove_vert_no, remove_vertices);
  GASP_Write_to_text_view("TEXT",
   "During the simplification phase, a low-degree vertex
    is removed and its vertex-graph is re-triangulated."
GASP_End_atomic();

/* Re-triangulate the hole */
GASP_Begin_atomic(add_atomic_name);
  GASP_Add_faces(new_mesh_names[0], face_no, faces);
GASP_End_atomic();

/* Remove the detached cone */
GASP_Begin_atomic(remove_atomic_name);
  GASP_Remove_object(new_mesh_names[1]);
GASP_End_atomic();
```

Fig. 4. The code of the simplification visualization

For instance, the code does not include the colors used, the number of frames used, the fact that objects are created be fading in, etc. These parameters can later be modified in the external style panel.

4 Realization of the Model in the Domain of Distributed Algorithms

The domain of distributed algorithms is another domain where visualization in one's mind of an execution of an algorithm is highly non-trivial. Moreover, in this domain users can greatly benefit from an algorithm animation.

There is, however, an inherent difficulty in implementing an algorithm animation system within a distributed asynchronous environment. In asynchronous distributed systems it is not possible to determine the precise structure of the execution based on the views of the execution that are obtained by the processes [11]. Each process in the system can only "remember" its own actions. It can also gain knowledge about actions performed by other processes through inter-process communication. It cannot, however, compute the relative timing of the actions performed by different processes. Moreover, no form of interaction among the processes in an asynchronous system can provide precise timing information.

An algorithm animation system cannot overcome these inherent limitations. The animation processes are just more processes in the distributed environment. The goal of an algorithm animation system for distributed environments is thus to produce a visualization that reflects as closely as possible the real execution of the algorithm.

The algorithm being visualized changes dynamically and the animation system needs to receive updates of the state from different sites. Since a true instantaneous snapshot is impossible, a "possible" (or *consistent*) snapshot should be constructed. Roughly speaking, we call a snapshot of the system consistent if

it describes a state that appears in a possible execution of the system that starts in the state at which the invocation of the snapshot actually happened, and ends in the state at which the snapshot algorithm completed its computation.

There exist several algorithm visualization systems for parallel or distributed algorithms [2, 8–10, 13, 20]. Here we briefly present VADE [12], our visualization system which realizes the general conceptual model described above. Unlike the geometric domain, where it is sufficient to supply domain-dependent libraries in order to apply to the conceptual model, here we need also guarantee consistency.

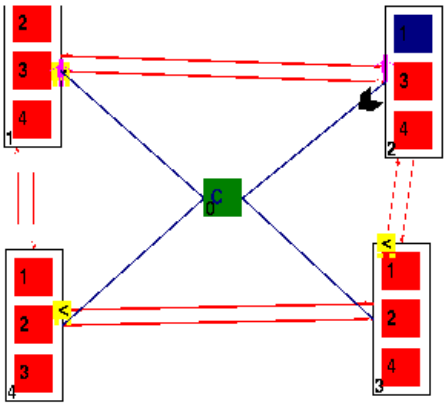
The objects common in the domain of distributed algorithms and thus the ones supported by VADE's libraries are network related objects. They include graphs, nodes, links (i.e., communication channels) and various types of messages and tokens which the nodes exchange over the links. VADE can produce a visualization for numerous operations defined on these objects. Again, the system chooses default styles for the visualization of objects and operations. The style can later be modified in an external style panel.

In [12] we describe the model of computation that our system is defined for. We also define the notion of *visualization consistency* that captures the sense in which the animation system is required to faithfully depict the execution. Using visualization consistency we can rearrange the sequence of events in a run. We discuss a couple of ways to implement visualization consistency.

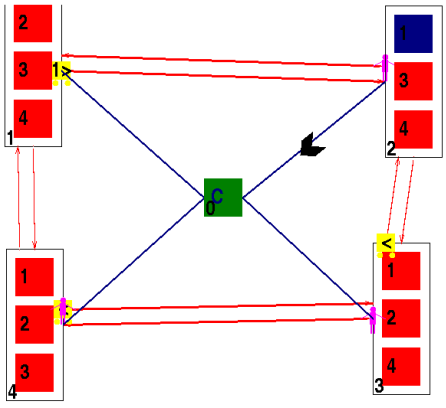
We illustrate the task of creating a visualization by describing a case study in which we simulate an Automatic Transport System (ATS). The ATS consists of a close dual-lane railroad system. The terminals are located along the railroad. Several cars can move on the railroad. In addition, there is a central control station (CCS) which has data links to all the terminals. Each terminal and each car are equipped with a display showing the stops along the cars' route. In front of each car there is an arrow-display which indicates whether it moves clockwise or counter-clockwise.

Figure 5 presents a few snapshots from the animation of this case study. Figure 5(a) shows the system immediately after a passenger arrived at Terminal 2 and pressed the Terminal 1 button. This button changes its color to blue. The terminal sends a request for a car to the CCS, animated by sending a marker over the appropriate link. Figure 5(b) illustrates the marker moving towards the CCS. In Figure 5(c) the request is received by the CCS and cars are sent from Terminal 3 and from terminal 4 to terminal 2. In Figure 5(d) the cars move towards the target terminal. Figure 5(e) illustrates the arrival of another passenger, this time at Terminal 3, which has no available cars. The appropriate button is pressed, and the request for a car is sent to the CCS. Finally, Figure 5(f) shows the arrival of a car at Terminal 2.

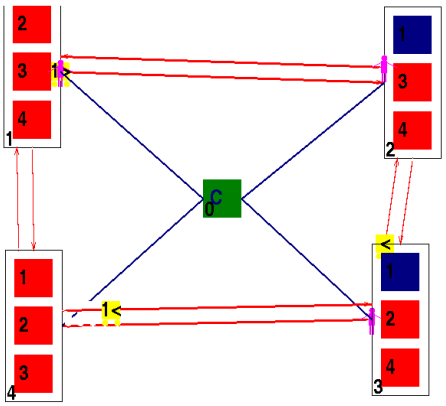
The programmer has very little work to do in order to create the above animation. This is both because the programmer need not be concerned with maintaining the consistency of the animation, and because the programmer is free from dealing with the graphical aspects of the animation which are determined by the system (and can be modified via the style panel). For instance, the code (in Java) animating the pressing of a button is shown in Figure 6.



(a) Terminal 2 requests a car

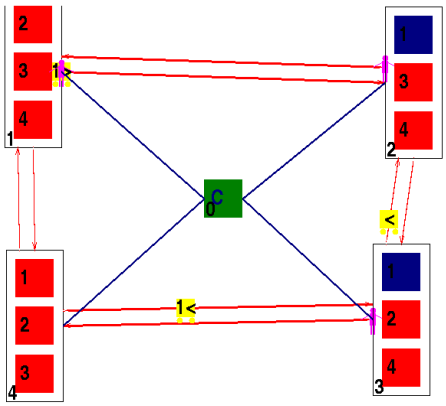


(b) A marker is sent to the CCS

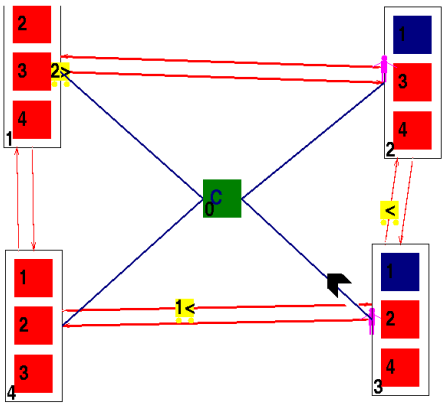


q

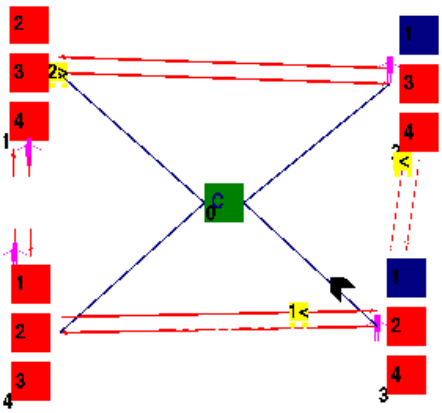
(c) The CCS received the request and cars are sent



(d) The cars move



(e) Terminal 3 is asking for a car



(f) The car from Terminal 3 is arriving

Fig. 5. Snapshots from the ATS animation

```
import ui.*;
import geometry.*;
import network.NetProtocol;
import java.awt.*;

public class TrainPressButton implements AnimationAction {
    static TrainAnimCommonData commData;

    public String perform(StringBuffer args[], int argsNum, AnimationProcess ap){

        if (argsNum != 1){
            return NetProtocol.BAD_ARGUMENTS_NUM_REPLY;
        }
        String neighbour = new String(args[0]);
        NetGraph graph = ap.parent.framedArea[0].GUIScreen.graph;
        int idx = graph.getNode(ap.processIndex).index.intValue();

        NetNode nd = commData.getCurrentNode(graph, neighbour, idx);
        nd.setState(commData.BUTTON_PRESSED, graph.getAnimStyle());

        ap.parent.framedArea[0].GUIScreen.paint(
                    ap.parent.framedArea[0].GUIScreen.getGraphics() );

        return NetProtocol.OK_REPLY;
    }
}
```

Fig. 6. The code of TrainPressButton class

5 Conclusions

We have presented in this paper a conceptual model for constructing an algorithm animation system for constrained domains. In a restricted domain, knowledge regarding the type of objects and the type of operations prevalent in that domain can be embedded into the system, thus enabling the creation of systems that let others use them comfortably.

We have also reviewed systems that realize this model in two specific domains – that of computational geometry and that of distributed algorithms. These domains have been chosen as representatives to domains which can greatly benefit from visualization. In geometry this is because of the difficulties in visualizing in one's mind three-dimensional scenes, along with extensive use of data structures which are heavily pointer-based and problems of degeneracies and robustness. Distributed algorithms are difficult to understand due to the added complexity of the interprocess communication and synchronization.

The major advantage of dealing with constrained domains is the automation it enables. This automation leads to time saving and facilitates the creation of animations for highly complex algorithms. The major disadvantage is that such algorithm animation systems are limited in scope.

In the future we intend to realize this model in other constrained domains, candidates being topology, databases and networks.

References

1. N. Amenta, S. Levy, T. Munzner, and M. Phillips, Geomview: A system for geometric visualization. *Proc. 11th Ann. ACM Symp. on Computational Geometry*,

C12–C13, 1995.
2. T. Bemmerl and P. Braun. Visualization of message passing parallel programs with the TOPSYS parallel programming environment. *Journal of Parallel and Distributed Computing,* 18:118–128, 1993.
3. M.H. Brown. *Algorithm Animation.* MIT Press, 1988.
4. M.H. Brown. Zeus: A system for algorithm animation and multi-view editing. *Computer Graphics*, 18(3):177–186, May 1992.
5. M.H. Brown, M.A. Najork. Collaborative Active Textbooks: A Web-Based Algorithm. Animation System for An Electronic Classroom. *Proceedings of the IEEE Symposium on Visual Languages*, 266–275, 1996.
6. M.H. Brown, M.A. Najork, and R. Raisamo. A Java-Based Implementation of Collaborative Active Textbooks. *Proceedings of the IEEE Symposium on Visual Languages*, 23–26, 1997.
7. A. Hausner and D.P. Dobkin. GAWAIN: Visualizing geometric algorithms with Web–based animation. , *The Fourteenth Annual Symposium on Computational Geometry*, pages 411–412, 1998.
8. E. Kraemer and J.T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing,* 18:105–117, 1993.
9. E. Kraemer and J.T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations *Proceedings of the 8th International Parallel Processing Symposium*, 902–908, 1994.
10. E. Kraemer and J.T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1), 36–46, 1998.
11. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558-565, 1978.
12. Y. Moses, Z. Polunsky, A. Tal and L. Ulitsky. *Algorithm Visualization for Distributed Environments*, IEEE Symposium on Information Visualization '98, October 1998.
13. G.-C. Roman, K.C. Cox, D. Wilcox and J.Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *J. Visual Languages Comput.*, 3(2): 161–193, 1992.
14. P. Schorn. The XYZ GeoBench: a programming environment for geometric algorithms, *Lecture Notes in Computer Science 553* (Springer-Verlag, Berlin, 1991), pp. 187–202
15. A. Shapiro and A. Tal. Polyhedron Realization for Shape Transformation. *The Visual Computer*, 14 (8-9): 429-444, 1998.
16. M. Shneerson, A. Shapiro and A. Tal. Polyhedron Realization and its Application to Metamorphosis. *Fifteenth Annual ACM Symposium on Computational Geometry*, June 1999.
17. M. Shneerson and A. Tal. *Interactive Collaborative Visualization Environment for Geometric Computing*, Journal of Visual Languages and Computing, Vol 6, Num 6, December 2000, 615-637
18. J. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interface. *Journal of Visual Languages and Computing*, pages 213–236, 1990.
19. J. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, September 1990.
20. J. Stasko and E. Kraemer A methodology for building application-specific visualizations of parallel programs. *J. Parallel Dist. Comut.* 18)(1):258–264, 1993.
21. A. Tal and D.P. Dobkin. *Visualization of Geometric Algorithms*, IEEE Transactions on Visualization and Computer Graphics, 1(2): 194-204, 1995.