CATAI: Concurrent Algorithms and data Types Animation over the Internet *

Giuseppe Cattaneo†
Dipartimento di Informatica e Applicazioni
Università di Salerno, Salerno, Italy

Umberto Ferraro-Petrillo‡
Dipartimento di Informatica e Applicazioni
Università di Salerno, Salerno, Italy

Giuseppe F. Italiano§
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Roma, Italy

March 6, 2001

Abstract

In this paper we present CATAI *(for Concurrent Algorithms and data Types Animation over the Internet), an algorithm animation distributed system.* CATAI *adopts an Object-Oriented animation approach that allows a programmer with an algorithmic background to transparently and efficiently animate a target algorithm. Multiple users can simultaneously watch and interact with a same animation made with* CATAI *using a light-weighted Java animation client. 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.*

*This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering" and by CNR, the Italian National Research Council under contract n. 00.00346.CT26

†Email: `cattaneo@dia.unisa.it`.

‡Email: `umbfer@dia.unisa.it`.

§Email: `italiano@info.uniroma2.it`. URL: `http://www.info.uniroma2.it/~italiano`.

1 Introduction

Algorithm animation is a form of software visualization that uses interactive graphics to enhance the presentation, development and understanding of algorithms. Systems for algorithm animation have matured significantly in the last decade [3, 4, 8, 12, 14, 32, 34, 35], 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 is supposed to 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. Indeed our own experience supported the same conclusions drawn in reference [28], namely that the effort required to animate an algorithm is one of the main factors limiting the diffusion of animation techniques as a successful tool for debugging, testing and understanding computer algorithms.

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. The main philosophy behind our system is that any algorithm implemented in an object-oriented programming language (such as `C++`) should be easily animated with CATAI. This should make this system easy to use, and is based on the idea 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 with previous systems, 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 types* 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.

1.1 Related Work on Algorithm Animation

Historically, the birth of algorithm animation is set to the early '80s with the video *Sorting Out Sorting* by Baecker [1], which shows the behavior of a set of popular sorting algorithms. The constant evolution of the computer processing power together with the introduction of less expensive graphical displays has promoted an increasing interest in algorithm animation. This led to the definition and the implementation of several algorithm animation systems. Brown et al. [8, 9, 10, 11, 12, 13] proposed the first system for the animation of general purpose algorithms, BALSA, followed few years later by BALSA II. These systems allow a programmer to define a set of *View* to be used to represent an input algorithm. Each *View* uses a graphical library to implement the representation of a particular point of view on the algorithm to be animated. The programmer has to annotate the original source code of the algorithm with *interesting events*. These are special-purpose functions used by the system to explicitly notify to the *Views* the evolution of the algorithm state with respect to the animation. Each *View* defines which representation must be performed for each event. When the system is run, each *View* reacts to the arrival of a new event by performing the associated representation (if present). Animated algorithms and algorithms' *Views* are run as different processes, multi users animation are supported by using network communication primitives to deliver *interesting events* to several users at once. The technique to describe the algorithm behavior using *interesting events* has been successfully adopted also by other systems. Among these, we cite Zeus [14], TANGO [31], XTANGO, Polka [32, 33] and ANIM [5, 6]. Zeus is an evolution of BALSA II that adds some object-oriented features like the possibility to create new *Views* by deriving a standard base *View* class. TANGO is an algorithm animation system developed by Stasko that introduces a path-transition paradigm for the creation of smooth animations. According to this paradigm, an animation can be seen as the composition of distinct, even concurrent, graphical transitions. XTANGO is an evolution of TANGO that uses the X-Window graphical framework. Polka extends both the functionalities of TANGO and XTANGO by introducing the support for the animation of concurrent programs. One of the key points of this system is the possibility for the programmer to assemble and to present the whole animation using, as a timing system, an explicit global clock counter. Finally, the ANIM system has been one of the first to support *post-mortem* visualization (the ability to represent the behavior of an algorithm already executed). This is done by annotating in a trace file the *interesting events* generated during the algorithm execution. After the end of the execution, a specialized visualization process is run using as input the trace file previously generated. An alternative to annotating *interesting events* is represented by the *declarative approach*. It works by associating to the states of a running algorithm a graphical representation. To do so, the algorithm is run inside the animation system where it will be constantly analyzed. Each changes into the algorithm's state is monitored and visualized using an associated graphical representation. Several systems like PROVIDE [26], Aladdin [22], PVS [21] and Animus [18] featured a declarative approach to software visualization. However the first system able to handle complex algorithms with a declarative approach has been Pavane [29, 30] by Roman. This system distinguishes three different actors into the process of building an animation. The first actor, the *programmer*, implements the algorithm without caring of the animation. The second actor, the *animator*, is in charge of defining a mapping from program states to a graphical representation, the third actor, the *viewer*, examines the results of the representation interacts with the algorithm being run. Other systems that adopted a declarative approach are TPM [19], UWPI [23] and Leonardo [17]. TPM was proposed in the '80 as a debugging tool for the *post-mortem* visualization of computer programs written in the `PROLOG` programming language. In order to be able to represent the inherent complexity of a `PROLOG` program while allowing the user to focus only on the local details of the executions, TPM featured two distinct views, a fine-grained view to represent the program's locality and a coarse-grained view to show the overall behavior of the algorithm. UWPI is a system developed at Washington State University by that is able to automatically represent the behavior of very simple data structures like arrays or stacks. Leonardo offers a `C` based programming environment with support for software visualization, one of its most interesting feature is the possibility to obtain smooth animation capabilities while using a declarative approach. Nowadays, algorithm animation systems are starting to offer some capabilities made possible with the most recent technologies: among these we cite Mocha [3, 4] and JAWAA [27]. Mocha has been one of the first system to adopt a distributed architecture with the separation between the running animated algorithm and the visualization clients implemented as portable `Java` clients. JAWAA works completely on the top of the Java Virtual Machine embedded in any `Java` compliant web browser. It offers a simple scripting language that can be used to write animations to be displayed in a web browser window.

The remainder of this paper is organized as follows: in section 2 we present the design principles of CATAI discussing which are the motivations that led us to propose CATAI together with the requirements that, in our opinion, are needed in an optimal algorithm animation system. In section 3 we present the architecture of CATAI in a bottom-up fashion, starting from a low-level object-oriented view we introduce the algorithm animation services implemented by CATAI. In section 4, we present the main features of CATAI discussing how they have been made possible by the underlying architecture. In section 5, we list the main advantages of CATAI. In section 6 we present a guided tour to the use of CATAI that features some example animations. Finally, in section 7 we present the conclusion of our work and we introduce some issues that will be addressed by our future work in the field of the algorithm animation.

2 The Design Principles of CATAI

When the CATAI [1] project was started, we had already implemented and animated a fair amount of algorithms. Thus, our own experience as programmers and designers of algorithms, strongly influenced our views and development of CATAI. In particular, while developing an animation for sophisticated algorithmic techniques (e.g., the sparsification technique of Eppstein *et al* [20]) we have experimented several problems concerning the difficulties of currently available systems to cope with very complex data structures. 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, tuning and teaching complex algorithms.

Our first main concern was *reusability*, as previous experiences with this were rather discomforting. In many cases, in the area of algorithm animation reusability was not considered at all, and very often the animation 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 type* (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

[1] CATAI project home page: http://isis.dia.unisa.it/Catai

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.

Another important issue, often underestimated in other animation systems, is to grant to the user the possibility to *interact* as much as possible with the animated algorithm. We wanted, for instance, to allow the user to interact with the algorithm not only while providing input data sets but even *during its execution*. One example could be changing *animated data structure*s 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 usual interactions consisted of the possibility 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 [4], 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 [32]). 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.

Finally, being part of 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 [3].

In our opinion, one of the principal weakness of many animation systems developed so far is their lack of abstraction. Let us consider, for example, the creation of a complex animation. This is often a time consuming task that involves the use of several skills including programming, algorithmic and computer graphics. The programmer needs to spend a lot of efforts and time into building a complex representation (the animation) starting from simpler building blocks provided by a graphic library. To be more precise, the programmer has to care about problems like defining an effective representation for a data structure, deciding which information must be considered relevant, assembling and coordinating the different graphical scenes, and so on. We believe that it would be extremely beneficial for the programmer to be able to reuse as much as possible of this work for future animations.

In order to achieve this result, we have, first of all, tried to describe the process of building an animation through the definition of a sequence of distinct activities which compose a roadmap from the algorithm to the completely animated version. Then, we have defined an animation model that fully complies with object-oriented paradigms [7] (OOP). We believe that the benefits of an OOP approach can be crucial in the design of a system that allows one to easily and effectively animate an algorithm. Among these benefits we cite the possibility to easily reuse a previously developed component and the possibility to extend an existing components adding it new features. The idea of using an object-oriented approach to algorithm animation is not new (see e.g., Zeus [14]); however our system tries to improve on other systems by proposing a completely OOP approach, starting from the algorithm design and ending with the visualization modules programming. From this approach, we have designed our system keeping in mind what we consider three main requirements for today's algorithm animation systems: reusability, remote access and interactivity. Our first decision has been to focus the animation more on data structures rather than on algorithms only introducing the concept of *animated data structure* as a data structure that is able to represent its behavior using an external visualization module. We believe that this focus on animating data structures has many benefits. First of all, it improves the reusability, as complex animated algorithms and data structures can be often obtained as a composition of simpler animated data structures. Furthermore, this often allows one to achieve a better and deeper understanding of the internal details of the algorithm, which is especially important at debug time.

Finally, it ensures a higher degree of safe interactivity, giving the opportunity to encapsulate dedicated functions to manipulate the state of the *animated data structure* invoked through animation.

This is a crucial point since 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, without giving much information on its internal data structures. 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. Here we give a more detailed explanation of the approach followed by Catai to fulfill the three requirements introduced above.

- **Reusability - Transparency**

  As we already said, Catai defines a standard roadmap that describe which are the steps needed to obtain a completely animated version of an input algorithm. These steps characterize the architecture of our system using a three-level approach. The first level (the bottom one) is the graphical representation of an input data structure, this representation refers to an abstract universal notion of the data structure and is independent from any implementation of the data structure self. In our system, this task is performed by a specialized visualization module named *animation window*. The second level acts as a link between an existing implemented data structure and the proper visualization module. More in details, it is in charge of representing the data structure behavior by interacting with the visualization module. All these functionalities are supported by extending the original data structure thus obtaining an *animated data structure*. In the third level an algorithm produces an animation by instantiating and interacting with several *animated data structures*. These interactions have the form of standard method invocations and they do not require the algorithm to be aware of the underlying animation. Using this approach we are able to transparently encapsulate the animation support into the data structures implementation in such a way that each *significant* operation to be executed is followed by a proper graphical representation. Each frame is able to work without knowing how the underlying frames have been implemented: in this way, it is possible for an algorithm to use transparently an *animated data structure* and, in the same way, a data structure can use a visualization module without explicitly dealing with graphical primitives.

- **Three-tier distributed architecture**

  We adopted a three-tier architecture based on distributed object technology. The main components are modeled as distributed objects and all the interactions have the form of remote method invocations. Animated algorithms and visualization clients may also reside on different hosts. Multi-user animations are obtained using replicated objects. Animated algorithms are implemented into the widely used `C++` language while the visualization modules are coded using the portable and device independent `Java` environment.

- **Interactivity**

  The support for interactions has always been one of the more discussed points in algorithm animation. The possibility to influence and to characterize the behavior of a running algorithm by interacting with its graphical representation has proved to be an hard task for both technical and logical reasons. From a technical point of view, the animation system should be able to control and to modify the normal execution flow of the algorithm together with allowing the user to access and to modify the data memory space of the process. For example, the user could be allowed to decide which is the next statement to be executed or to modify the content of an allocated variable. Of course, the effects of these interactions should be graphically represented as well as the ordinary animation. From a logical point of view, a deep interaction facility has many dangerous side effects, for example it could make it possible to bring the algorithm into a faulty state. CATAI faces these problems by introducing two different categories of interactions. The first category is made of explicit interaction requests coming from the algorithm and targeted to the users who are attending the animation. The algorithm can request the users to input some data or to interact with the displayed graphical representation. Using these interactions the coherence of the execution state is preserved. The second category of interactions allows the users to invoke at any moment some methods on the running algorithm. During the invocation of these methods the algorithm execution is suspended. Using these methods it is possible to access directly the content of the algorithm's data structures or to execute any code that can be encapsulated in an ordinary method call. The programmer of the animation is in charge of preserving the algorithm's coherency by defining the interactions in such a way that they cannot damage the algorithm's state.

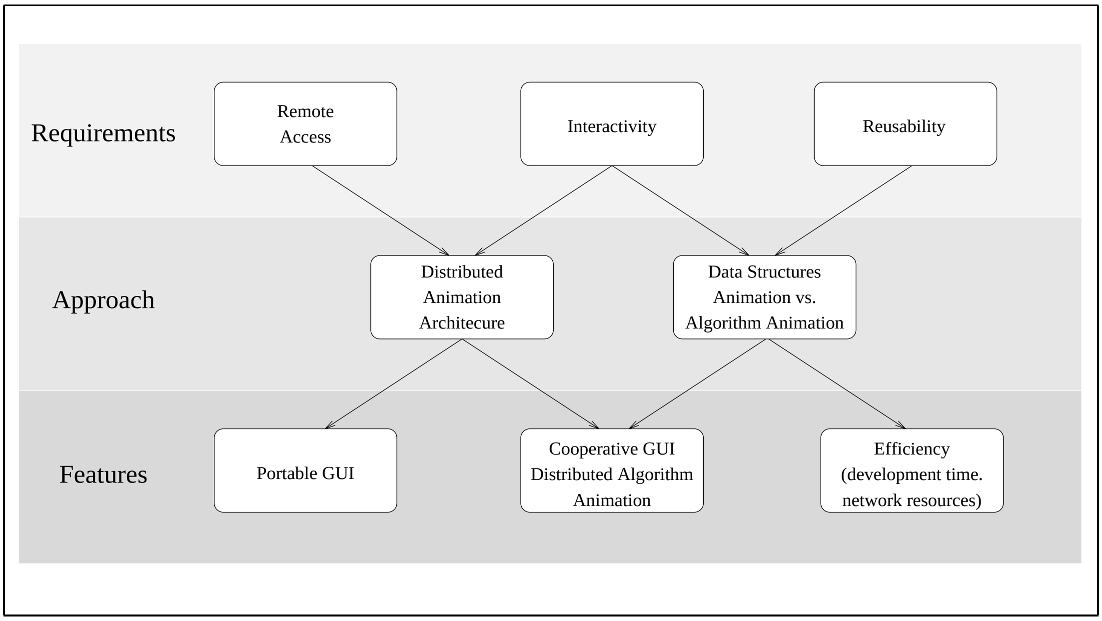

Figure 1: The main ideas behind the design of CATAI.

3 The Architecture of CATAI

As we already said, CATAI uses a distributed `C++/Java` mixed architecture. Due to its complexity, we start introducing the user perspectives of our system and then we present the architecture of CATAI in a bottom-up fashion. First, we describe the components of the CATAI system. Second, we introduce the communication and interoperation infrastructure used to make these components work together. Finally, we provide a list of all the animation-oriented services that have been implemented using this architecture.

3.1 CATAI User perspectives

The CATAI animation system can be considered according to three different perspectives: the animator perspective, the algorithmic programmer perspective and the final user perspective.

- **Animator Perspective**

  We denote by *animator* the person in charge of producing a graphical representation for a given set of data structures. To accomplish this job the *animator* has to code in `Java` a set of visualization modules whose purpose is to offer an abstract universal effective graphical representation of an input data structure. Once such objects are provided, the *animator* has to interface them with the data structures that need to be animated. The results

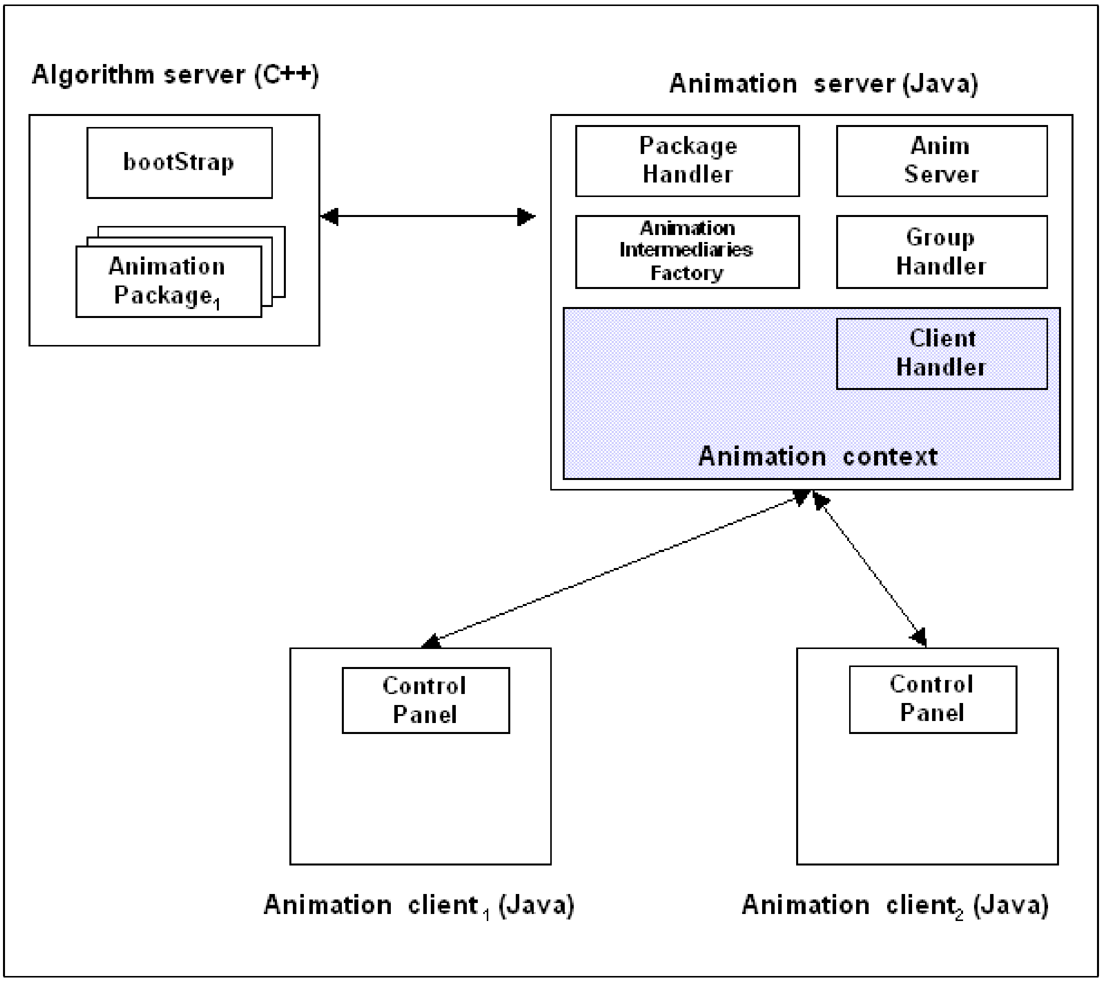

Figure 2: A snapshot of CATAI at start-up.

are the *animated data structures*, these objects retain the behavior of the original data structures while adding them the support for the animation.

- **Algorithmic programmer Perspective**

  We denote by *algorithm programmer* the person in charge of implementing from scratch an animated algorithm or revising an existing implementation in order to add the animation support. In the first case, the *algorithmic programmer* has to make use in its implementation of *animated data structures* in the second case he has to substitute the original data structures with animated ones. The animation obtained so far could be further improved by enriching the algorithm implementation with animation methods, these are special purpose functions used to enhance the presentation of an animated algorithm.

- **Final user Perspective**

Due to its flexibility, CATAI can be used for different purposes. Final users could be a group of students that are attending an algorithm course using CATAI as a learning media that allows them to better understand the way some data structures work. In the same way, CATAI could be ever used by an algorithm developer that is interested to check the behavior of an algorithm using an abstract high-level debugging tool. In both cases, the final user is a person that owns a local `Java` client application and uses it to connect to a remote server where a collection of animated algorithms are stored. The user gets the representation of the algorithm on its computer while the same algorithm is running on a remote machine. The communication between these two components is not direct, it uses an intermediary level needed to handle and to synchronize communications when there are several persons interested to view the same algorithm.

3.2 CATAI Components

In order to efficiently support multi-users reusable animations CATAI adopted a distributed architecture made of three distinct components. The **Algorithm server** is a C++ coded program that implements several sets of animated algorithms, animations are visualized to final users using **Animation clients**, these are lightweighted Java applications in charge of displaying the animation of a remote algorithm. Multiple users can be connected and can interact with a same animation using an **Animation server**, this component is coded as a Java application. Mainly, an **Animation server** has two principal purposes, to receive and to forward the animation directives coming from a running animated algorithm to all connected **Animation clients** and to receive, to synchronize and to issue interaction requests coming from the **Animation clients**.

- **Algorithm server**

  The algorithm server (*AlgServer*) implements a repository of `C++` algorithms coded using the libraries of CATAI. The *AlgServer* allocates new running instances according to the instructions received from a remote client application (*AnimClient*). Animated algorithms are grouped into *animation packages*, these are implemented as CORBA objects. An *animation package* object has as many methods as is the number of animated algorithms it offers. Each of these methods, when invoked, runs a different animated algorithm. An algorithm server can implement multiple *animation packages*, in order to allocate one of them a standard object factory (`bootStrap`) is provided. This object offers some remote methods that can be used to create new instances of *animation packages*. A remote animation client interested into an animation needs to allocate the proper *animation package* and then it can select the desired animated algorithms inside the allocated package. The visualization of animated algorithms is made possible by the use of *animated data structures*: these are objects that describe the behavior of the data structure they implement by sending animation messages to remote visualization modules. The execution flow of a running algorithm can be influenced by the interactions of remote animation clients.

- **Animation Server**

  The animation server (*AnimServer*) acts as a broker between the animated algorithms and the animation clients. It forwards the animation messages from the *animated data structure* to all the *animation windows*. Multiple users are organized using a group based approach. The animation server is able to handle concurrently several groups at once using *animation contexts*. Each context consists of a set of classes implementing high level animation-oriented functionalities and run in a separate thread. Some of these functionalities are: connecting the animated data structures to the proper visualization modules, implementing a virtual classroom paradigm and allowing the users to interact with the algorithms. All of the *animation contexts* make use of a set of standard services implemented by the animation server. These services are: accepting and initializing the clients connections and maintaining a list of all discovered *animation packages* together with additional information describing their status, their availability and their physical location.

- **Animation Client**

  The animation client (*AnimClient*) displays to the final users the visualization of a running animated algorithm. Animations are made possible by the use of a local set of `Java` visualization classes. Each final user can interact with the animation system by using a local animation control panel, in this way it can, for example, alter the speed of the animation, communicate with other users and select new animated algorithms. The representation performed by the *AnimClient* component is not just a passive visualization. By interacting with it using standard input peripherals (mouse and keyboard) it is possible to influence the behavior of the displayed algorithm.

3.3 The CORBA Framework

The communication and interoperation framework used by CATAI is CORBA. This technology allows one the creation of distributed applications modeled according to a distributed object programming paradigm. A CORBA distributed application is made of concurrent objects (even running in different remote processes) that *transparently* interoperate using the same semantic of ordinary methods calls. One of the principal advantages of CORBA is the possibility to write applications where components written using different programming languages and running on different operating systems can interoperate with little efforts.

The basic building blocks of a CORBA application are the *remote objects*, these are objects that can be located everywhere on the network and whose methods can be accessed by an application as if they were local. The remote methods implemented by a *remote object* are described using interfaces. An interface reports a set of services as a list of methods . In order to access the services offered by a *remote object*, a client application needs a local *proxy* object (here referred also as the *remote reference*). A *proxy* object implements the same interface of a remote object. Its methods, when invoked, determine where is physically located the remote object and send it the request invocation. After a request has been processed and executed, the remote object sends to the *proxy* object the return value. This information is used as the return value of the method invoked on the *proxy* object. Apart from its basic behavior, CORBA introduces several advanced services that can be used when building complex applications. The design of CATAI has been deeply influenced by the advanced features of CORBA listed here:

- **Naming Service**
  The Naming Service is in charge of creating and maintaining a table of mappings between string names and *remote reference* to remote objects. Any application can register a new entry by providing a name and a *remote reference*. By querying this service, a remote application is able to retrieve the *remote reference* of an object whose registration name is known.

- **Dynamic Invocation Interface**
  The Dynamic Invocation Interface (DII) allows an application to invoke a method on a remote object even if this is not known at run-time (i.e. the local *proxy* object is not available or it does not include the method to invoke). This is especially useful when new services must be supported without recompiling or rebuilding a distributed application. In order to make a dynamic invocation, a process has to assemble a method invocation request from scratch providing information like the name of the method to invoke, the list of parameters and the type of the return value.

- **Interface Repository Service**
  The Interface Repository Service (IRS) maintains a database of interfaces. This database can be used by a remote application to know which is the interface of a remote object. Additional interface definitions can be added at run-time. Using this service, an application is able to know which are the services offered by a given object even if it does not own a local *proxy* for that object.

3.4 The Infrastructure of CATAI

We now describe in details the solutions we have studied and implemented while designing the Catai architecture.

- **Animation clients start-up and registration**

  At start-up, the animation server publishes its *remote reference* toward the CORBA Naming Service using a standard name and then starts to wait for a connection requests from the clients. When started, the *AnimClient* asks the CORBA Naming service for the reference of the *AnimServer* using the network address provided by the user. If such registrations exists, the client uses the obtained *remote reference* to contact the server and register itself. During the registration the *AnimClient* invokes a standard initialization method on the server providing its *remote reference*. The *AnimServer* assigns a unique ID number to the client and adds a new entry to an internal table reporting the list of connected clients. Then, the *AnimClient* is put into the default animation context. Finally, the initialization method ends by returning to the registering client its ID number.

- **Algorithm server discovery and initialization**

  As we already stated, Catai animations are grouped into *animation packages*, and each algorithm server is able to offer several of these packages. In order to allow a remote animation client to know which are the available packages and animated algorithms, the algorithm server uses the CORBA IRS service. This service is used at start-up to register the interfaces of all the packages the server is offering together with the interface of the `bootStrap` object. After providing these information to the IRS service, the algorithm server publishes its *remote reference* toward the CORBA Naming Service using a standard name and then starts to wait for a remote method invocation.

  On the animation server side, at start-up there are no information available about the location of any algorithm server. After being started, the animation server waits for an *algorithm server discovery request* from one of the connected clients. This request reports the network address of the Naming Service to be queried. Upon receiving such request, the algorithm server contacts the Naming Service using the previously obtained address. If the service is available, the animation server asks for the *remote reference* of the algorithm server. If available, the animation server uses the obtained reference to query the IRS service in order to retrieve the interface of the remote `bootStrap` object. This interface is parsed in order to determine which are the animation packages offered by that server. The list of the discovered packages is broadcast to all clients and then stored in an internal animation packages list.

- **Communication services**

  Catai implements two different communication services. The first service is used to delivery animation messages to the animation clients through remote methods invocations. The second uses an internal message-based communication protocol for implementing additional services. We now describe them in more details.

  - **Animation messages delivery**

    In order for the animation to be represented, the messages generated by the animated algorithms must be delivered to all the connected clients. Since each message must be reported without changes to all clients, we have modeled the visualization modules as a single object replicated on several clients. In this context, the animation intermediaries act as *proxy* objects, the messages they receive from the *animated data structures* are replicated[2] and sent to all the visualization modules (Figure 3). Each instance of animation intermediary holds a list of references to a group of visualization modules (a single replicated object). When a new animation message is received, the animation intermediary dynamically builds a method invocation using the DII and sends it to all the clients. Client failures are handled by throwing an exception to the `clientHandler` object.

  - **Internal communication service**

    Each animation context features an internal communication service implemented by the `clientHandler` object. The services offered by this object allow an *AnimClient* both client-to-client communications and client-to-group communications. The `clientHandler` stores the *remote reference* of all the clients belonging to that context and a set of permissions used to determine which are the rights granted to each client. The communication protocol uses remote methods invocations to deliver string based messages. Different clients are distinguished using a unique ID number. Client IDs are used in every operation that requires a target client to be specified.

- **Dynamic interactions**

  Our system supports interactive animations implementing a mechanism to allow a user to influence the behavior of the running algorithm by interacting with the local representation. To this end we designed an event-based interaction scheme. Each time a user interacts with the algorithm representation using the mouse and the keyboard a new *animation event*

[2] By replication we mean the content of the animation displayed by each module, so we are not considering information like the mouse status or the position of the windows on the screen.

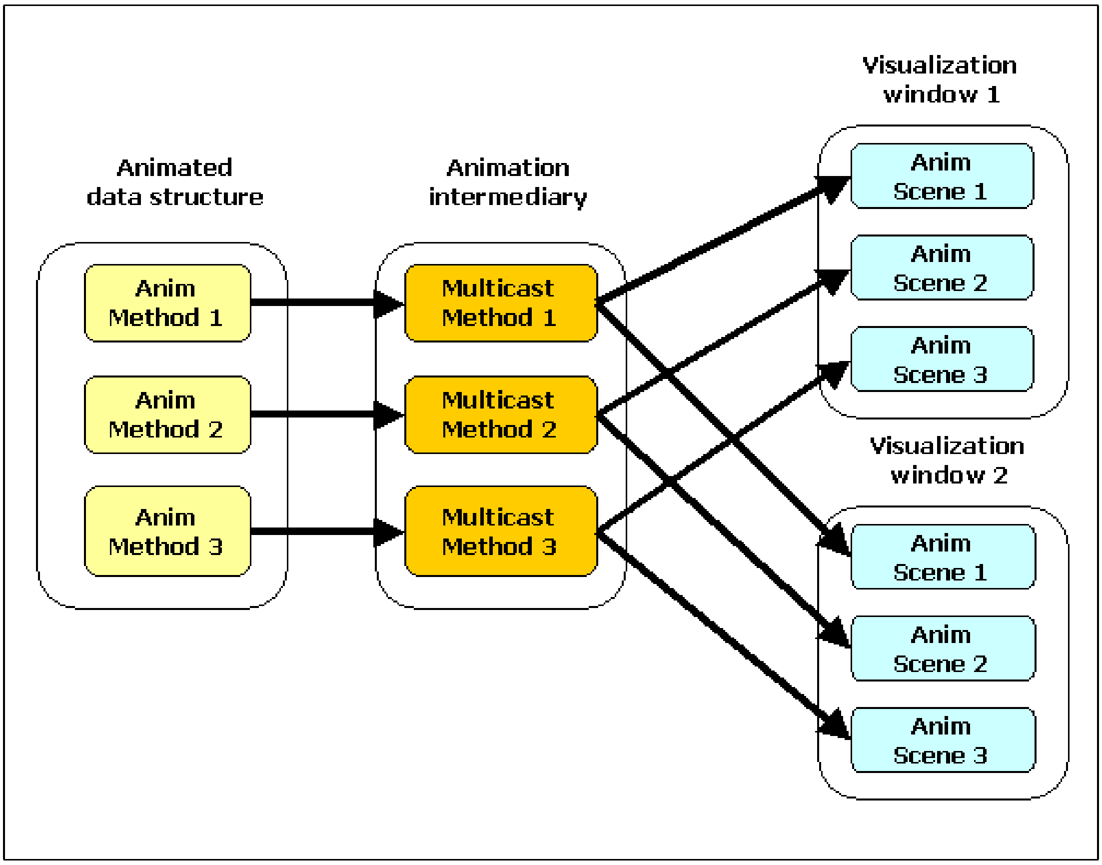

Figure 3: The scheme of the animation messages delivery.

is generated. These events report detailed information about the actions performed by the users (e.g. which graph node has been selected). *Animation events* are forwarded to the animation intermediary where they will be parsed. Each animation intermediary holds a virtual table that maps *animation events* into remote method invocations to be performed on the running algorithm. At startup this table is empty, in any moment an *animated data structure* can add new entries to this table. To do so, it must provide the id of the event to be handled, the name of the method to be invoked, the number of arguments to be used and, finally, which information of the animation event must be used as method arguments. These information will be used by the animation intermediary to dynamically assemble a new method invocation each time an animation event matches one of the entry of the virtual table. CATAI supports also the definition of new animation events at run-time. An *animated data structure* can define a new *animation event* by specifying a unique string name and, optionally, the list of the information the event should report.

4 The Main Features of CATAI

4.1 Visualization Modules

CATAI animations can be described as sequences of *animation scenes* represented by visualization modules and inferred by the *animated data structures* used by a running algorithm. Visualization modules are built using a standard representation framework provided by CATAI. This framework is based upon the concept of animation objects (`anim_objects`). These are the building blocks of CATAI animations. All animation objects have a standard set of attributes, such as color or shape, that can be customized by each *animation window*. Relationships between several animation objects be expressed via animation links (`anim_links`): in particular, `anim_links` define a set of animation objects with a common property. Generally, a data structure can be animated by representing its state and its behavior via animation objects, animation links can be used to represent the relations existing between items in the data structure. CATAI provides a basic *animation window* (`animWindow`) that implements a set of *animation scenes* that can be used to create, manipulate, represent and destroy both animation objects and animation links. The `animWindow` performs the visualization using a graphical panel, animation objects and animation links are visualized according to their attributes and to the representation methods implemented by the `animWindow` class. To this end, all visualization modules implement two standard representation methods, `paint_anim_objects` and `paint_anim_links`, that are in charge of defining how animation objects and links must be represented.

In order to build a new visualization module, the standard `animWindow` class must be derived and extended. The new class must redefine the representation methods (`paint_anim_objects` and `paint_anim_links`) according to the data structure it visualizes. Together with these methods, the new module has also to define all the additional *animation scenes* to be used for rendering typical data structure behaviors (e.g. defining a graphical representation for rotations in a binary tree visualization module). All the *animation scenes* are encapsulated into CORBA remote methods that can be directly invoked from the *animated data structure*. In this way an *animated data structure* is able to infer the contents and the appearance of an animation without explicitly dealing with graphical primitives but delegating this task to the proper visualization module.

The CATAI representation framework supports both *smooth animations* and *discrete animations*. The first are those animations where the changes occurring in a data structure are rendered using smooth graphical transitions, the second are those animations where any changes in the data structures is rendered instantaneously without using graphical transitions. Smooth transitions are able to better present the details of the behavior of an algorithm, on the other side as pointed by Stasko [35], discrete transitions are useful when the input data set is very large or when the users have fully understood the insight's of the animated algorithm.

CATAI supports both kind of transitions by defining an *animation delay* control. All the instructions devoted to the manipulation and the representation of animation objects and animation links are bound to a delay value set by the user. If the *animation delay* is set to zero, all the instructions will be executed in a unit of time without transitions, otherwise the execution will take an amount of time proportional to the delay value set.

4.2 Animated Data Structures

The animated version of a given data structure is obtained by properly interfacing the existing data structure implementation with a visualization module (See figure 4). To perform this task, the class implementing the data structure is extended in order to add animation capabilities. The animated class is obtained using multiple inheritance from the original non-animated class and the virtual class `AnimatorInvocator`. This last class provides all the functions needed to instantiate a proper visualization module and to communicate with it. Moreover, it implements a remote interface towards all the methods present in the standard *animation window* visualization module. By simply deriving this class, an *animated data structure* is able to access all the features of *animation window* without extra efforts. The constructor of the *animated data structure* needs to specify which visualization module must be loaded to represent its behavior. We distinguish in an *animated data structure* five categories of methods:

- *Animated methods*

  These are the methods defined by the original data structure that need to be animated. We map each of these methods into one or more *animation scenes*. To do so, the new definitions maintain the same behavior of the original implementation except for some remote method calls used to activate the proper *animation scenes* on the visualization module.

- *Standard methods*

  These are methods defined by the original data structure implementation that do not need to be animated. They are left untouched.

- *Animation-only methods*

  These methods are introduced to represent special purpose *animation scenes* that do not correspond to any existing method of the original data structure implementation. The invocation of these methods does not change in any way the state of the underlying data structure but affects only its representation.

- *Interaction methods*

  These methods are used by the animation to interact with the running algorithm. Since CATAI allows to create interactive animations, each *animated data structure* can define a set of CORBA remote methods that can be directly invoked by users. These methods can take input arguments from the animation using an event-based messaging system.

- *Service methods*

  These methods are defined by the `AnimatorInvocator` class. They are used to handle the connection with a remote *AnimServer*, to request and to maintain a *remote reference* to a visualization module and to configure the interactions methods.

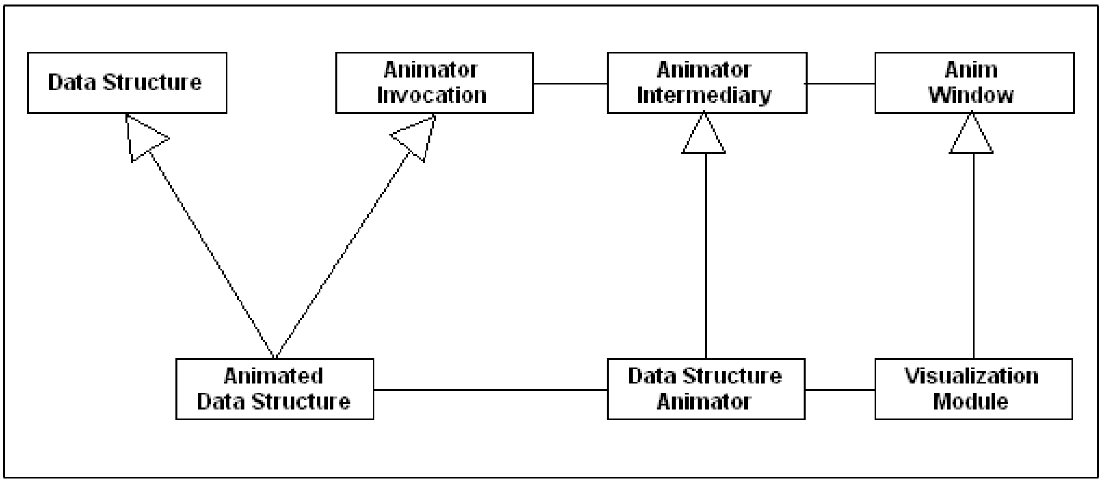

Figure 4: The structure of an *animated data structure*.

The object-oriented approach makes possible to transparently animate an existing algorithm by simply developing and using animated versions of its significant data structures. These data structures can be further extended or integrated by explicitly invoking additional *animation scenes*.

4.3 Sharing and Collaborating on a Same Animation

CATAI implements multiple users animations through the replication of the visualization modules on several clients. The introduction of an intermediary level allows the animated algorithms to send their messages to multiple clients without having to deal explicitly with each single client. In addition, the existence of the animation intermediaries allows to delegate all the issues concerning network faults, users permissions and communication scheduling to a specialized distinct component. This holds also on the client side, each user can refer to a remote component that is persistent with respect to the animated algorithms. This allows a group of users to stay connected and to use multiple *algorithm servers* at once.

CATAI users are grouped into several distinct sets organized in *virtual classrooms*, each classroom is mapped into a different *animation context*. Inside a classroom, CATAI distinguishes between a *teacher* user and several *student* users. When a user starts a new session, he is put into the default classroom, then he can change classroom. Each user has associated a set of attributes used to determine which are the permissions granted to him. These permissions report the system whatever the user is allowed to perform a set of restricted actions such as requesting a connection with a remote *AlgServer* requesting the allocation of a remote animation package and interacting with a running animated algorithm. When creating a new classroom a user becomes the *teacher* of that classroom, so gaining all the available permissions. A user entering an existing classroom will be registered as a *student*. A connection failure of the *teacher* is handled converting in a *teacher* the first *student* to have joined the classroom. CATAI provides to *teacher* users two additional tools, a *teacher control panel* and a *log book*. The *teacher control panel* allows the teacher to make questions to one or to all the users present, the user's answers are recorded and then available for browsing in the *log book*. Finally, the *teacher* can choose to grant some of the existing permissions to one or more *students* for letting them experiment the algorithm behavior.

4.4 Real-time Interactions

CATAI implements two different kinds of interactions, the first kind of interaction, here referred as *synchronous*, allows the running algorithm to query for information the animation and the users that are attending it. The second kind of interaction, here referred as *asynchronous*, allows the user to explicitly steer the algorithm execution by invoking remote methods:

- *Synchronous interactions*

  These interactions allow the algorithm to pause its execution and to query the running algorithm or the users who are attending it for some information. Animation queries are implemented as a set of standard remote methods available to all the *animated data structure*. CATAI supports both low level and high level animation queries, low level queries are targeted on the visualization framework while high level queries are targeted to the users. In the first case, an *animated data structure* can acquire information about the current state of its representation (e.g. the color of a given object, the list of animated objects previously selected by the user). In the second case, the algorithm can ask the user to input some data or to select one of the items displayed in the animation (e.g. choosing the weight of a tree edge or selecting a node to delete). Using these primitive, a user is able to build from scratch the data set to be used by an animated algorithm. Finally, as pointed by Gloor [35] many interesting situations can only be represented by an algorithm while processing particular input data set. For this reason it is desirable to have the possibility to assemble different input data sets and then choosing at run time which of them use. To solve this problem, the CATAI interactions primitives implements a simple file requester that can be used to pick a file local to the running algorithm.

- *Asynchronous interactions*

  These interactions allow the user to implicitly invoke remote *interaction methods* on a running algorithm using the animation as a GUI. To make this possible, the *animated data structures* must communicate to their intermediary objects which are the animation events they want to catch and which are the methods to be invoked upon receiving one of the caught event. In this way, using the underlying event based interaction architecture, the users are able to influence the algorithm behavior by simply interacting with the local representation. Animation events defined at run-time by an *animated data structure* are presented to the user as menu-items. If the user clicks on one of these item the associated animation event will be generated, optionally these events could require the user to input some additional data (e.g. requesting the number of nodes and edges while invoking a graph generator).

5 Main Advantages of CATAI

We conclude this section with some more comments about the main advantages offered by CATAI.

5.1 Reusability and Transparency

These issues can be discussed according to two different perspectives, the animator perspective and the algorithmic programmer perspective. In both cases, we should mention that an object-oriented paradigm by itself offers reusability. This is especially true for the animator: when a new *animated data structure* must be created from scratch, the availability of a standard graphical framework implementing a set of animation oriented functions minimizes the time and the efforts to be spent for its development. A similar result can be achieved also for the data structures: a data structure can be animated by simply deriving an existing *animated data structure* and extending it with the addition of new animation methods. For instance, we can inherit an animated list class to define easily an animated stack class. On the algorithmic programmer perspective, it is crucial the possibility to fully reuse an animation in different contexts without paying extra efforts. Our system supports this notion of reusability by encapsulating the whole animation into the data structure implementation: in this way, the animation becomes a feature of the data structure and not of the animation. The choice of encapsulating the animation support in the data structure enhances also the transparency of our system since an algorithm can produce a complex representation of its behavior without even knowing which are the details of its representations. This can be useful when the algorithm to be animated is already implemented, in this situation we can animate the algorithm by just substituting its data structures with their animated counterparts.

Moreover, in case a standard animation is not enough to represent the algorithm behavior, it is possible to enrich it by using the animation methods offered by the *animated data structure* without explicitly using graphical primitives.

5.2 Interactivity

The solution proposed by CATAI for implementing fully interactive animations can be considered effective for several reasons. First of all, the implementation of a whole set of synchronous interactions primitives allow the algorithm to use the animation as a natural Graphical User Interface. Moreover, these primitives can be wrapped into data structures specific interaction functions in such a way to be transparently used. On the other side, the asynchronous interaction functions can be used to influence the algorithm behavior using an high-level approach. As we already said, the possibility to directly access and modify the data memory of the algorithm or to arbitrarily change the program counter can be devastating for the algorithm execution state. In order to minimize such risks, the asynchronous interactions supported by CATAI are implemented as methods. In this way, given an algorithm to be animated, it is possible to define exactly a standard interaction policy that must be necessarily complied with. For the same reason, during the execution of such interaction the algorithm is paused. We believe that this approach allows the creation of very complex interactions while minimizing the risks that a user could damage the algorithm execution state.

5.3 Multi-users Animations

Several users at the same time can interact with the animations produced by CATAI. The communication protocol has been designed in such a way to maximize the number of client connections that can be efficiently handled. The content of the animation is identical for all clients, however each user can decide some details of the representation like graphical objects positioning and label displaying. The support tools offered by CATAI can be used to transform a simple animation into an interactive collaborative lesson.

6 A Guided Tour on the Use of CATAI

We now describe in some details how to prepare an algorithm animation with the help of 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 Prim's algorithm for computing a minimum spanning tree (MST) of a graph [24].

Prim's algorithm initially partitions the nodes of an input graph into two distinct sets. The first set contains an arbitrary node while the second set contains the remaining nodes. At each iteration a node will be moved from the second set to the first so as to grow a minimum spanning tree. In order to achieve such result the node whose connecting edge to the grown minimum spanning tree has minimum cost (*light edge*) is chosen. The critical step of this algorithm is the computation of the *light edge*. This problem can be solved by using a priority queue to hold the list of the edge crossing the two sets. The edges are stored in the priority queue using as keys their cost, in this way the top element of the queue will be the edge whose cost is minimal. We refer to LEDA's implementation of Prim's algorithm [25], which make use of the class `node_pq` to implement node priority queue data structures.

6.1 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 priority queue or the minimum spanning tree resulting from the computation. Once this has been decided, the process of developing an animation can be broken in three different steps.

Creating a visualization module. As we have already said, the component in charge of graphically representing a data structure behavior is the visualization module which intuitively provides an animated interpretation of "how data structures work".

These modules are totally independent from the data structures being animated and can be easily reused. If our system does not support already visualization modules for the algorithm at hand, then we must develop the modules required by the algorithm. In our example of minimum spanning trees, CATAI contains already the modules needed to represent both graphs, priority queues and n-ary trees. 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. However we present the structure and the design of the class `graphWindow` that implements a visualization module for both directed and undirected graphs. This class extends the base class `animWindow` and it provides an implementation for the standard methods `init`, `paint_anim_objects` and `paint_anim_links`. Moreover, it defines some additional animation methods needed for a complex representation of animated graphs and, finally, it implements some local methods that can be used to redefine the animated graph layout (embedding functions). We have chosen the traditional representation for the animated graphs with the nodes represented as labeled spheres and the edges represented as straight lines connecting pair of nodes. To this end we represent each node using an animation object and each edge using an animation link. The standard support for the animation object representation offered by the `animWindow` class can be left untouched in the `graphWindow` class since it already represent animation objects as labeled spheres. On the other side, the representation of the edges is specific to the graph data structures and must be defined implementing the method `paint_anim_links`. The following code details the implementation of the `paint_anim_links` as it can be found in the `graphWindow`. The representation works by drawing each animation link as a straight line connecting the two related animation objects. An additional boolean flag (directed) is used to determine whatever the graph is directed or not, in the first case an additional arrow is drawn upon each edge.

```
 public void paint_anim_links(Graphics g, Vector links) {
    Point       fcoord, tcoord;
    double      a,b;
    int         count;
    animLink    currentLink;


     for (count=0; count<links.size(); count++){
        currentLink = (animLink) links.elementAt(count);
            // We consider one at time all the animation links
        fcoord = currentLink.getFrom();
        tcoord = currentLink.getTo();
            // We retrieve the location of the two animation objects shared by the current link
        g.setColor(currentLink.getColor());
            // We set the pen color to the link color
        g.drawLine(fcoord.x, fcoord.y, tcoord.x, tcoord.y);
            // We draw a straight line connecting the two related animation objects
        if (directed){
            // If the graph is directed then we draw an additional arrow to show the direction of the edge
           if (tcoord.x == fcoord.x){
              drawArrow(g, tcoord, new Point(tcoord.x,(int) (tcoord.y + fcoord.y)/2));
           }
           else{
              a = (double) ((tcoord.y - fcoord.y)/(tcoord.x - fcoord.x));
              b = tcoord.y - (a * tcoord.x);
              drawArrow(g, tcoord, new Point((tcoord.x + (int)(fcoord.x - tcoord.x)/2),
                tcoord.y + (int)(fcoord.y - tcoord.y)/2));
}}}}}
```

Figure 5: The implementation of the `paint_anim_links` method in the `graphWindow` class.

Creating animated data structures. Once the visualization modules are available, we need to revise the implementation of the original data structures to support some animation capabilities. As we have already said, CATAI offers a specialized `C++` library to assist in the development of animated classes. The principal component of this library is the `animatorInvocation` class, which provides animation server communication primitives and binding mechanisms between a data structure and the related visualization module. An animated class can be derived from the original non-animated class and from the `animatorInvocation` 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. In our running example of minimum spanning tree, the non-animated algorithm uses the LEDA graph and priority queue 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 `anim_graph` from the LEDA `graph` class and from the `animatorInvocation` 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 `anim_graph` class and the implementation of some animated methods is given in Figure 6. Among the reported methods we can distinguish *animated* methods as `new_node` and `del_node`, *animation-only* methods as `color_node`, *interaction* methods as `delNode`. The animated priority queue class can be defined with the same technique.

Animated algorithm. We are now ready to show how to animate the implementation of Prim's algorithm at hand (the implementation we are referring to is the one proposed by LEDA). Starting from the original code, we replace the standard graph with its animated counterpart. Next, we add some animation-specific code to highlight the behavior of the algorithm. For instance, we can choose to color *blue* all the nodes and the edges belonging to the spanning forest being grown. Moreover, every time we consider an edge as a candidate to be a light edge we color it *green*, if the edge is so then we color it *cyan* otherwise we restore its original color. The *blue* forest will converge to a minimum spanning tree. We detail the source code of the animated algorithm comparing it with the original code in Figure 7. As it can be clearly seen, this animation has required only slight modification in the original source code.

A more sophisticated animation. The animation presented in the previous section has been obtained at a very low cost by simply replacing the original graph data type implementation with the animated and by using some special purposes animation methods. As it can be seen, CATAI provides a transparent and elegant solution to the animation of algorithms without requiring to the algorithmic programmer any computer graphic programming skill. It is possible to obtain more complex animations by simply spending some additional efforts as it could be seen below. Here we present a second version of Prim's algorithm animation that uses some additional CATAI features. To be more precise, this version uses two additional visualization modules, the first module, namely `pqWindow`, is in charge of displaying the behavior of the LEDA priority queues, the second module, namely nTreeWindow, can be used to represent a generic n-ary tree. The `pqWindow` module has been used into the animation of the LEDA `node_pq` data structure while the `nTreeWindow` module is interface with a generic n-ary tree data structure to be used for animation purposes only.

By using these animated data structures we are able to show not only how Prim's algorithm deals with the input graph but also the contents of the auxiliary priority queue and the content of the spanning tree being grown. Apart from just representing a greater number of data structures, this animation makes also a deep use of the CATAI interaction and presentation primitives to allow the users to better understand and experiment the algorithm behavior. Among these we cite the `advice` function used to describe the users the meaning of each single step of the algorithm and the `pick_node` function used by the algorithm to allow the users to choose the graph node where to start the computation. Moreover, we redefined the *interaction* methods implemented by the `anim_graph` in such a way that each time that an edge or a vertex is added or deleted the Prim's algorithm is re-run. In Figure 8 we report the definition of a CATAI animation package containing the complex animation of the Prim's algorithm together with the redefinition of the graph *interaction* methods. Even if more complex than the previous animation, this version still requires only generic programming skills. All the animation features, even the most advanced, are wrapped into an high-level algorithmic oriented library and can be used by means of an ordinary method invocation.

```
class anim_graph : public graph , public POA_anim_graph, public
animatorInvocation{

protected:

graphAnimator_ptr graphAnim;
// graphAnim is the remote reference to the graph visualization module

public:

anim_graph(const char * stringAnimServ, CORBA::ORB_ptr orb):
    graph(),animatorInvocation(orb){initAnimation( stringAnimServ);}

// build anim\_graph as a graph plus animatorInvocation
// stringAnimServ is the stringified CORBA remote reference to the animation server
// orb is the reference to the CORBA framework


void initAnimation(const char * stringAnimServ){

    animatorHandler_ptr animServ =
            animatorHandler::_narrow(orb->string_to_object(stringAnimServ));
        // create a reference to the remote algorithm server
    String_var sGraphAnimator =
            animHandler->rifGraphAnimator(CORBA::string_dup(orb->object_to_string(this));
        // next request the allocation of a graph visualization module
    graphAnim = graphAnimator::_narrow(sGraphAnimator);
        // finally store the reference to the allocated visualization module
}

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

edge new_edge(node src, node dst) {
  edge e = graph::new_edge(src, dst);
       // create a new edge
  animatorInvocation::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 set_color(node v, int color) {
 animatorInvocation::color_obj((long) v, color);
        // set the color of the animation object associated to v  to color
}

void delNode(const char * targetNode) {

    long tNode = atol(targetNode);
        // targetNode is converted into a long representation
    del_node((node) tNode);
        // we convert tNode into a node reference and then we
        delete id
}

};
```

Figure 6: The definition of `anim_graph` and some of its methods.

Original Algorithm

```
void SimplePrim(graph *mainGraph){

 int solutionCost = 0;

 node_pq<int> PQ(*mainGraph);
 node_array<int> dist((*mainGraph),MAXINT);
 EDGE_COST = new edge_array<int>(*mainGraph);
 edge e;
 forall_edges(e, *mainGraph){
    (*EDGE_COST)[e] = mainGraph->inf(e);
 }

 T.init(*mainGraph, nil);
 node v;
 forall_nodes(v,(*mainGraph))
    PQ.insert(v,MAXINT);

 while (!PQ.empty()){

    node u = PQ.del_min();
    if (dist[u] != MAXINT){
        solutionCost += (*EDGE_COST)[T[u]];
    }

    dist[u] = -MAXINT;
    forall_inout_edges(e,u){
        v = mainGraph->opposite(u,e);
        int c = (*EDGE_COST)[e];
        if (c < dist[v]){
            PQ.decrease_p(v,c);
            dist[v] = c;
            if (T[v] != NULL){
            }
             T[v] = e;
        }
    }
 }

string msg("Minimum spanning tree obtained with
    cost: %d.", solutionCost);
cout << msg <<endl;

}
```

Animated Algorithm

```
void SimplePrim(anim_graph *mainGraph){

 int solutionCost = 0;

 node_pq<int> PQ(*mainGraph);
 node_array<int> dist((*mainGraph),MAXINT);
 EDGE_COST = new edge_array<int>(*mainGraph);
 edge e;
 forall_edges(e, *mainGraph){
    (*EDGE_COST)[e] = mainGraph->inf(e);
 }

 T.init(*mainGraph, nil);
 node v;
 forall_nodes(v,(*mainGraph))
    PQ.insert(v,MAXINT);

 while (!PQ.empty()){

    node u = PQ.del_min();
    mainGraph->set_color(u, BLUE);
    if (dist[u] != MAXINT){
        mainGraph->set_color(T[u], BLUE);
        solutionCost += (*EDGE_COST)[T[u]];
    }

    dist[u] = -MAXINT;
    forall_inout_edges(e,u){
        v = mainGraph->opposite(u,e);
        int c = (*EDGE_COST)[e];
        mainGraph->set_color(e, GREEN);
        if (c < dist[v]){
            PQ.decrease_p(v,c);
            dist[v] = c;
            if (T[v] != NULL){
                mainGraph->set_color(T[v], BLACK);
            }
             T[v] = e;
             mainGraph->set_color(e, CYAN);
        }
        else
            mainGraph->deselect(e);
    }
 }

string msg("Minimum spanning tree obtained with
    cost: %d.", solutionCost);
mainGraph->advice(msg);

}
```

Figure 7: Prim's algorithm for MST in its original (LEDA-like) version and its animation in CATAI.

```
MST2::MST2(const char * tempRef, CORBA::ORB_ptr o) {
  remoteRef = tempRef;
  orb = o;
  mainGraph = new anim_graph(remoteRef, orb);

  short args1[]= {STRING, 0, STRING, 1, STRING, 2, EOA};
  mainGraph->assign_event(LINK,            "newEdge", args1);
  short args2[]= {STRING, 0, EOA};
  mainGraph->assign_event(CLICKCTRLOBJ,    "delNode", args2);
}

void MST2::ComplexPrim(){
      int solutionCost = 0;
      anim_NTREE     NTREE(remoteRef, orb);
      anim_node_PQ<int> PQ(*mainGraph, remoteRef, orb);
      node_array<int> dist((*mainGraph),MAXINT);
      EDGE_COST = new edge_array<int>(*mainGraph);
      edge e;
      forall_edges(e, *mainGraph){
         (*EDGE_COST)[e] = mainGraph->inf(e);
      }
      T.init(*mainGraph, nil);
      nodearr.init(*mainGraph, NULL);
      node v;
      forall_nodes(v,(*mainGraph))
        PQ.insert(v,MAXINT);
      node u;
      u = mainGraph->pick_node("Pick the source node");
      PQ.decrease_p(u, PQ.prio(u)-1);
      while (!PQ.empty()){
          u = PQ.del_min();
          NTREE->new_node(index(u));
          mainGraph->set_color(u, BLUE);
          if (dist[u] != MAXINT){
              mainGraph->set_color(T[u], BLUE);
              solutionCost += (*EDGE_COST)[T[u]];
          }
          dist[u] = -MAXINT;

          forall_inout_edges(e,u){
              v = mainGraph->opposite(u,e);
              int c = (*EDGE_COST)[e];
              mainGraph->set_color(e, GREEN);
              if (c < dist[v]){
                  PQ.decrease_p(v,c);
                  dist[v] = c;
                  if (T[v] != NULL){
                      mainGraph->set_color(T[v], BLACK);
                  }
                  T[v] = e;
                  NTREE->new_node(index(v));
                  NTREE->make_parent(index(u),index(v));
                  mainGraph->set_color(e, CYAN);
              }
              else
                  mainGraph->deselect(e);
              }
           }

      string msg("Minimum spanning tree obtained with cost: /C1%lu/C0.
                  All blue edges belong to the MST", solutionCost);
      mainGraph->advice(msg);
}
```

```
void MST2::newEdge(const char *source, const char *target, const char *weight) {
  mainGraph->newEdge(source, target, weight);
  ComplexPrim();
 }

 void MST2::delNode(const char *source) {
  mainGraph->delNode(source);
  ComplexPrim();
}
```

Figure 8: Prim's algorithm complex animation.

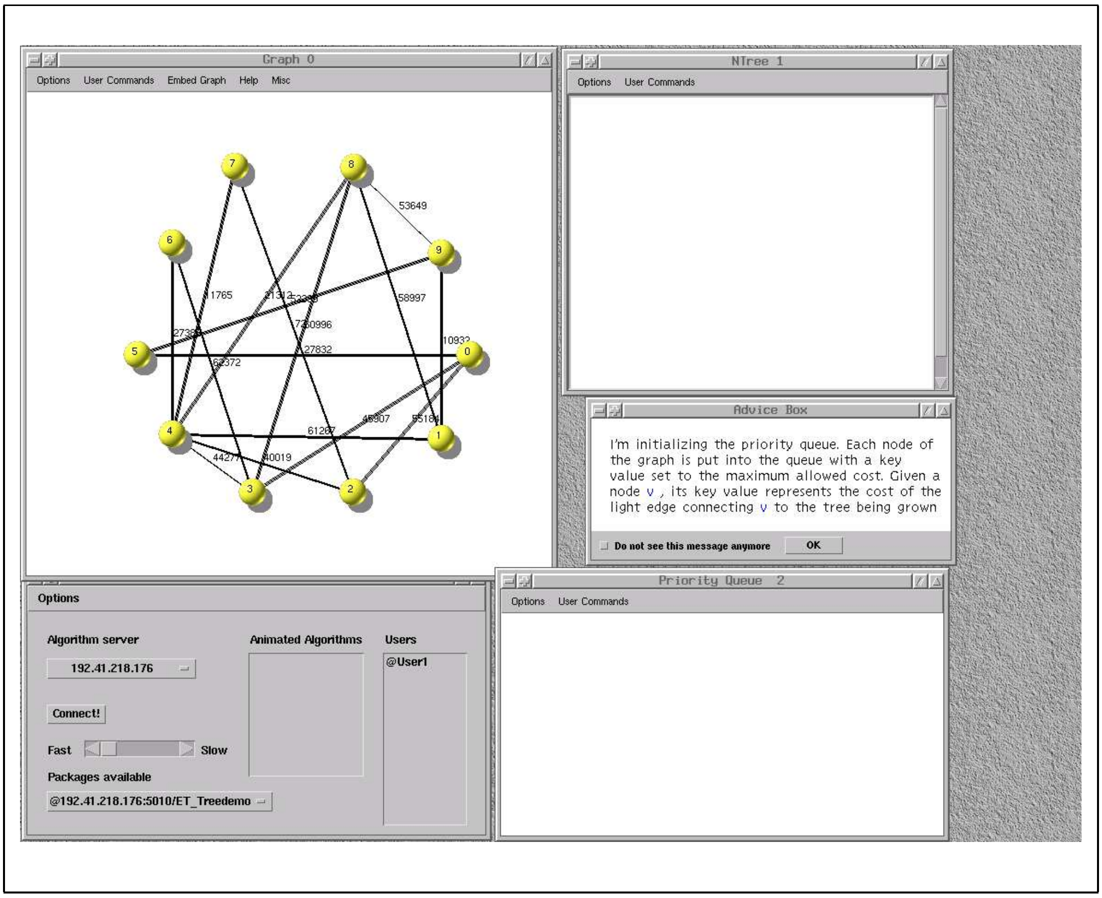

Figure 9: The animation starts on a graph. The priority queue is initialized by inserting in it each node of the graph with an initial cost set to the maximum allowed cost. All vertices in the graph are originally colored yellow, and all edges are colored black.

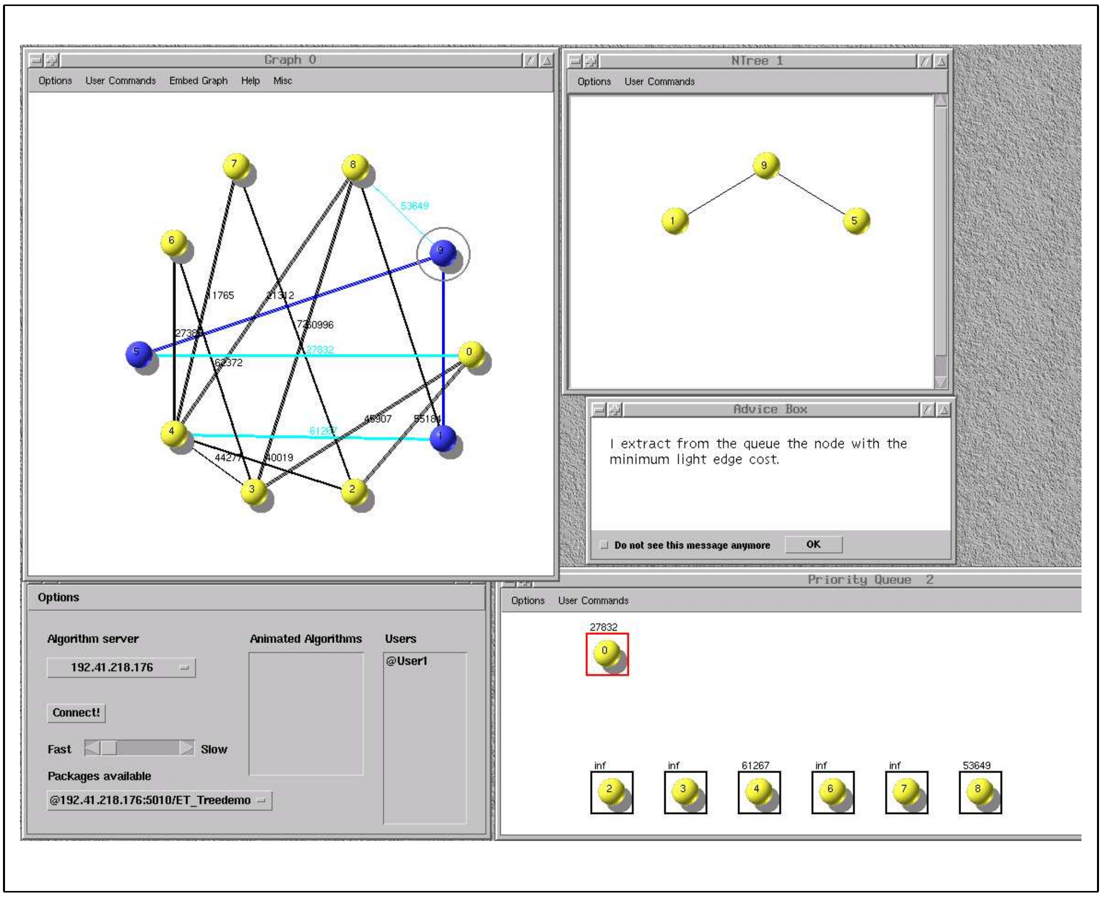

Figure 10: Prim's algorithm is running. The spanning tree grown so far has three nodes. Node 0 gives the current minimum light edge cost and is extracted from the priority queue. Light edges are colored CYAN in the graph window.

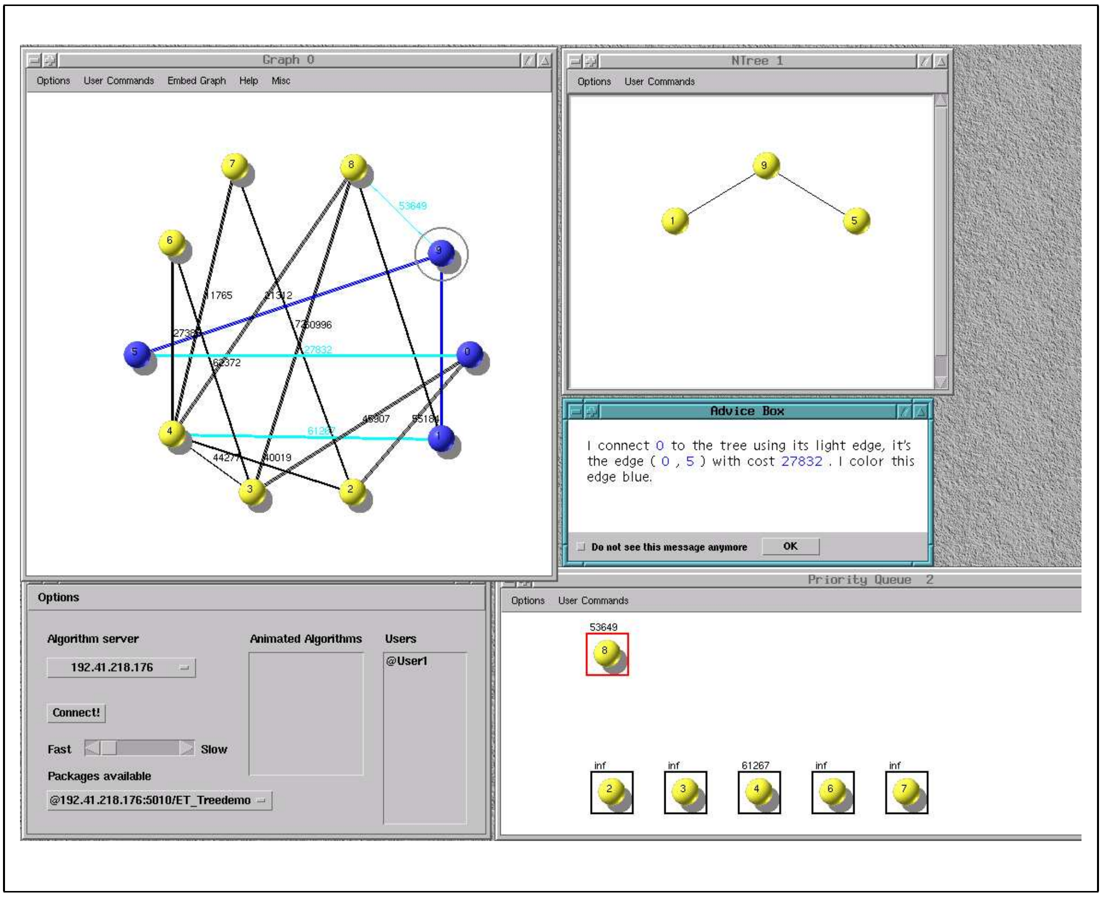

Figure 11: Node 0 is connected to the spanning tree using its light edge (0,5). All edges and nodes belonging to the solution are colored BLUE.

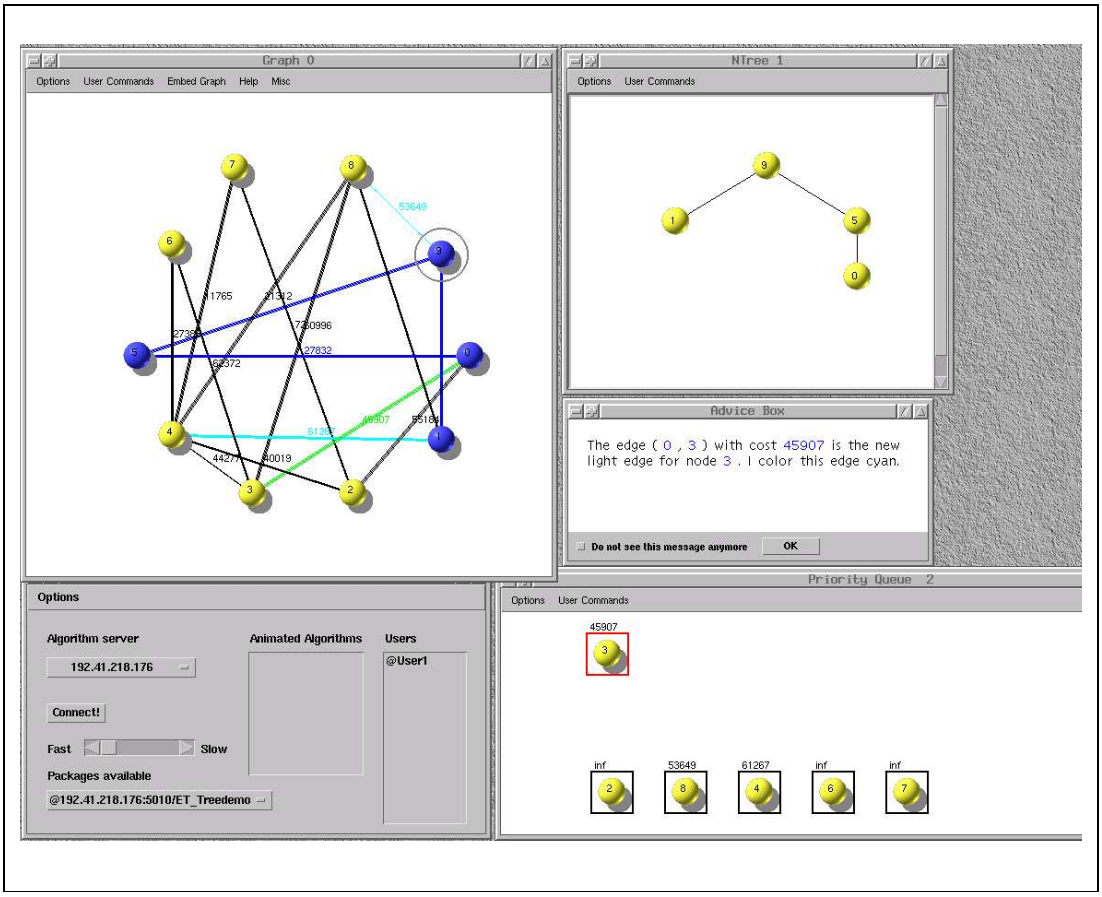

Figure 12: We start exploring all the edges outgoing from the node 0 searching for its light edge. The currently selected edge, the one colored GREEN, is the new light edge for 0. It is colored CYAN.

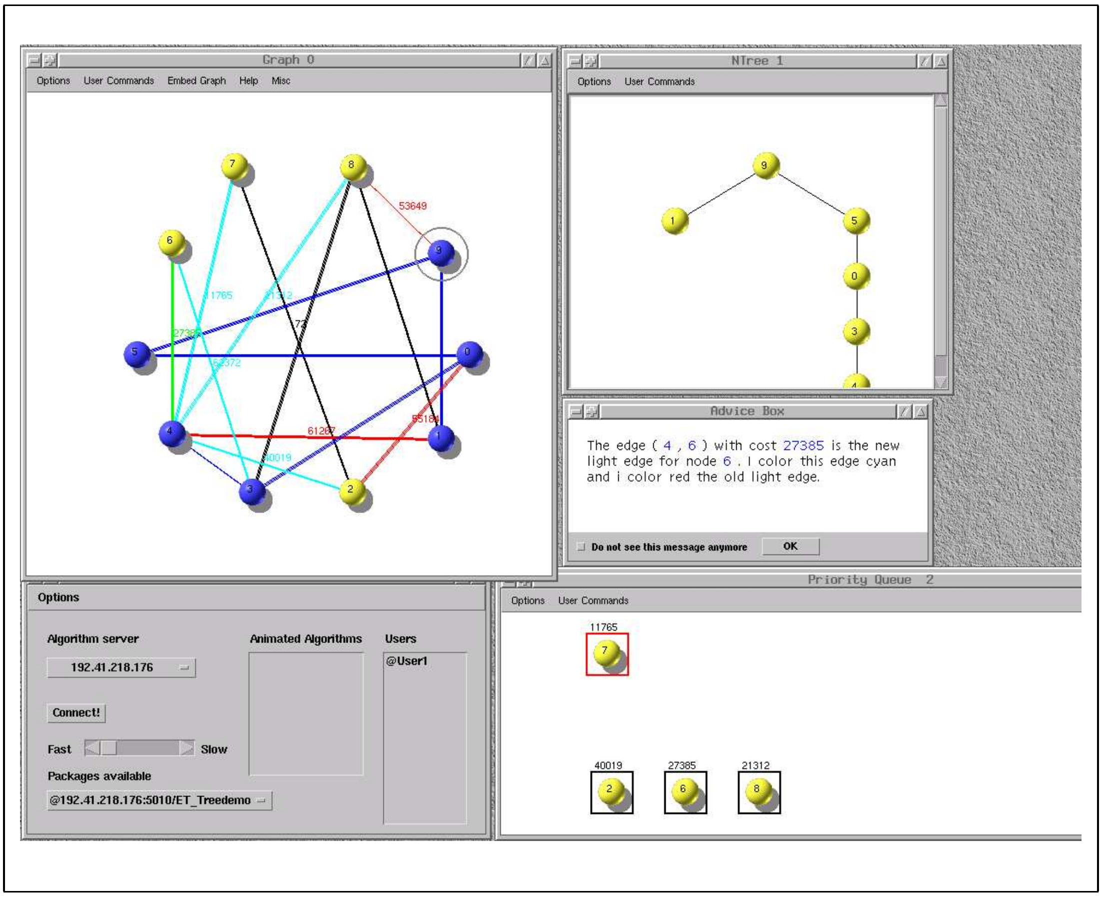

Figure 13: Edges incident to node 4 are explored. Edge (6, 4) is the new light edge for node 6 and replaces the edge (6, 3) which is colored RED.

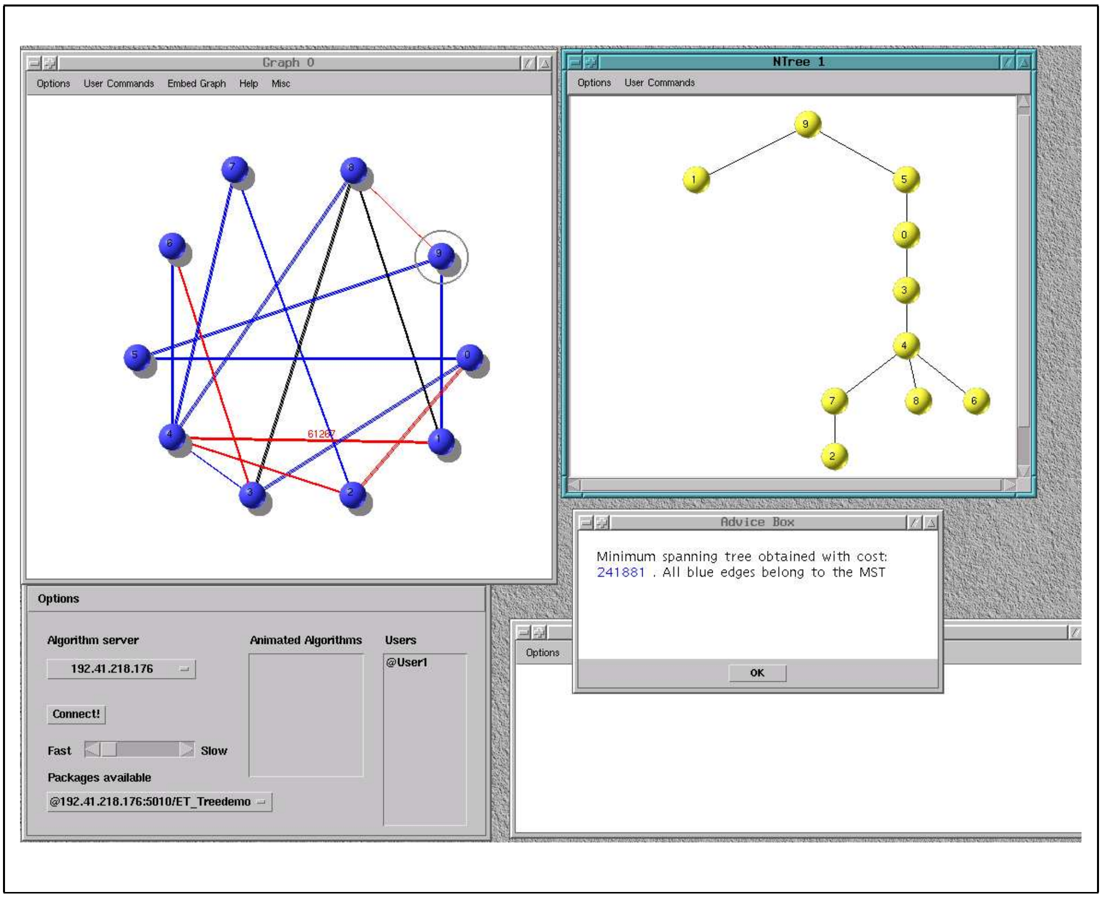

Figure 14: The algorithm has succesfully computed the MST for the input graph. BLUE edges belong to the solution.

7 Conclusions and Future Work

In our opinion, there are many benefits that could come from the use of algorithm animation could as a support tool during the developing of complex algorithms, however developing an efficient and flexible animation systems such CATAI is not enough. Nowadays, it is not realistic to think to code a complex algorithm without using a standard library of reusable basic data structures and algorithms (e.g. LEDA, STL). In the same way, we think it could be certainly useful to provide a library of basic data structures and algorithms already animated. These could be used as building blocks into the animation of more complex algorithms. Starting from the observations, we have started the developing of a library of animated data structures and algorithms. The data structures we are considering are the ones commonly used into the developing of algorithmic software (e.g. graph, stack, list, array), the reference implementation we are considering is the one offered by the LEDA library. Our goal is to obtain a collection of animated data structures that could be used as their LEDA original counterparts while providing a transparent animation support.

Traditional animation systems allow the programmer to directly access and modify the animated representation using graphical primitives. In our case, animated algorithms can influence their representations by just interacting with a remote component specialized into the representation of a family of data structure, the resulting animation represent exactly and in real-time the way the algorithm interacts with its data structures. If, by a side, this choice allows the algorithmic programmer to easily and transparently code an animation on the other side it does allow a far limited interaction on the graphical representation. To solve this problem we are investigating the possibility to introduce a director module in the CATAI animation model. This module should be in charge of orchestrating the animated representation taking decisions such as defining the duration of each single animation scene, merging together different animation scene and ...

References

[1] R.M. Baecker. "*Sorting Out Sorting*". Narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review #7, 1983. Los Altos, CA: Morgan Kaufmann.

[2] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. "A new model for algorithm animation over the WWW". ACM Comput. Surv., 27(4):568-572, 1995.

[3] 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.

[4] 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.

[5] J. Bentley, B. Kernighan. "A system for Algorithm Animation: Tutorial and User Manual". *Computing systems*, vol. 4, no. 1, 1991, pp. 5-30.

[6] J. Bentley, B. Kernighan. "A system for Algorithm Animation(Tutorial and User Manual)". AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Computing Science Technical Report 132, August 1991.

[7] G. Booch. "Object Oriented Analysis and Design with Applications". The Benjamin/Cummings Publishing Company, 1994.

[8] Brown, M. H., Sedgewick, R. . " Progress Report: Brown University Instructional Computing Laboratory". In *Proceedings of ACM SICCSE Bulletin*, vol.16,no.1,Feb.1984, pp. 91-101.

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

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

[11] M.H. Brown. "Perspectives on algorithm animation". In *Proc. of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pp.33-38. Washington D.C., May 1988.

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

[13] Brown, M. H. "Exploring Algorithms using BALSA-II",*Computer*, vol. 18, no. 8, August 1988, pp. 14-36.

[14] 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.

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

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

[17] C. Demetrescu. "Smooth animation of algorithms in a declarative framework". *In Proc. of the 1999 IEEE Symposium on Visual Languages, IEEE Computer Society Press*, 1999.

[18] R.Duisberg. "Animated graphical interfaces using temporal constraints". *In Proc. of the ACM SIGCHI '86 Conference on the Human Factor on Computing Systems*, Boston, MA, April 1996, pp. 131-136.

[19] M. Eisenstadt, M. Brayshaw. "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming" *Journal of Logic Programming*, vol. 5, no. 4, 1988, pp.277-342.

[20] D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig, "*Sparsification – A technique for speeding up dynamic graph algorithms*". In *Journal of ACM*, Vol. 44, 1997, pp.669-696.

[21] J. Foley and C. McMath. "Dynamic Process Visualization ". IEEE Computer Graphics and Applications, vol. 6 no. 2, March 1986, pp.16-25.

[22] E. Helttula, A. Hyrskykari, K.J. Raiha. "Graphical Specification of Algorithm Animations with ALADDIN". In: *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii, January 1989, pp. 829-901.

[23] R. Henry, K. Whaley and B. Forstall. "The University of Washington Illustrating Compiler". The ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, ACM, New York, June 1990, pp. 223-233.

[24] R. C. Prim. "Shortest connection networks and some generalizations". *Bell System Technical Journal*, 36:1389-1401, 1957.

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

[26] T.Moher, "PROVIDE: A Process Visualization and Debugging Environment", *IEEE Transactions on Software Engineering*, vol. 14, no. 6, June 1988, pp. 849-857.

[27] W. Pierson and S. H. Rodger. "Web-based Animation of Data Structures Using JAWAA". Twenty-ninth SIGCSE Technical Symposium on Computer Science Education, p. 267-271, 1998.

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

[29] G.C. Roman and K. Cox. " A Declarative Approach to Visualizing Concurrent Computation". Computer, vol. 22, no. 10, October 1989, pp. 25-36.

[30] G.C. Roman, K. Cox, C. Wilcox, J. Plun "Pavane: A system for Declarative Visualizing of Concurrent Computations". *Journal of Visual Languages and Computing*, vol. 3, no. 1, January 1992, pp.161-193.

[31] J.T.Stasko. "TANGO: A Framework and System for Algorithm Animation". PhD thesis, Brown University, Providence, RI, May 1989. Available as Technical Report no. CS-89-30.

[32] 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`)

[33] J.T.Stasko, E.Kraemer. "A Methodology for Building Application-Specific Visualizations of Parallel Programs". *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, June 1193, pp. 258-264.

[34] 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.

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

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