Concurrent Algorithms and Data Types Animation over the Internet

Giuseppe Cattaneo * Umberto Ferraro*

Giuseppe F. Italiano† Vittorio Scarano*

Abstract

We present a distributed algorithm animation system called Catai *(for Concurrent Algorithms and data Types Animation over the Internet). Among the features of this system are a low effort required for animating algorithmic code, and the possibility of embedding animation clients in standard* `Java`*-enabled Web browsers. We believe this to be a good compromise between two different viewpoints: the programmer's perspective, which typically includes the goal of animating efficiently and unobtrusively a given algorithmic code, and the user's perspective, which can benefit from interactive, easy-to-use, distributed and cooperative interfaces.*

1 Introduction

Algorithm animation is a form of software visualization that uses interactive graphics to enhance the presentation, development and understanding of computer programs. Systems for algorithm animation have matured significantly in the last decade [2, 3, 6, 8, 9, 28, 29, 30], perhaps due to their relevance in many areas, including computer science education, design, analysis and implementation of algorithms, and performance tuning and debugging of large and complex software systems.

Several algorithm animation systems have been developed with one or more of these applications in mind. However, many of these systems require heavy modifications to the source code at hand and, in some instances, even require writing the entire animation code in order to produce the desired algorithm visualization. Thus, a user of these systems not only should invest a considerable amount of time writing code for the animation but also needs to have a significant algorithmic background to understand the details of the program to be visualized. This is not desirable, especially when algorithm animation is to be used in program development and debugging.

*Dipartimento di Informatica ed Applicazioni "R.M. Capocelli", Università di Salerno, Italy.

†Dipartimento di Matematica Applicata ed Informatica, Università "Ca' Foscari" di Venezia, Italy. Supported in part by EU ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244, by a Research Grant from University "Ca' Foscari" of Venice, and by Grants from CNR, the Italian National Research Council. Part of this work was done while visiting The Hong Kong University of Science & Technology. Email: `italiano@dsi.unive.it`. URL: `http://www.dsi.unive.it/~italiano`.

In our own experience, the effort required to animate an algorithm is perhaps one of the main factors limiting the diffusion of animation techniques as a successful tool for debugging, testing and understanding computer algorithms (see also [25]).

This paper describes an algorithm animation system called CATAI (for Concurrent Algorithms and data Types Animation over the Internet). One interesting aspect of our system is that it tries to impose a little burden on the task of animating algorithms. If the program is written according to certain specifics (e.g., in `C++`, using an algorithmic software library such as LEDA [22]), then it should be easy to obtain a simple animation of this program with CATAI.

This should make this system easy to use, and is based on the philosophy that an average programmer or an algorithm developer should not invest too much time in getting an actual animation of the algorithm up and running. In our experience, this was not always the case, and often animating an algorithm was as difficult and as time consuming as implementing the algorithm itself from scratch. Our approach has an advantage over systems where the task of animating an algorithm is highly non-trivial. Producing animations almost automatically, however, can limit flexibility in creating custom graphic displays. If the user is willing to invest more time on the development of an animation, he or she can produce more sophisticated graphics capabilities, while still exploiting the features offered by our system.

We sketch here some other features of our system. One of our design principle was the possibility of animating *complex* algorithms and data structures. This is mainly achieved by fully exploiting an *object-oriented* abstraction, which allows one to build complex animated data structures starting from very simple objects. Particular attention was devoted to *efficiency* issues: the system was designed so as to be resource-efficient (graphical resources are largely reused during the animation), and time-efficient (little extra effort is required when assembling animated objects, as a software layer is in charge of coordinating the presentation). Our system is *distributed*, as animation clients can be placed anywhere on a network, and has a high degree of *interactivity*, as clients are able to have a deep graphical interaction with the running programs, thus influencing the animation. Besides offering interactive and easy-to-use distributed user interfaces, our system can be easily integrated in the Web. Another of the main characteristics of CATAI is that it is inherently *cooperative*: multiple animation clients can interact and influence the behavior of a single shared algorithm instance. Furthermore, our system has a high degree of *privacy*, which is preserved on open networks, and is *not pervasive*, as animation does not interfere with the execution of the algorithm: the animated algorithm keeps overall the same behavior as its original execution. This is particularly important for debugging.

The rest of this paper is organized as follows. We first describe the main principles and architectural choices underlying the design of CATAI in Section 2. Next, we provide some examples on its usage in Section 3, where we describe in details how to prepare an actual algorithm animation with CATAI. The main features of CATAI are summarized in Section 4, where we also mention the next developments, most notably some further integration with the World Wide Web to provide cooperative workgroup capabilities.

2 System Design and Architecture

When our first prototype of CATAI was started, we had already implemented and animated a fair amount of algorithms. Thus, our own experience, both as programmers and as users of algorithms and animations, strongly influenced our views and development of CATAI. This can perhaps explain some bias or some particular choices made in the development of our system,

but at the same time can be seen from a different, and perhaps more practical, viewpoint: we developed CATAI because we actually needed an effective animation tool, especially suited for debugging, testing and tuning algorithmic software.

First of all, being a research group spread over different sites, we wanted to ensure *remote access*, which implied designing a distributed system to animate algorithms. While jointly developing some complex algorithmic software, we wanted to quickly test and visualize the code produced by people in different sites. Thus, we thought of an animated algorithm like any other "network resource", which can be accessed and interacted with. Of course, a system with remote access has many other advantages. Just to mention a few, a remote interface is a natural, simple and often more efficient alternative to porting any special-purpose or specialized software package needed by a given algorithm. Furthermore, remote access can also ensure some form of privacy and encapsulation: the user can watch an algorithm running on a particular input set (which can be chosen by the user), while the algorithm itself and its details are "hidden" from the user, since the code is located on a different machine. This was somehow in line with other current animation systems, such as Mocha [2].

However, and differently from other animation systems, we wished to allow the user to *interact* as much as possible with the animated algorithm. We wanted, for instance, to allow the user to interact with the algorithm *during its execution*. One example could be changing the animated data structures at run-time: in CATAI this is done in such a way that the execution needs not to be restarted (as it happens in other systems) but is simply continued on the changed data. In previous systems, user interfaces were strongly oriented towards the goal of presenting the algorithm. The typical interaction allowed between the user and algorithm was very light (e.g., run the algorithm, present its execution frame-by-frame, control the frame speed, checkpointing). No truly deep interaction was available towards the underlying data structures: the only possibility was to assign default values at initialization time; usually, animated algorithms offered only the choice among different input sets, including defining a new one. In our experience, this gave a rather limited view of interaction, and while visualizing algorithms, we needed a much deeper interaction with their execution. For instance, we found extremely beneficial to use the animation to debug complex algorithmic code: in this case, and for educational purposes as well, we often perceived the need to visualize what happens if we interactively force the algorithm to fall into non-standard or wrong configurations. This requires a much stronger interaction between the user and the execution of the algorithm at animation time. As pointed out in [3], from which we cite verbatim, this strong interaction seems somehow implicit in the concept of algorithm animation:

> "Algorithm animation appeals to the strengths of human perception by providing a visual representation of the data structure [.....] Algorithm animation helps the end-user to understand the algorithms by following visually step-by-step execution".

A crucial point here is the meaning of "step", as in current animation systems one can find nearly two opposite interpretations of this. The first interpretation is the one used in line-oriented debuggers, where coherence with the algorithm execution is guaranteed at the expense of a very fine granularity, which sometimes can even disturb the interpretation of the behavior of the algorithm. The second interpretation of step is that of a logical frame, which can be defined by the programmer at the desired level of granularity (as for instance in POLKA [28]). With this approach, coherence with the algorithm execution is a total responsibility of the programmer and, therefore, it is not always guaranteed. For us, according to an object-oriented paradigm, a step is *any* change of state of the animated objects. Indeed, to ensure a strong interaction

between users and algorithm executions, our system monitors *all and only* the events that change the state of an animated object.

Our third main concern was *reusability*, as previous experiences with this were rather discomforting. In many cases, reusability was not considered at all, and very often the animation of an algorithm was so heavily embedded in the algorithm itself that not much of it could be reused in other animations. We wanted to enforce reusability in a strong sense: if the user produces a given animated data structure (e.g., a stack, a tree, or a graph), then all its instances in any context (local scope, global scope, different programs) must show some standard graphical behavior with no additional effort at all. Of course, when multiple instances of different data structures are animated for different goals, a basic graphical result may be poor without an additional, application-specific coordination effort that by its own nature seems not (and perhaps could never be) reusable. We designed our system so that it can offer different levels of sophistication: non-sophisticated animations can be basically obtained for free. If one wants a more sophisticated animation, for instance by exploiting some coordination among different data structures for the algorithm at hand, then some additional effort is required.

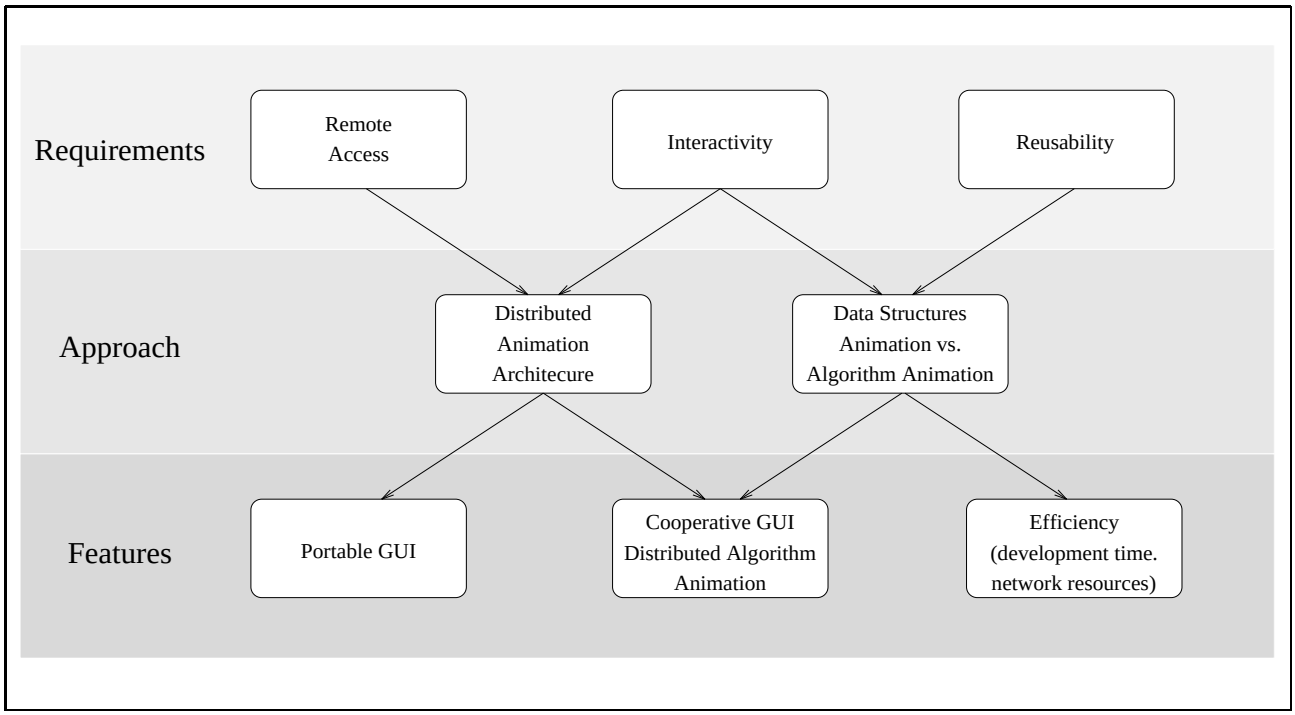

Figure 1: The main ideas behind the design of CATAI.

Figure 1 shows how these three requirements (remote access, interactivity and reusability) deeply influenced the design of our system. To allow remote access, our system had obviously to hinge on a distributed architecture. However, our desire of a high degree of interactivity as well had a profound influence on the choice of a distributed architecture. Our second main architectural choice was to focus the animation more on data structures rather than on algorithms only. Roughly speaking, we wanted to allow a deep interaction between the user and the animated data structures used by an algorithm, while the algorithm itself was running on them. This is opposed to other algorithmic animations which choose an input, visualize it, and then run the algorithm on it, without giving much information on its internal data structures. We believe that this focus on animating data structures has many benefits. First of all, this often allows one to achieve a better and deeper understanding of the internal details of the al-

gorithm, which is especially important at debug time. Furthermore, it ensures a higher degree of interactivity, by exploiting the fact that data structures have methods for input that can be animated as well. Finally, it also improves the reusability, as complex animated algorithms and data structures can be often obtained as a composition of simpler animated data structures.

We believe that there are three main natural outcomes from this approach, as shown in Figure 1. First of all, CATAI is designed so as to provide naturally a portable Graphical User Interface, and this interface is given simply by a `Java` program. The system can of course be easily used in the Web: `Java` applets within animation clients can be accessed by using any standard `Java`-enabled WWW browser. The second feature is that CATAI allows cooperation among multiple animation clients towards the same algorithm, and even allows one to animate distributed algorithms. Finally, we believe that the focus of CATAI on reusability assures a great degree of efficiency with respect to both development time and the usage of network resources.

We are now ready to present in more details the architecture of CATAI. As illustrated in Figure 2, there are three main components in our system:

(1) The *algorithm servers* (*AlgServer*). These are executable programs written in `C++` and obtained from the original non-animated source codes of the algorithms with the support of some communication primitives from the CATAI library.

(2) The *animation clients* (*Ci*). They implement the users' interfaces toward the algorithms, and are written in `Java`.

(3) The *animation server* (*S*). It is the middle layer between animation clients and the algorithm server, and coordinates all the interactions between them. One of its main goals is to provide coherence between the non-animated data structures used by the algorithm server and the corresponding graphical objects handled by a client *Ci*. It is also written in `Java`.

The Algorithm Server

An algorithm server *AlgServer* is an executable program written in `C++` with the support of some communication primitives from the CATAI library. These primitives are grouped in two virtual classes: `Animator` and `Bootstrap`. The `Animator` class provides some communication primitives to the animated objects defined in *AlgServer*, while the `Bootstrap` class mainly provides the initialization routines necessary to the communication primitives. In order to be able to communicate with the animation server *S*, the original algorithm must be encapsulated in a class that has an interface known to *S*.

We describe now how to build an algorithm server *AlgServer* starting from an original non-animated algorithm *A*. This process is standard, so that it could be easily made automatic. First of all, the data structures used by *A* that we want to display are extended to animated data structures. We denote by *A'* the resulting algorithm. Next, *A'* is encapsulated in a class called `Animated_A'`, which is derived from the `Animator` and `Bootstrap` virtual classes. The class `Animated_A'` is responsible for:

(a) Defining the top level graphical objects which will be handled by *A'*;

(b) Defining the method `Animated_A'::A'()` which is used to call algorithm *A'*;

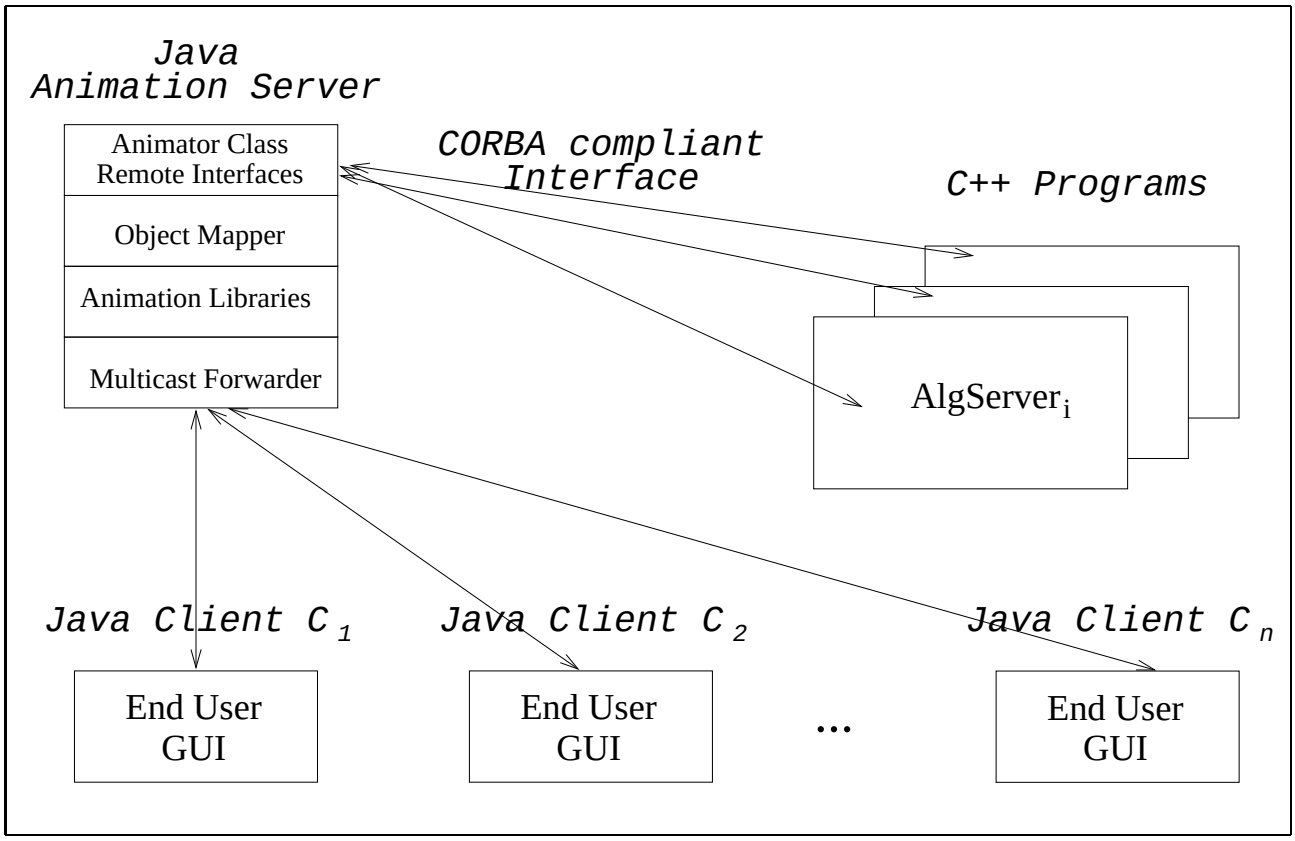

Figure 2: CATAI Basic Architecture.

(c) Defining any extra methods that handle the graphical objects bound to the user interface. All these methods will be offered by the server S to the clients Ci.

The animated classes (i.e., animated data structures) are obtained using multiple inheritance from the original non-animated class and the virtual class `Animator`. They are are defined on the base of the services offered by the animation server S, which are included in the *animation libraries* of S. This implies that an animated class contains *no graphical object representation*, a task which is left entirely to the animation server S. Namely, an object resides on an algorithm server *AlgServer* while its graphical representation resides on the animation server S: the binding between these two entities is established only at run-time, by late binding.

This is perhaps one of the advantages of this architecture with respect to reusability. Different data structures can use the same object representation: for instance, any algorithm using nodes can invoke the same node representation offered by the animation server S. This will speed up the process of generating new animated classes, and will allow the user to ignore completely the details behind the graphical representation of the animated objects. Note that the animated classes will maintain their original interfaces and thus will not cause any modification to the original algorithm.

We are now ready to explain how an algorithm server *AlgServer* works. When one runs *AlgServer*, there is a *bootstrap* phase where *AlgServer* announces its services to the animation server S. As any distributed system, *AlgServer* needs to agree with S on common object interfaces: after an initial simple handshake, the animation server S learns what is the local support required by the animated data structures in *AlgServer* and what are the interfaces exposed by *AlgServer*. The algorithm server completes the bootstrap phase by entering into an event

handler loop, where *AlgServer* waits for connection requests from the animation server S (i.e., to be selected by a user wishing to run algorithm *A'*).

When one or more users select *AlgServer*, the animation server S remotely instantiates an object of the class `Animated_A'`. The constructor of this class will enter in a second event handler loop that will provide the basic GUI to the relevant clients through the algorithm server S. In this GUI the user can have many options, including running the algorithm. If this option is selected, the animation server S will invoke the execution of algorithm *A'* through the remote invocation of `Animated_A'::A'()`.

The Animation Server

All the coordination between the algorithm servers on one side and the animation clients on the other side is carried out by the animation server S: indeed, one of the primary goals of an animation server is to interface algorithm servers with their graphical representations on the animation clients. As illustrated in Figure 2, an animation server contains four basic modules:

The Animator Class Remote Interfaces. This layer defines the communication primitives and implements the `Java` / `C++` interoperability as a collection of *stubs/proxies*. Proxies encapsulate network calls and access distributed objects: a proxy looks like the server object in term of the methods it implements. However, its implementation of the object methods would carry out the remote calls to forward them to the remote object. The mechanism for making remote function calls is carried out via sections of the code called *stub code*. This layer will interface the remote objects instantiated by an algorithm server by parsing the events received from the GUI of an algorithm client and by mapping them onto events for objects on an algorithm server, and vice versa.

Object Mapper. This module is in charge of maintaining a mapping between graphical objects and their counterparts on the corresponding algorithm server. Each time a user interacts with a visual object, an event will be sent to the corresponding object on a particular algorithm server. On the other hand, each time an algorithm server modifies the state of an object, a corresponding change will be produced on the user display.

Animation Libraries. This layer contains a repository of `Java` classes able to draw graphical objects and to define the standard interactions with them. For instance, in this module it is defined how to draw a binary tree and how to deal with it (moving, deleting, inserting nodes, etc etc). When an algorithm server *AlgServer* is started, it requests all the entries of the Animation Libraries that are necessary for its own animated data structures. If available, the corresponding classes are downloaded to the clients that will request connection to *AlgServer*.

Multicast forwarder. This module provides one-to-many communication and let many clients interact with the running instance of the algorithm. As an animation client might join the running instance of an algorithm even after the algorithm is started, the multicast forwarder contains a coordination protocol that aligns all the clients with the algorithm execution. The same coordination protocol maintains the synchronization primitives that allow each client to interact with the algorithm server in a mutually exclusive way. For example if we are executing an algorithm based on dynamic graphs, the user should be able to create or delete new nodes and edges. When these updates are requested from one client, all the other clients are locked until the operation is terminated.

The Animation Clients *Ci*

Animation clients are `Java` programs targeted to handling three different types of events. The first type are *user-generated* events towards an algorithm server *AlgServer*; these include mouse-generated commands such as "click", "drag-and-drop", and "SHIFT+click". The second type of events are *animation directives*: these are events originated by an algorithm server *AlgServer* and directed towards the user interface of an algorithm client which require some changes in the graphical presentation; examples can be operations like create a new graphical object, establish a connection between two different graphical objects, etc. The third type of events are the *alignment directives* used by the animation server S to coordinate multiple clients that can be dynamically entering the pool of displaying devices for a particular algorithm. Since all the communication between animation clients and algorithm servers is handled by the animation server S, all these events (including user-generated events and animation directives) are between animation clients and S.

We now describe how animation clients interact with the animation server. When a client program *Ci* starts, it looks for an animation server at a specified address. Next, *Ci* receives from the animation server S a list of available algorithms, namely all the algorithms running on some algorithm server. If the user chooses the algorithm offered by a particular algorithm server *AlgServer*, then a new virtual connection is established via the animation server between the client *Ci* and the corresponding *AlgServer*. The animation server S maintains information about the pool of clients currently connected to that algorithm and coordinates all the interactions and the updates on the display for all the members of the pool. At this point, the client *Ci* can download all the data it needs to display the data structures animated by *AlgServer*. From now on, each client will be able to call some services, such as some update operations on *AlgServer*, and to visualize the resulting evolution. On the other hand, when *AlgServer* allocates a new animated object, either at initialization time or during the execution of the algorithm, this new object will be displayed on the clients as instructed by the animation server S.

The basic GUI presented by the clients follows the interface proposed by *AlgServer*, which is defined in term of abstract objects contained in the animation libraries resident in the animation server. The server in turn will use the basic graphic primitives present in the client code built on the top of `Java` AWT (the Abstract Window Toolkit of the `Java` Virtual Machine). Each client keep its own map of all the animated objects in order to be able to address them on *AlgServer* and the GUI will directly call their methods as a remote method invocation. Again, this map will be updated when a new object is created by *AlgServer*.

We conclude this section with some more comments about the level of reusability offered by CATAI. First of all, we should mention that an object-oriented paradigm by itself offers reusability: an animated class, once it has been designed, can naturally be recycled and extended. For instance, we can inherit an animated list class to define easily an animated stack class. The reusability features of CATAI are of course deeper than that. Indeed, one can also define animated classes by reusing all the graphical representations defined in the animation libraries. For instance, the graphical representation of a node is the same for graphs and trees, and different implementations of graph classes could share much of the same graphical representation. Or we can use animated arrays and animated lists to obtain, at no extra cost, a basic animation of many sorting algorithms. Finally, and perhaps more importantly, with our approach, we can easily exchange and reuse animated data structures that are built-in somewhere in algorithm servers. This was particularly useful in our own experience.

3 An Example of Animation with CATAI

We now describe in some details how to prepare an algorithm animation with CATAI. For sake of clarity, we will describe the general steps that must be followed for accomplishing this task and at the same time illustrate them through a working example: the animation of Kruskal's algorithm for computing a minimum spanning tree (MST) of a graph [20].

Kruskal's algorithm first sorts all the edges by increasing cost, and then grows a forest that will finally converge to a minimum spanning tree. Initially, the forest consists of n singleton nodes (i.e., the vertices in the graph). The algorithm scans the edges, one at the time and in increasing order of their cost. If the endpoints of the edge under examination are already connected in the current forest, this edge is discarded. Otherwise, the endpoints are in two different trees: the edge is inserted into the forest (i.e., it will be a minimum spanning tree edge), and the two trees are merged into one. For efficiency issues, the trees are maintained as set union data structures [31]. We refer to LEDA's implementation of Kruskal's algorithm [22], which make use of the class `partition` to implement set union data structures.

3.1 How to animate an algorithm

While building an algorithm animation, the first decision to be taken is which data structures are to be animated. In the example at hand, for instance, it seems natural to visualize the graph being explored; additionally, we could also choose to animate the underlying partition given by the set union data structures. Once this has been decided, the process of developing an animation can be broken in three different steps.

Creating an animation library. A crucial module which provides the basic animation in CATAI is given by the animation libraries, which intuitively provide an animated interpretation of "how data structures work". These libraries are totally independent from the data structures being animated and can be easily reused. If the animation server does not support already animation libraries for the algorithm at hand, then we must develop the animation libraries required by the algorithm. In our example of minimum spanning trees, CATAI contains already animation libraries to represent graph objects, and thus this task is empty. We remark here that CATAI supplies animation libraries for most textbook algorithms, and thus in the average one would rarely need to build algorithmic libraries from scratch.

Animation libraries manages animated objects (`anim objects`) which are the building blocks of animations. Each `anim object` can have a customizable appearance and can be related to other `anim objects` via animation links (`anim links`): in particular, `anim links` define a set of `anim objects` with a common property. To create a new animation library, we must first create a new `Java` Object derived from a CATAI class: the `lib struct` class. Each animation library must implement two special methods (`paint_anim_object` and `paint_anim_links`) whose purpose is to define how an `anim object` and an `anim link` is graphically represented. The `lib struct` class provides the user with a basic set of animation primitives to add, remove, select, label and access both an `anim object` and an `anim link`.

Creating animated data structures. Once animation libraries are available, we need to revise the implementation of the original data structures to support some animation capabilities. We call *animated classes* the classes that implement data structures with support for animation: CATAI offers a specialized `C++` library to assist in the development of animated classes. The

principal component of this library is the `Animator` class, which provides animation server communication primitives and binding mechanisms between a data structure and the related animation library. An animated class can be derived from the original non-animated class and from the `Animator` class: for each method of the original class that we want to animate, we define a new method with the same prototype which holds the original method invocation together with the use of one or more animation primitives. These primitives maps data structure operators to their animated counterparts. The interface of these functions is common for all animation libraries, while their behavior depends on the animation library selected. To achieve better animations, we can extend data structures by introducing some animation-specific operators, i.e., operators whose functionality does not affect a data structure but only its visual representation.

```
class animgraph : public graph , public Animator
{
public:

animgraph(int tsockd): Animator(tsockd, GRAPH), graph(){}
// build animgraph as a graph plus Animator
// tsockd is the communication channel reference obtained in the animation boot
strap
// GRAPH is the animation library id defined in Animator

node animgraph::new_node()
{
  node n = graph::new_node();
       // create a new node
  Animator::new_obj((long) n, index(n));
       // next create a new anim object and bind it to the original node
  return n;
}

edge animgraph::new_edge(node src, node dst)
{
  edge e = graph::new_edge(src, dst);
       // create a new edge
  new_link((long) src, (long) dst, index(temp), (long) e);
       // next create a new anim link and bind it to the original edge
  return(temp);
}

void animgraph::color_node(node v, int c)
{
 Animator::color_obj((long) v, c);
}
};
```

Figure 3: The definition of `animgraph` and some of its methods.

In our running example of minimum spanning tree, the non-animated algorithm uses the LEDA graph and partition data types. The LEDA graph class uses a single object which acts as a container to hold nodes and edges. To obtain the animated class, we derive the class

`animgraph` from the LEDA `graph` class and from the `Animator` class. The methods that we wish to animate are those which change the graph: adding, removing and modifying edges or vertices. Apart from these methods, we could also add some extra methods for animation purposes. The definition of the animgraph class and the implementation of some animated methods is given in Figure 3. The animated partition class can be defined with the same technique.

Animated algorithm. Once both animation libraries and animated data structures are available, we can finally code the *animated algorithm*. The resulting program will comply to CATAI protocols and will exhibit a standard animation. This animation could be further improved using animation-specific operators and CATAI director tools. These tools allow one to interact in more depth with algorithm and to coordinate its execution. CATAI defines all the interactions as a standard set of events: each of these events can be easily bound to a method invocation. Apart from these interactions, CATAI provides also tools for requiring acknowledgments to end-users, for showing customizable text boxes and for synchronizing animations playing.

Original algorithm

```
....
G = new graph();
....
list<edge> MST::KRUSKAL(graph &G)
{
  node_partition P(G);
  list<edge> L = G.all_edges();
  list<edge> T;

  L.sort(CMP_EDGES);
  edge e;
  forall(e,L) {
    node v = source(e);
    node w = target(e);
    if (! P->same_block(v,w)) {
      T.append(e);
      P->union_blocks(v,w);
    }
  }
  return T;
}
```

Animated algorithm

```
....
G = new animgraph(sockd);
....
list<edge> MST::KRUSKAL(animgraph &G)
{
  anim_node_partition P(G);
  list<edge> L = G.all_edges();
  list<edge> T;

  L.sort(CMP_EDGES);
  edge e;
  forall(e,L) {
    color_edge(e, GREEN);
    node v = source(e);
    node w = target(e);
    if (! P->same_block(v,w)) {
      T.append(e);
      color_edge(e, BLUE);
      color_node(v, BLUE);
      color_node(w, BLUE);
      P->union_blocks(v,w);
    }
    else {
      color_edge(e, RED);
    }
  }
  return T;
}
```

Figure 4: Kruskal's algorithm for MST in its original (LEDA-like) version and its animation in CATAI.

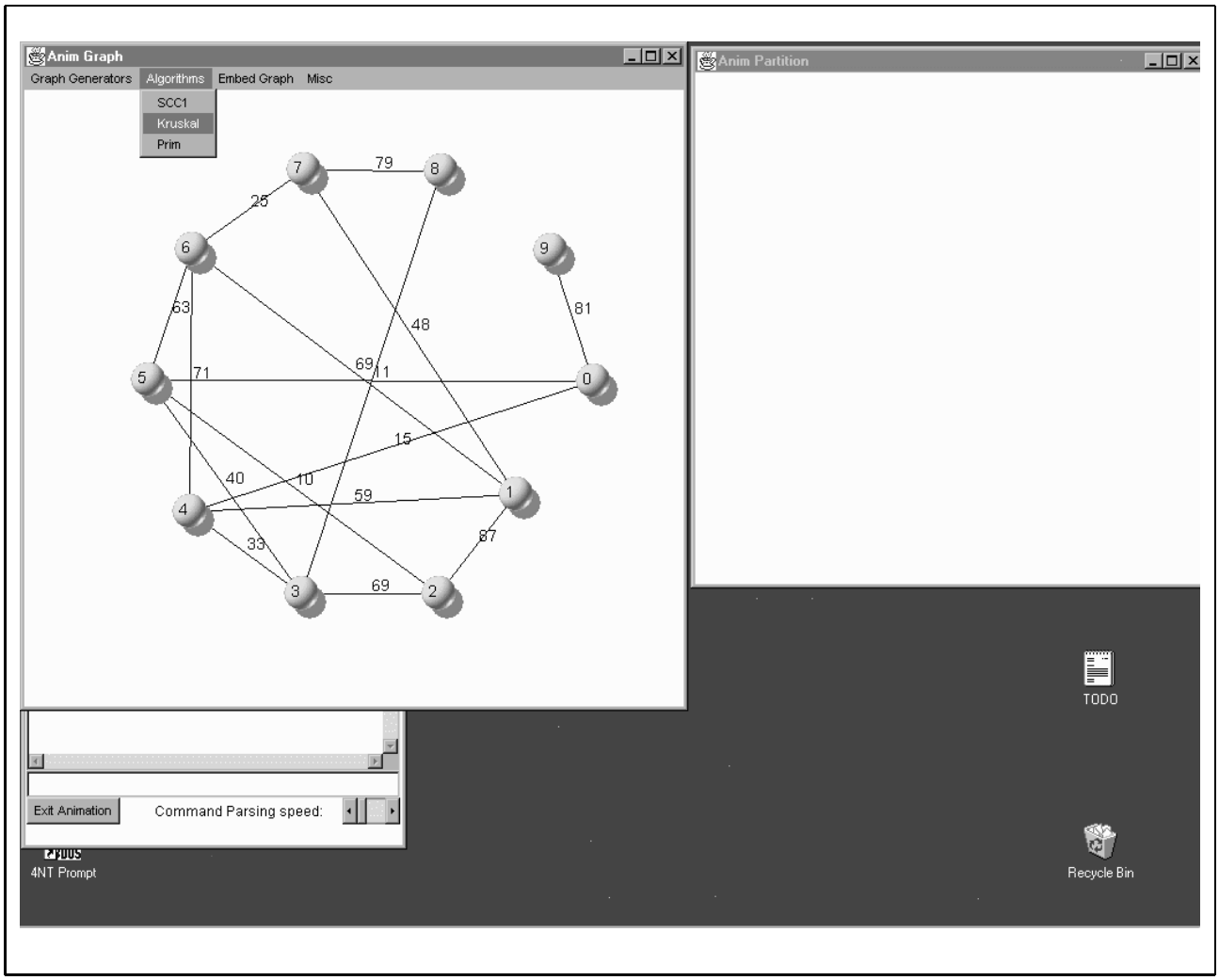

Figure 5: The animation starts on a graph. Three algorithms are offered by the algorithm servers: `SCC1`, `Kruskal` and `Prim`: the user selects `Kruskal`.All vertices in the graph are originally colored yellow, and all edges are colored black.

We are now ready to show how to animate the implementation of Kruskal's algorithm at hand. Starting from the original code, we replace the standard graph and partition with their animated counterparts. Next, we add some animation-specific code to highlight the behavior of the algorithm. For instance, we can choose to color *red* the edge which we are currently considering. If this edge will be included in the minimum spanning tree, then we will color it *blue*, and otherwise we will color it *red*. Endpoints of *blue* edges are colored *blue*, so that a forest of *blue* trees is visualized throughout the execution of the algorithm. This *blue* forest will converge to a minimum spanning tree. The resulting algorithm is proposed as a method of a container object, i.e., an MST class, and the public interface of this object will report the services (methods) that can be requested by the end-user. A comparison between the code of the original non-animated algorithm and the code of animated algorithm can be obtained from Figure 4. Some snapshots of the actual animation are contained in Figures 5–7.

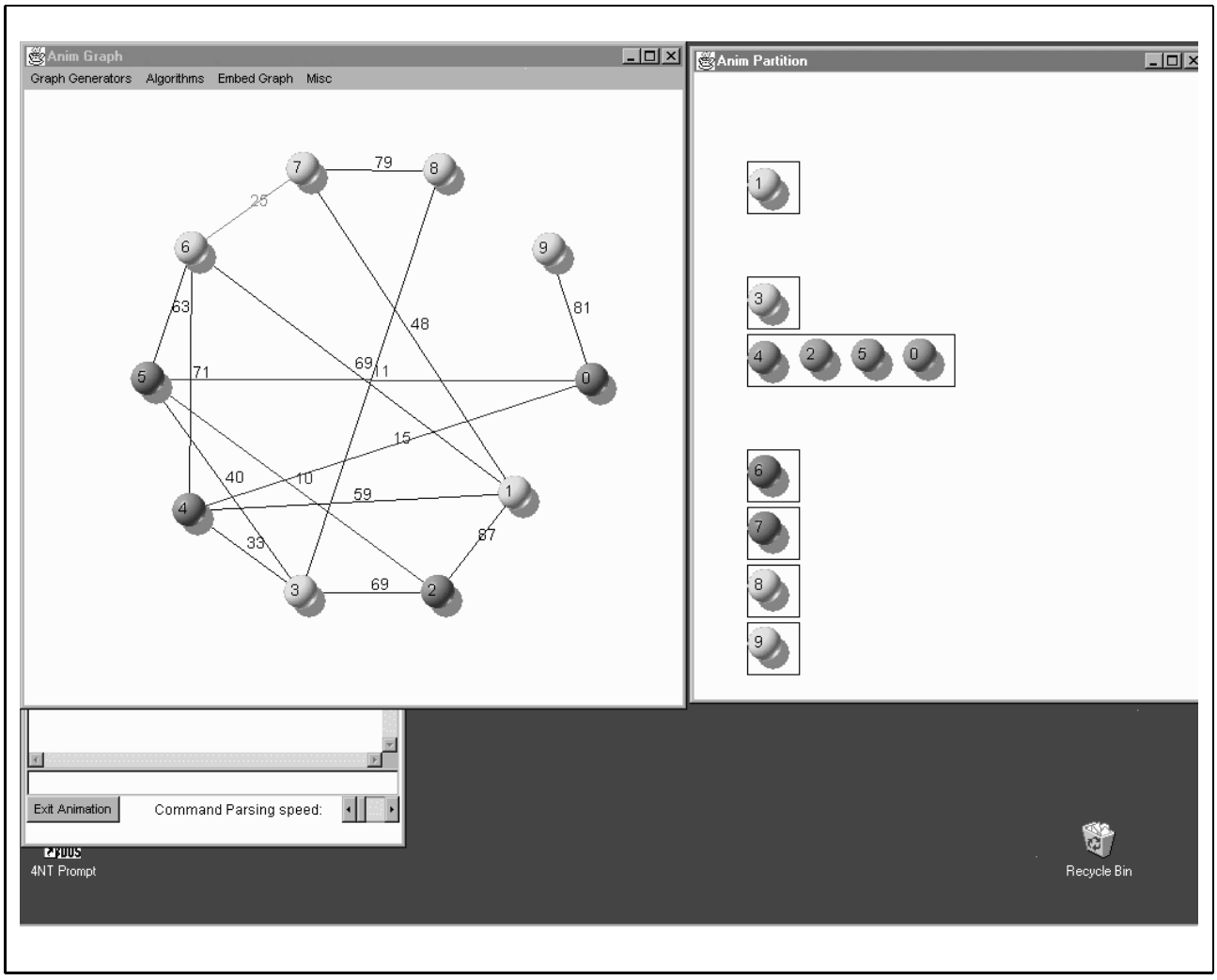

Figure 6: The animation at the fourth step: edges *(2,5)*, *(0,5)* and *(4,0)* have been examined and colored blue together with their endpoints, and the edge *(6,7)* is currently examined and colored green. The state of the partition is shown to the right: we have grown only one blue tree (containing vertices *0,2,4,5*). All the other vertices are still in singleton trees.

4 The Main Features of CATAI

In this section, we summarize the basic characteristics of CATAI, and try to put its contributions in perspective.

First of all, we designed CATAI so that it offers an advanced support for *interaction*. In a certain way, much of the control is transferred by the algorithm to the user, as at run time the user is allowed a deep coordination of the algorithm animation through the visual manipulation and navigation of data structures. Consequently, the animation can be used truly as a basic GUI for accessing the algorithm properties and functionalities. We found this very useful especially when using algorithm animation in program development and debugging: we could put some data structures in non-standard or wrong configurations, and visualize the effects on the running algorithm. Other systems that we are aware of mainly implement very simple forms of interactions, which do not involve visual representations and only allow the user to execute a fixed set of functions for each algorithm; very often, the arguments of these functions cannot be specified by interacting with the animation.

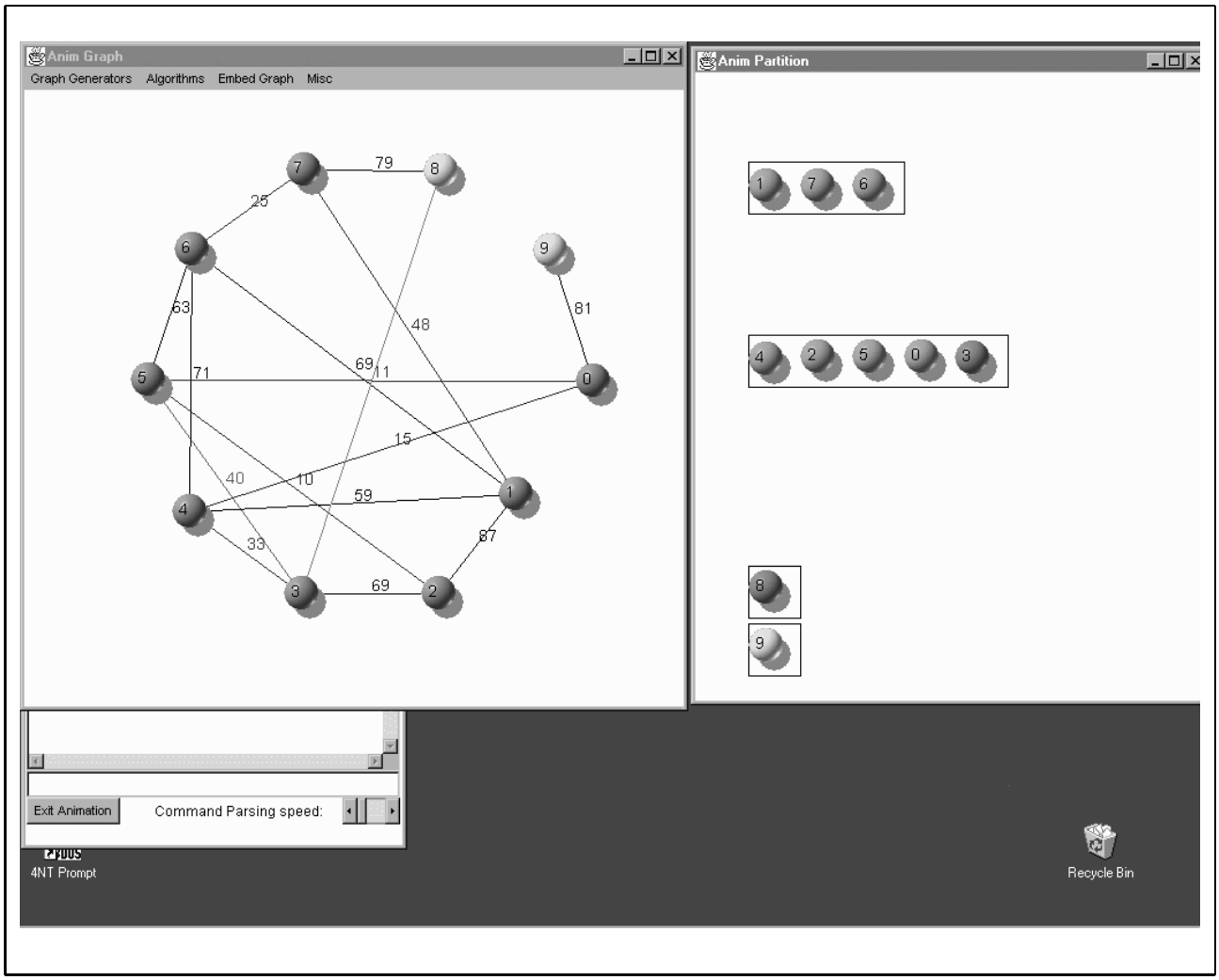

Figure 7: The animation at the eighth step: edges *(2,5)* *(0,5)*, *(4,0)*, *(6,7)*,*(4,3)* and *(1,7)* have been examined and colored blue together with their endpoints, edge *(5,3)* has been colored red, and the edge *(3,8)* is currently examined and colored green. The state of the partition is shown to the right: we have grown two blue trees (one containing vertices *0,2,3,4,5* and the other containing vertices *1,6,7*). Vertices *8* and *9* are still in singleton trees.

Furthermore, CATAI offers a high degree of *adherence* between the algorithm and its animation. Indeed, the system guarantees a close match between the behavior of the algorithm execution and its animated counterpart. Other systems use a classical event-driven approach: animation directives rely on specific function calls spread along the algorithmic source code. This tecnique does not always assure adherence as the only relation between the execution of a function and its animated visualization is the *concomitant* invocation of the right animation directive. This is a condition which is left entirely to the programmer, and thus cannot be always automatically assured. On the contrary, CATAI presents an evolution of the event-driven approach, as it encapsulates each animated operator together with the animation link in a container method. In this way, the animation follows closely the algorithm: each animated operator invocation will be followed by the execution of its animated counterpart and thus there is the guarantee that each "step" in the algorithm induces "step" in the animation. Conversely, the algorithm follows closely the animation: indeed, the animation is directly bound to the

methods of the algorithm, so that each user interaction is forwarded by the animation to the algorithm, where it takes the shape of method invocation.

The next feature we mention is *scalability*, i.e., how a system deals the animation of a large number of possibly very complex data structures, and we believe that CATAI scales up naturally with respect both to the number and to the complexity of data structures. Indeed in CATAI, graphical objects and animation references are internally kept in the animation server S, so that users do not need to deal directly with them. Consequently, animating a large number of data structures requires no more effort to the user than animating a small number of them. The only extra effort required when animating a large number of data structures is for highly interactive animations, as in this case the user must specify which are the interactions and the data structures involved with these interactions. This is much more complicated for other systems, which require, apart from writing the animation code, some additional efforts to track all the allocated graphical objects and their correspondence with data structures. The effort in these systems clearly grows with the number of data structures involved. As for scalability with respect to the complexity of data structures, other systems do not seem to provide a specific solution to this problem: it seems to us that the user needs to write complex animation code in order to animate complex data structures. In CATAI, most of the times the animation of complex data structures can be built by first animating simpler data structures and then by assembling them in an object-oriented fashioned manner.

In our personal experience, CATAI was an effective tool for reducing the overall complexity of setting up an animation. Indeed, it offers a good trade-off between the level of reusability, the average time it takes to a non-sophisticated programmer to develop an animation, and the basic knowledge required to use the system. We emphasize here that we consider the case where a *user* of the system (not the *provider*!) wishes to set up an animation. This is particularly important, as in many cases tasks that are trivial for the knowledgeable and expert provider of a system might be very difficult to achieve for the typical user. If one wishes to animate an algorithm with CATAI, there are three possible scenarios:

(A) No support is available (no animation library and no animated data structures).

(B) Only the animation library is available;

(C) The animation library and an animated implementation of the underlying data structures are already available;

In case (A) when no support for the data structures is available, the user has to create animation libraries from scratch: this implies a medium/high development time (perhaps comparable to the development time required by other systems), and a basic knowledge of `C++`, `Java` and Corba. In case (B), which we believe will be the average case for a CATAI user, the main task is to derive the animated data structures from the non-animated code. As it can be inferred from Figures 3 and 4, in this case the development time is typically low/medium, and only a moderate knowledge of `C++` is required. Finally, in case (C) the animation with CATAI is straightforward: only a simple knowledge of `C++` is required to compose the existing pieces, and the time required to develop the animation is extremely low (if any), as it can be seen from Figure 4.

Similarly to Mocha [2], CATAI is a truly *distributed* architecture: all its components can be placed anywhere provided that they share a communication channel. Perhaps even more than Mocha, CATAI heavily relies on distributed objects, which have been developed using a CORBA

compliant protocol. In addition, the HTTP protocol is used for downloading the `Java` classes needed and a BSD socket message interface is used for the communication addressed by an algorithm server towards the animation server. The overall communication load is typically low since all the animated scenes reside on animation clients and their activation can be obtained at a low cost using short messages: basically only invocation of methods on remote objects. This gives an effective support to the animation of distributed algorithms, which can be naturally implemented with multiple algorithm servers.

Finally, as many other systems CATAI offers a good degree of code protection, as an animation can be exported remotely while the animated program is maintained on a safe host, and overall security, as communication protocols export limited resources and only to the pool of animation components.

We conclude this section by mentioning another feature of CATAI, which relates to its ease of integration in the World Wide Web (WWW) environment. Although originally developed to make information publicly available, the World Wide Web has already shown its large potential in many areas, including educational support especially for distance learning. In fact, many educational systems based on WWW have been recently developed and the educational use of WWW has attracted a lot of attention (see e.g., [16, 19]). The potential of WWW as an educational tool have also been studied in the area of *Computer Supported Cooperative Workgroup* (CSCW) [1, 13, 14, 15, 17, 23, 24] which in general is concerned with the new possibilities offered by the current network technology to cooperative workgroup. In both areas, there is a great deal of attention on the usage of WWW for a fully integrated cooperative environment between students and instructors. In fact, the aspect of the "ordinary" classroom that lies in the interactions between the students and the instructor and among students themselves has stimulated a lot of work on several systems for "Shared Workgroup" on WWW [4, 5, 11, 18, 21, 26, 32]. The architecture of CATAI is naturally integrated in the Web: in fact WWW servers (HTTP server) and WWW clients (browsers) can be mapped, respectively, to CATAI animation servers and animation clients, and an animation client can be even used as an applet within an HTML page (while the animation server can be managed by a WWW server).

The integration of CATAI with the Web can be extremely useful in the design of integrated educational systems, where a student can interact with algorithmic courseware: he or she can learn a particular set of algorithms, see some of their implementations, and interact with the visualization of the algorithms cooperatively with other students or with the instructor. Moreover, CATAI offers the possibility of visualizing algorithms that can be related to data repositories stored somewhere else, such in the case of real-time monitoring processes: indeed CATAI can allow an easy, efficient, distributed and interactive access to such processes. For instance, we can easily add methods to animated data structures and algorithms that allow following "links" (URL's) to the rest of the WWW. As an example, we are currently investigating a particular application which is connected to monitoring network congestions via the Web: we have a network, and we wish to find the shortest path (or some other particular paths) to a set of destinations. In some cases, shortest paths are not deeply informative, and one would like to obtain a truly real-time interaction with the network, i.e., by clicking on a link to get information on the current traffic pattern on that link (e.g., road conditions, link congestions, etc). A prototype of this kind seems to require not much effort with CATAI.

5 Conclusions

We have presented a new system for algorithm animation, whose main features include a typical low effort required for setting up a new animation, and the possibility of embedding animation clients in standard `Java`-enabled Web browsers. We believe this to be a good compromise between two different viewpoints: the programmer's perspective, which typically includes the goal of animating quickly and efficiently a given algorithmic code, and the user's perspective, which clearly benefits from highly interactive, easy-to-use, distributed and cooperative interfaces. We believe that taking into account both these two views is important, as the programmer and the user of an animation system can often be united in the same person: e.g., a programmer wishing to use algorithm animation to visualize and debug his/her own algorithmic code, or a student or an instructor wishing to set up and interact with his/her animation of an algorithm.

References

[1] W. Appelt, U. Busbach. "The BSCW System: A WWW based Application to Support Cooperation of Distributed Groups". GMD-German National Research Centre for Information Technology. Sankt Augustin, Germany. 1996.

[2] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. "The Mocha algorithm animation system". In *Proc. Int. Workshop on Advanced Visual Interfaces*, pp. 248-250, 1996.

[3] 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*, pp. 203-212, 1996.

[4] R. Bentley, T. Horstmann. "*Supporting collaborative information sharing with the World Wide Web: the BSCW Shared Workspace system*". Workshop on World Wide Web and Collaboration, Massachusetts Institute of Technology, Cambridge (USA) 1995.

[5] R. Bentley, U. Busbach, K. Sikkel. "The Architecture of the BSCW Shared Workspace System". In: Proceedings of the ERCIM workshop on CSCW and the Web. Sankt Augustin, Germany, February 1996.

[6] Brown, M. H., Sedgewick, R. . "*A System for Algorithm Animation*". In *Proceedings of ACM SIGGRAPH '84*, (pp. 177-186). New York: ACM. 1984

[7] Brown, M. H., Sedgewick, R. "*Techniques for Algorithm Animation*". IEEE Software, 2(1): 28-39, 1985.

[8] Brown, M. H. "*Algorithm Animation*". New York: MIT Press.

[9] Brown, M. H. "*Zeus: A System for Algorithm Animation and Multi-View Editing*". In *Proceedings of IEEE Workshop on Visual Languages*, (pp. 4-9). New York: IEEE Computer Society Press. See also: Brown, M. H. "*Zeus: A System for Algorithm Animation and Multi-view Editing*". (Research Report No. 75). DEC Systems Research Center, Palo Alto, CA.

[10] M. H.Brown, J. Hershberger. "Color and Sound in Algorithm Animation". *Computer*, n.25. pp.52-63. 1991.

[11] D.M. Chiu, D. Griffin. "*Workgroup Web Forum: Tools and applications for WWW-based group collaboration*". Workshop on World Wide Web and Collaboration, Massachusetts Institute of Technology, Cambridge (USA) 1995.

[12] A. van Dam. "The Electronic Classrom: Workstations for teaching". *Intnl. Journal of Man-Machine Studies"*.. 21 (4), pp. 353-363, 1984.

[13] D. Decouchant, M.R Salcedo. "Alliance: A Structured Cooperative Editor on the Web". In: Proceedings of the ERCIM workshop on CSCW and the Web. Sankt Augustin, Germany, February 1996.

[14] P. De Bra, A. Aerts. "Multi-User Publishing in the Web: DRess, A Document Repository Service Station". In: Proceedings of the ERCIM workshop on CSCW and the Web. Sankt Augustin, Germany, February 1996.

[15] A. Dix. "Challenges and Perspectives for Cooperative Work on the Web". In: Proceedings of the ERCIM workshop on CSCW and the Web. Sankt Augustin, Germany, February 1996.

[16] D. Dwyer, K. Barbieri, H.M. Doerr. "*Creating a Virtual Classroom for Interactive Education on the Web*". Proc. of WWW 95, Third International Conference on World Wide Web.

[17] J.I. Griera. "Managing CSCW". In: Proceedings of the ERCIM workshop on CSCW and Web. Sankt Augustin, Germany, February 1996.

[18] T. Gruber. "*Collaborating around Shared Content on the WWW*". Workshop on World Wide Web and Collaboration, Massachusetts Institute of Technology, Cambridge (USA) 1995.

[19] B. Ibrahim, S.D. Franklin. "*Advanced Educational Uses of the World Wide Web*". Proc. of WWW95, 3rd International Conference on World Wide Web.

[20] J. B. Kruskal. "*On the shortest spanning subtree of a graph and the traveling salesman problem*". Proc. Amer. Math. Soc. 7 (1956), pp. 48–50.

[21] K.MacArthur. "*Collaboration, Knowledge representation and Automatability*". [URL: `http://www.w3.org/pub/WWW/Collaboration`]

[22] K. Mehlhorn and S. Näher: "LEDA: A Platform for Combinatorial and Geometric Computing", *Communications of the ACM*, 38(1), 96–102, 1995.

[23] D.R.Newman, "How can WWW-based groupware better support critical thinking in CSCL". In: Proceedings of the ERCIM workshop on CSCW and the Web. Sankt Augustin, Germany, February 1996.

[24] R.Peters, C.Neuss. "CrystalWeb – A Distributed Authoring Environment for the World Wide Web". Proc. of WWW95, 3rd International Conference on World Wide Web.

[25] Price, B.A., Baecker, R.M., and Small, I.S. "A Principled Taxonomy of Sofware Visualization". Journal of Visual Languages and Computing 4(3):211-266.

[26] M. Roscheisen, T. Winograd. "*Generalized Annotations for Shared Commenting, Content Rating, and Other Collaborative Usage*". Workshop on World Wide Web and Collaboration, Massachusetts Institute of Technology, Cambridge (USA) 1995.

[27] J.T. Stasko. "*Tango: A Framework and System for Algorithm Animation*". IEEE Computer, 23(9), pp. 27-39, 1990.

[28] J.T.Stasko, E.Kraemer. "A Methodology for Building Application-Specific Visualizations of Parallel Programs". Tech. Rep. GIT-GVU-92-10. (`ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/92-10.ps.Z`)

[29] J.T.Stasko. "Supporting Student-Built Algorithm Animation as a Pedagogical Tool". In *Proc. of of the ACM SIGCHI '97 Conference on Human Factors in Computing Systems*, Atlanta, GA, USA. 1997.

[30] *Software Visualization*. John T. Stasko, John B. Domingue, Marc H. Brown, and Blaine A. Price (Eds.). MIT Press, 1998.

[31] R. E. Tarjan, J. van Leeuwen. "*Worst-case analysis of set union algorithms*. J. Assoc. Mach. 31 (1984), pp. 245–281.

[32] S. Virdhagriswaran, M. Webb, J. Mallatt. "*Shared Information Space: An Interactive, Collaborative System Enablement Perspective*". Workshop on World Wide Web and Collaboration, Massachusetts Institute of Technology, Cambridge (USA) 1995.