INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

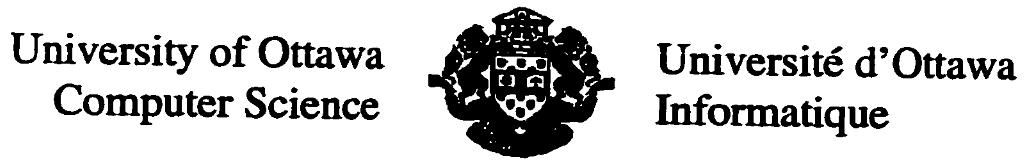
Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa



Interactive Animation of Ordered Sets Algorithms Using Three-Dimensional Graphics

Nabil Ben Saidane

Thesis submitted to the school of graduate studies and research in partial fulfillment of the requirement for the degree of:

Master of Science in Computer Science

Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada
January 1997

© **Copyright 1997, Nabil Ben Saidane, Ottawa, Canada**

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-20966-0

Dedication

To my Parents, Mounira and Hassen
To my Sister Rym
To my new-born niece Dahlia

Acknowledgments

I've looked forward to writing this page for quite some reasons. I would like to thank my supervisor, Dr. Najib Zaguia who provided full and unconditional support and a plethora of stimulating intellectual challenges. Najib Zaguia sets high standards for himself and those around him and more than anyone I know, appreciates one's obsession for detail and perfection.

I would also like to thank Mr. Pierre Fauvel for his continuous support. His encouragement, keen insight into technical matters helped me deal with lots of issues related to the system's implementation.

Many thanks to Dr. Georg Sander, Dr. Roberto Tamassia and Dr. George Gratzer for their comments and their constructive discussion and also to the people who read part of this thesis for their perceptive comments.

I would like to express my gratitude to the Government of the Republic of Tunisia and the Canadian International Development Agency (C.I.D.A.) for their financial support.

Abstract

"You do not understand anything until you understand it in more than one way"
Marvin Minsky

After reading the book *Combinatorics and Partially Ordered Sets* written by W. T. Trotter, we wondered how much more effective an interactive version of this book would be. Using hypertext linking techniques a reader would get immediate access not only to the referenced index entry point, but to all referred to or linked information. Such a system would become even more powerful if the algorithms were not only explained in words, but available as interactive animations which could be played with.

Algorithm animation is a form of program visualization that includes a number of specialized subareas that will be addressed in this thesis:

In the first part of the thesis, we will address the issue of data structure visualization, for instance, the structure of partially ordered sets (Posets). We will describe methods for visualizing Posets in 2D and then motivate the need and the importance of providing three-dimensional representation of these structures and discuss how 3D graphics can provide additional information to the structure, while maximizing readability and visibility through the use of computations and metrics.

In the second part we will focus on the area of the algorithm animation. We will introduce our model for abstracting the data, the operations and the semantics of computer programs, and the creation of graphical views of those abstractions. We will then, explain what is an interactive mapping (operation mapping) and their relations with input data and the algorithms, and finally we will present our system's architecture and discuss its components.

We emphasize that the potential of such algorithm animation environment is great, but can be fully realized only if they are sufficiently easy, highly interactive, and enjoyable to use. This dissertation is a step toward achieving these goals.

Contents

1. Introduction 1

1.1 Motivation and background 1
1.2 Definitions 1
1.3 Application of algorithm animation 3
1.4 Perspectives on graphics in programming 4
1.5 Requirements 6

2. Related work and thesis contribution 8

2.1 Previous work 8
2.2 Discussion 12
2.3 Thesis contribution 13
2.4 Thesis disclaimers 14

3. Data structure visualization 15

3.1 Problem domain 15
3.2 Uses of three-dimension 16
3.3 Ordered sets: From 2D to 3D 17
3.4 Concern for readability and clarity 28
3.5 Use of metrics 33
3.6 Chapter summary 42

4. Algorithm Visualization 44

4.1 Abstractions 44
4.2 Underlying computation 45
4.3 Visualization process 54
4.4 Rendering process 60
4.5 Chapter summary 63

5. Interactive environment 64

5.1 Case Sudy #1 64
5.2 Case Sudy #2 69
5.3 System's Architecture 72
5.4 System's Components 74

5.5 3D Navigation 78

6. Conclusion 81

6.1 Evaluation 81
6.2 Research Directions 81
6.3 Final Thoughts 82

Annex I

References

1. Introduction

1.1 Motivation and background

For many years the primary way to understand a computer program's or process' execution was to examine its source code and utilize a debugger or sometimes using hand simulation. During the last decade, with the advance of hardware: personal workstations - with their high-resolution displays, powerful dedicated processors, and large amounts of virtual and real memory - can support the required interactiveness and dynamic graphics. In the future, such workstations will become cheaper, faster, and more powerful, and will have better resolutions. an algorithm animation environment exploits these characteristics, and can also take advantages of a number of features expected to become common in future hardware, such as sound, and parallel processors.

Nowadays, it becomes crucial to formally investigate questions of how animation may best be incorporated into areas such as animated graphics, animated interfaces and animated presentation of concepts such as data, programs and for instance algorithms, to help gain a better understanding of the inherent process of an algorithm and to prove usefulness in the field of design and analysis of algorithms.

1.2 Definitions

1.2.1 Algorithm animation

The terms algorithm animation [Bro88], program visualization [Mye90] and software visualization all have been used to describe systems seeking to aid program understanding. our view of the scope of these terms is in Figure 1.1 [Sta92].

Algorithm animation (visualization) is the most specific category, it is considered as being the process of abstracting data, operations and semantics of algorithms, and the creating of animated graphical views of these abstractions.

Program visualization illustrates data structures, program state and program code as well. *Software visualization* illustrates computer processes and data in addition to regular programs.

Finally, *computation visualization* includes both software and hardware views. In the dissertation, we will use the term "computation visualization" as a general encompassing notion.

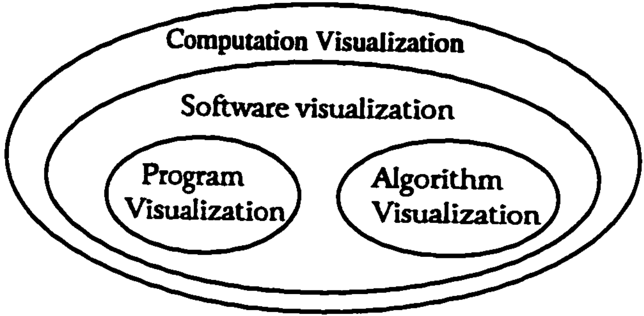


Figure 1.1: Scope of the terminology.

1.2.2 Visual Programming

Visual programming (VP) [Mye90] refers to any system that allows the user to specify a program into a two-(or more)-dimensional fashion in contrast with conventional languages which are not considered two-dimensional since the compilers or interpreters process them as long, one-dimensional process. Visual programming does not include systems that use conventional (linear) programming languages to define pictures. It also does not include drawing packages like draw utilities, since these do not create 'programs' as defined above.

1.2.3 Program Visualization

'Program visualization' (PV) [Mye90] is an entirely different concept from Visual Programming. In visual programming, the graphics are used to create the program itself, but in Program Visualization, the program is specified in a conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time execution.

Program visualization systems can be classified using two axes: whether they illustrate the *code*, *data* or *algorithm* of the program, and whether they are static or dynamic.

'*Data visualization*' systems show pictures of the actual data of the program. Similarly, '*Code visualization*' illustrates the actual program text, by adding graphical marks to it or by converting it to a graphical form (such as flowchart).

Systems that illustrate the '*algorithm*' use graphics to show abstractly how the program operates. This is different from data and code visualization, since with algorithm visualization the pictures may not correspond to specific pieces of the code. For example, an algorithm animation of a sort-routine might show the data as lines of different heights, and swaps of two items might be shown as a smooth animation of the lines moving. the 'swap' operation may not be explicitly in the code.

'*Dynamic*' visualization refers to systems that can show an animation of the program running, whereas '*static*' systems are limited to snapshots of the program at certain points

1.2.4 Interpretive vs. Compiled

Any programming language system may either be 'interpretive' or 'compiled'. A compiled system has a large processing delay before statements can be run while they are converted into lower-level representation in a batch fashion. An interpretive system allows statements to be executed when they are entered.

1.3 Applications of algorithm animation

An obvious application of an algorithm animation environment is *instruction* in general, and computer science in particular, especially courses dealing with algorithms and data structures, e.g., compilers, graphics, databases, algorithms, programming. Rather than using a chalkboard or viewgraph to show static diagrams, instructors can present simulations of algorithms and programming concepts on workstations. Moreover, students can try out the programs on their own data, at their own pace, and with different displays (from library of existing displays) from those the instructor chose. Students can also code their own algorithms and utilize the same set of displays used by the instructor in demonstration programs.

Another proven application of an algorithm animation environment is a tool for *research* in algorithm design and analysis. Human beings' ability to quickly process large amounts of visual information is well documented, and animated displays of algorithms provide intricate details in a format that allows us to exploit our visual capabilities. For instance, a variation of shellsort was discovered in conjunction with static color displays of bubblesort, and standard shellsort [Bro88].

Animations developed for instruction can be used for research, and vice versa. For example, animations for an algorithm course on graphs were used with minor changes to investigate shortest-path algorithm on other types of graphs [Bro88]. Conversely, displays developed in conjunction with research on pairing heaps were later incorporated into classroom lectures on priority queues [Bro88].

Another application of an algorithm animation environment is a testbed for *technical drawings* of data structures. It allows interactive experimentation with input data and algorithm parameters to produce a picture that best illustrates the desired properties.

A prime but so far unexplored application area of algorithm animation is in *programming environments*. Pioneering environments on graphics-based workstations, such as Cedar [Tei84], Interlisp, and especially Smalltalk, are, by and large, text-oriented. Recent workstation-based program development environments incorporate graphical views of the program structure and code, not data, and consequently have had limited success in giving additional insight into the programs: These systems could be enhanced by the display capabilities of an algorithm animation environment.

Algorithm animation has also been used for *performance tuning* [Dui86], and has the potential to be helpful in *documenting programs* [LD85] and in *systems modeling*, especially for multi-threaded applications.

1.4 Perspectives of graphics in programming

1.4.1 Advantages of using graphics

The human visual system [Mye90] and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in one-dimensional textual form, not utilizing the full power of

the brain. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured program, have long been known to be helpful aids in program understanding. A number of Program Visualization [Bro88] systems have demonstrated that two-dimensional pictorial displays for data structures, such as those drawn by hand on a blackboard, are very helpful. Researchers claim that graphical programming uses information in a format that is closer to the user's mental representations of problems, and will allow data to be processed in a format closer to the way objects are manipulated in the real world. It seems clear that a more visual style of programming could be easier to understand and generate for humans, especially for the user user, non programmers or novice programmers.

Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (other de-emphasizing issues of syntax and providing a higher level of abstractions) and may therefore make the programming task easier even for professional programmers. This may be especially true during debugging, where graphics can be used to present much more information about the program state (such as current variables and data structures) than is possible with purely textual displays.

1.4.2 Dimension Usage

Why three-dimensional graphics ?

Andy van dam, in his keynote speech at the 1991 UIST conference, implored researchers to "escape flatland" and explore how 3D graphics can be used in user interfaces and information displays

Three-dimensional graphics have begun to appear in certain information displayers, primarily those of data. For example [Sta92], the information visualizer from Xerox Parc is a system that uses three-dimensional imagery to present structured information such as computer directories and project plans. Scientific visualization [Baker] presents yet complex scientific data from areas such as meteorology, biology and chemistry in a 3D format.

The system's designers noted that the three-dimensional computation visualization, for instance the three-dimensional algorithm animation help shift the viewing process and

the understanding of the computation from being a *cognitive* task to being a *perception* task. This transfer helps to enable human's pattern matching skills.

1.5 Requirements

Our approach to building 3D computation visualization would be to use an existing sophisticated 3D animation package or library such PHIGS+, GL, or RenderMan. Although appealing in a general sense, the effort involved in such a project would not warrant the understanding or comprehension gain achieved, if any, by the animation for two key reasons: First, graphic systems for building such realistic imagery have steep learning curves and they require familiarity with 3D graphics concepts, terminology, and techniques. Second, computation visualization does not require all the power these systems provide. A package with lesser capabilities could be able to well provide the kind of information displays that computation visualization exhibit. Our goal here is to find a middle ground in the inevitable trade-off between image sophistication and ease-of-use.

Computation visualization must illustrate how computers and software are used to perform computations and solve problems. The domain of computer hardware and software is a restricted world: In hardware, we use processors, memory registers, CPU's and so on. Software involves programs with statements, data structure, code, and operations.

Below we list the basic requirements for 3D computation visualization.

1. The visualization must provide a suite of graphical objects including lines, rectangles, circles, polygons, arrows, blocks, spheres, and text. Color displays are critical.
2. The graphical objects must be able to undergo changes in position, size, color, and visibility, thereby providing a sense of animation.
3. The visualization must be able to adapt to run-time execution description data that controls how the display appears.

The final requirement also preludes the use of many preexisting 3D animation toolkits, which are typically used for pre-determined, precisely-specified scenarios. Computation views are parameterized animations; they cannot be hard-wired, predefined

animation sequences. The views must be adaptable, presenting imagery that corresponds to programs' run-time conditions, data and input.

Unfortunately, the term 'animation' has been loosely applied in the past, and a large variety of systems claim to provide animation capabilities. For example, actions such as simply highlighting lines of code as they are executed, altering the boundary style of a graphical object, or changing color intermittently have been called animation.

Fundamentally [SP91], animation consists of the rapid sequential display of pictures or images, with the pictures changing gradually over time. These pictures are the frames of the animation. If the imagery's changes from frame to frame are small enough and the speed of displaying the frames is fast enough, the illusion of continuous motion is achieved.

We consider data structure animation systems and program state animation systems to be systems which support repeated display of data structure and program state visualization, respectively, with changes in view sufficient in both content and time to provide a viewer with the essence of how the data and program transform continuously throughout execution.

2. Related work and thesis contribution

2.1 Previous Work

This section reviews previous and current work that has directly influenced our research. Three types of work have influenced the present research, movies of algorithms in action, debuggers displaying data structures graphically, and algorithm animation systems, and we are going to discuss each of these in turn.

2.1.1 Algorithm Movies

From the start of the "Computer Graphics" era in the early 60's through the end of the 70's, graphics hardware was a scarce and expensive commodity. while the opportunity to use the hardware was small, its potential - especially for educators - was great.

The obvious solution to this problem in the mid 70's was to record the computer-generated images on film since the cost of on-line computer use is, of course, much higher than the print cost of a film.

Hopgood's movie on hashing algorithms in 1974 [Hop74] was the first movie whose ostensible purpose was to portray an algorithm. The movie contained three synthetic views which update discretely. The views were of a hash table where each entry showed not only whether or not the slot was taken, but also how many collisions were encountered when the element was inserted. A significant contribution of this movie was that it showed a data structure in a state that was too large to have been computed by hand simulation. The movie also displayed the effects of altering parameters to the algorithm (e.g. the size of the hash table), as well as how the algorithm performed with different types of input.

The basic limitation of film is, of course, that they do not allow viewers to experiment with the model displayed. Viewers must watch the film in the exact form in which it was produced, with the exact parameters, the exact data, almost always, even at the exact speed.

2.1.2 Graphical Displays of Data Structures

A number of systems [Bro88] that have been built, automatically produce a static graphical display of a program's data structures from the information available to the system debugger at runtime. These systems have the advantage that data structures in any arbitrary program can be viewed without altering the program in any way, but the disadvantages that the generic representation does not necessarily convey how the data is really used. Moreover, as discussed in chapter 1, a static display, even graphical, does not reveal how the algorithm is processing the data, nor does updating the algorithm in response to each change in the data.

Displaying data structures for debugging has different constraints, from creating views for algorithm animation. For debugging, the displays must be robust, and consequently more complex to implement than displays oriented to algorithm animation. Another problem unique to displays for debugging is that of naming: every data structure must identify as well as each of its components. In algorithm animation, as mentioned in chapter 1, simplifications can be made to the data format.

A system that provided graphical displays of arbitrary data structures for debugging was baskerville's GDBX [Bas85]. GDBX, integrated with the standard UNIX debugger, DBX, ran on a Sun Microsystems workstations. GDBX displayed a record in nested-box notation. Data structures were displayed in a standard Sun window with scroll and zooming capabilities. The user could pick-up a pointer field and drag it to any node on the screen. GDBX also animated variables by redisplaying the changed fields of the specified data structure without first erasing them after each step in the debugger. Modified fields were highlighted by blinking.

In general, graphical debuggers were not well-suited for algorithm animation because they could only monitor data, interesting displays, especially those that reflect interrelations among variables could not be supported. Synthetic views crucial to algorithm animation were not supported in most of the systems also.

2.1.3 Algorithm Animation Systems

In this section, we look at systems developed explicitly to illustrate algorithms. There are various well-known effort to develop algorithm systems, starting as early as in

the late sixties and early seventies. One of best-known and most successful is the BALSA system [Bro88]. Briefly, BALSA is a standalone system based on a kernel running on Macintoshes. Views for new algorithms are constructed in Pascal, using an animation library, algorithms are coded in high-level language such as Pascal and are annotated with markers called interesting events, indicating the phenomena that will be of interest when the program runs. When the program runs within the algorithm environment, all views on the screen (see Fig 2.1) are notified and each view updated itself.

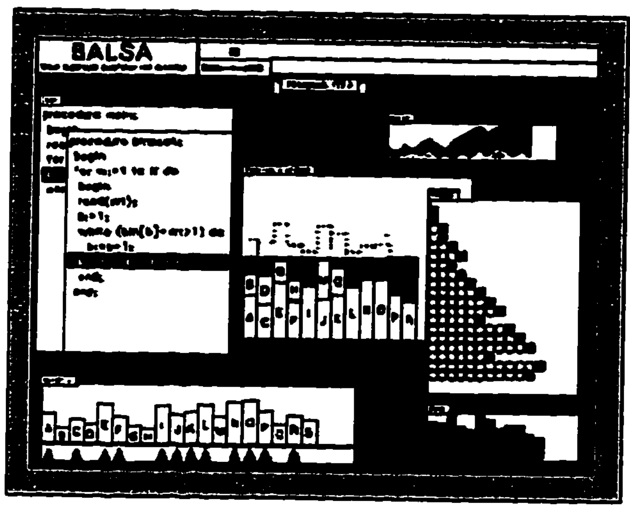


Figure 2.1: First-fit binpacking algorithm.

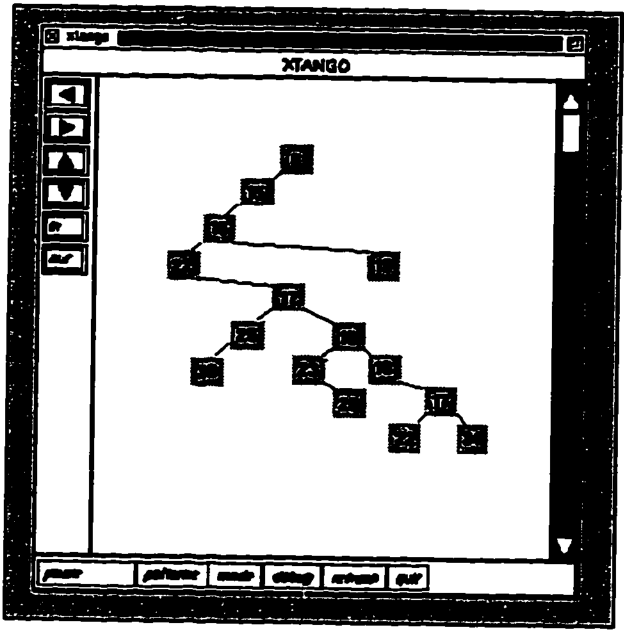


Figure 2.2: Frame from the XTANGO pairing heap animation.

John Stasko [Sta89] implemented a system called TANGO (short for Transition Based animation Generation). He pursues the ambitious goal of trying to define a theoretical framework for algorithm animation. A large part of this work centers around finding the right graphical building blocks. (see Figure 2.2)

During 1988, Marc H Brown and Marc Najork began developing ZEUS [Bro91], one of the most straightforward algorithm animation systems. In addition to animating algorithms from the domain of computational geometry, operating systems, hardware design, distributed spanning trees, and communication protocols, ZEUS is the framework for formEdits, a multi-view editor for building user interfaces.

ZEUS allows its user to run an algorithm and to observe it through several views. Algorithm and views communicate by passing events back and forth. ZEUS is noteworthy for its use of objects, strong-typing, parallelism, and graphical development of views and allowed the use of color and sound, previously uncharted areas in algorithm animation.

2.1.4 Systems Using 3D Graphics

There are three algorithm animation systems that have been used for developing 3D views, Pavane, Polka3D, and GASP.

In Pavane [CR92], the computational model is based on tuple-spaces and mappings between them. Entering tuples into the "animation tuple space" has the side-effect of updating the view. A small collection of 3D primitives are available (points, lines, polygons, circles, and spheres), and the only animation primitives are to change the positions of these objects.

Polka3D [CR92], like ZEUS, follows the BALSA model for animating algorithms. Algorithms communicate with views using "interesting events", and views drawn on the screen in response to the events. The graphics library is slimmer and more focused on algorithm animations (unlike zeus) meanwhile views are not interpreted.

The GASP [CR92] system is tuned for developing animations of computational geometry algorithms involving three (and two) dimensions. Because a primary goal of the system is to isolate the user from details of how the graphics is done, a user is limited to choosing from among a collection of animation effects supplied by the system. The

viewing choices are typically stored in a separate "style" file that is read by the system at runtime; thus, GASP provides rapid turnaround. However, it does not provide the flexibility to develop arbitrary view with arbitrary animation effects.

2.2 Discussion

In the previous section we gave an idea of the existing systems, ranging from algorithm movies to true algorithm animation systems, and then to those who support three-dimensional graphics. In this section, we are going to discuss some of the weaknesses and limitations of them.

We have noticed, so far, two delicate trade-offs:

The first trade-off is between algorithmic concepts and its related animations. In the TANGO system, for example, John Stasko does not try to find an educationally "best" representation of an algorithm, but he only just transforms any program fragment into an algorithm without weighting the instructional value of this program fragment. In fact, if we experiment with TANGO, we will never notice the relationship between the view containing the algorithm code and the data structure being used, contrary to that, a pop-up window will be displayed and an animation of graphical objects will take place.

The second trade-off is between ease-of-use and technical robustness. In the ZEUS system, only expert programmers could add or change algorithms and their associated annotations in order to create animation, for instance, one to one animation. Which leads to an extensive lack of interactivity between the end-user and the system interface.

The BALSA is one of the few system which successfully managed these trade-offs, meanwhile, system's developers did not take advantage of the graphic enhancements, such as straightforward animation tools and visualization techniques, for instance, three-dimension graphics. In general, researchers from all over the world still agree that BALSA is considered the starting point for any algorithm animation system.

We remind, that all these environments described above run on either Sun/UNIX workstations or Macintoshes, none of them has the capabilities to run on a PC / Windows based workstation, which are nowadays the most popular platforms and the widely used throughout the world.

2.3 Thesis Contribution

Our major contributions to the current research are the following. The first contribution is a methodology for drawing partially ordered sets in three-dimension. The second consist in developing an interactive model for adding and event-based graphical animation to algorithms, which has the characteristics of fully satisfying the trade-offs stated in the previous section. Other minor contributions are also introduced at the end of the current section.

2.3.1 Contribution 1: Three-dimensional Drawing of Partially Ordered Sets

In chapter 3 we will present a methodology for drawing partially ordered sets in 3D space. We describe two techniques: the uniform level decomposition and the barycentric level decomposition. We will then explain how 3D imagery can best enhance the viewing of the structure through navigation, and multiple-views.

We also motivate the need of minimizing crossing, maximizing the coupling between the components of the 3D structure and finally, reducing the density of the 3D representation. In order to accomplish these objectives, we will introduce the concept of metrics and measurements.

2.3.2 Contribution 2: An interactive model for algorithm animation

The issue of adding animation(s) to algorithm has never been addressed before. In chapter 4, we will see how we can shift the process of adding animation to algorithm from being one to one *(1-1)* to one to many *(1-N)*. We will also introduce our model, its components, the issue of real-time mapping, the notion of off-line animation, and of course the proof of its flexibility and interactivity.

What are the algorithms which are going to be animated ?

None of the previous systems have attempted to consider the class of algorithms on ordered sets. In our work, we're going to deal mainly with this category of algorithms and their complex data structure. We will focus on their syntax and their semantics, and see how we can generate their proper animations.

2.3.3 Systems contributions.

- In chapter 5, we will describe our straightforward graphic view, which has lots of advanced features such as basic zooming, object translation, camera positioning, object motion, and 3D navigation.
- As a growing concern, user interface is now very important criterion for acceptability of new systems, and for us, it's a must, to comply with user interface requirements, in order, to produce a highly interactive algorithm animation environment.
- Other minor contribution will be also presented all along the coming chapters.

2.4 Thesis disclaimers

In our system we are not going to consider the animation of algorithms on large graphs. In fact, experiences with the previous systems has shown that relatively small data associated with their algorithms are best understood than large data.

Another aspect we do not address here is how fast the animation should take place, or with what granularity. It is essential, however, that an algorithm animation system allow users to control the speed. Not surprisingly, it has been shown that information is lost if an animation is either too fast or too slow [Mod79]. The optimum speed depends on the viewer and the purpose of the animation. In practice, we have found that, as one would expect, viewers can get a high-level intuition for the dynamics of an algorithm when relatively fast-speed are used. To understand details of the behavior, however, relatively slow speeds are used. The absolute speed also depends on the view on the screen; complex view often require slower speeds to let the user digest all of the information on the screen. Displaying representations that are unfamiliar to viewers also require slow-speeds, at least until the displays are incorporated into the viewers visual vocabulary.

Another issue not addressed in this thesis, is the building of a straightforward algorithm interpreter which could bring more functionalities like backward execution, history of execution, and real-time code modification. Meanwhile, the current solution is suitable and adequate to comply with all requirements needed to accomplish an interactive algorithm animation environment.

3. Data Structure Visualization

3.1 Problem domain

Ordered sets have been widely described by mathematicians, computer scientists, and theoreticians (see [Riv84], [Riv87], [Tro92], [Riv94]). Ordered sets occur widely in computation, in scheduling, in sorting, in social choice, decision-making and even in geography.

For some years research on these themes has focused on combinatorial optimization and on "algorithms" involving ordered sets. Important advances have been made at practical, and at theoretical levels. There is little doubt that the modern mathematical theory of ordered sets owes much of its vitality to those recent developments.

Recently, we noticed that the attention is shifting from the usual optimization themes to data structures. Indeed, there is an emerging need for efficient representation of data structures and their best visualization, in order to gain a better understanding and enhance its readability.

3.1.1 Definition and Notation

We consider, an ordered set (*partially ordered set* or also *poset*) to be a pair (X,P) where X is a finite set of elements, and P a reflexive, antisymmetric, and transitive binary relation on X.

If we take $x, y \in X$ with $x \neq y$. We say x and y are comparable in P, and write $x \perp y$, when either x is in relation with y [denoted (x,y) is in P, or $x < y$]. In the example in Figure 3.1 (a) we consider an ordered set X = {1,2,3,4,6,12} and P = {(x,y) $\in$ X $\times$ X, where x is a divisor of y}. Here, the comparability relation < is divisor. On the other hand, x and y are incomparable in P, denoted $x \,||\, y$ in P, if neither $x < y$ in P nor $x > y$ in P (in Figure 3.1 (a), 6 || 4). We say x is *covered* by y in P (also y *covers* x in and (x,y) is a *cover* in P), denoted by $x \ll y$ in P, when $x < y$ in P, and there is no element $z \in X$ for which $x < z$ in P and $z < y$ in P (in Figure 3.1 (a) , 2 << 4). Graphically, an element of X is represented by a circle in two-dimension, we will call it a *vertex* and in 3D by, a labeled -

or not labeled - colored small cube. The labels does not necessarily reflect the value associated with the vertex. A cover is represented in both 2D and 3D by an edge.

Example: *Let* $X = \{1,2,3,4,6,12\}$ *and*

$P = \{(x,y) \in X \times X$, *where x is a divisor of y*$\}$

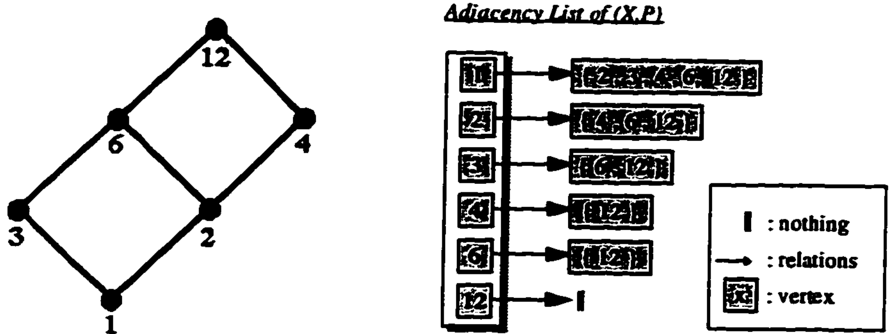


Figure 3.1 (a) Ordered Set *(X,P)* Figure 3.1 (b) Internal representation of *(X,P)*

3.2 Use of three-dimension

This section describes the different uses of three-dimensional interactive graphics for algorithm animation [Bro94]. Three-dimensional interactive graphics provide another level of expressiveness to the animation, akin to the way that smooth transitions, color, and sound have increased the level of expressiveness in the past.

Specifically, in [Bro94] there are three distinct uses of 3D:

1. *Expressing fundamental information about structures that are inherently 2D*

Consider how one might display the values of an *n-by-n*, two-dimensional matrix of positive numbers. An obvious 3D display is to draw sticks at each cell of an *n-by-n* grid, where the height of each stick is proportional to the value of the corresponding element. Of course, there are other techniques for displaying the matrix without using 3D graphics, such as displaying a number in each cell or modifying the color, shape, or size of each cell according to the value of the corresponding element in the matrix. However, showing sticks of varying heights seems to be an effective technique, perhaps because it allows direct visual comparison of the elements.

2. *Uniting multiple views of an object*

Finding a single view of a complex object that reveals all of its features can be difficult, but not impossible. Therefore, presenting multiple views of that object is a helpful visualization technique. However, it can be difficult for the user to understand the relationship between the multiple views. A carefully crafted 3D view can incorporate multiple 2D views into a single image, thereby helping the user to see how the views are related, by means of navigation, translation, rotation, and zooming.

3. *Capturing a history of a two-dimensional view*

Often, a visualization of a program's entire execution history can be just as helpful for understanding a program's behavior as an animation of the current state of the program. When running programs on small amounts of data, a history often gives the user a context of how the algorithm has progressed each time the state has changed. When running programs on large amounts of data, a history often exposes patterns that are not otherwise observable.

In any event, we are not proposing to use 3D for showing objects that are intrinsically three-dimensional, such as the convex hull of points in 3D space. We are also not advocating the use of 3D for enhancing the beauty of a picture that is easily shown in 2D. Our use of the 3D has rather the purpose of uniting mulitple views of an object (2nd category) in order to increase the quantity of information conveyed in a graphical display.

3.3 Ordered sets: From 2D to 3D

In this section, we will introduce and describe two methods for generating three-dimensional representation of orders; uniform level decomposition and barycentric level decomposition. They have both proven to be relatively successful when applied to our plethora of samples. The barycentric-based technique, hence, preserve the semantic meaning of the structure in terms of grouping.

Important: We emphasize that it is possible to incorporate new methods for representing orders in three-dimension, since data structure visualization and algorithm animation are completely independent from each other.

Before going into details, we should present our 3D coordinate system which will be the reference and the basis for all 3D drawing and mapping.

3.3.1 Coordinate System

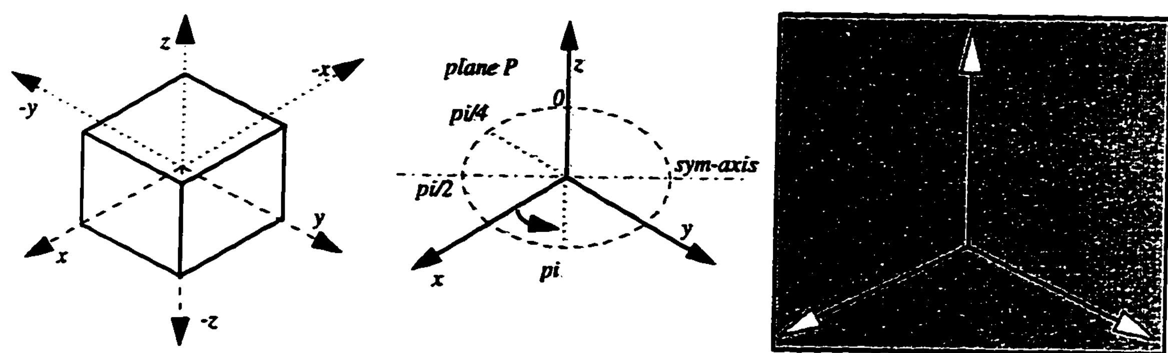


Figure 3.2: 3D Coordinate system.

The reference point (0,0,0) is the intersection of the three axis -xx, -yy, and -zz. All graphical objects can be drawn on the 3D view by two means:

They can be represented by their Cartesian *x, y, z* coordinate or their polar coordinate *(c(z,r),theta)*, where *c* is a circle whose center is the point *(0,0,z)* and *r* its radius, *theta* denotes the polar angle where the vertex is placed.

3.3.2 Method one: Uniform Level Decomposition

We consider the following level decomposition algorithm:

Data structures:

Let (X,P) be an ordered set and C a linked list representing the set of covering edges.

Let AL(1..i) be an adjacency list

Algorithm: Level_Decomposition (X,P) ==> AL(1..i)

i = 1

AL(i) = max (X,P) /* find all elements which does not have an upper cover */

For all nodes x in AL(i)

```
            For all edges (x,j) containing the element x
                    remove (C, edge (x,j))
                    If j is maximal then AL(i+1) = AL(i+1) + { j }
            Loop
            Inc (i)
    Loop
End.
```

Analysis of the complexity:

Based on the internal representation of an ordered set and the set of covering edges. The first *Max* operation looks for elements who are covered by nil, and thus provides the first level $AL(1)$, this operation runs in $O(n)$.

Then, for all elements n_i in a level i, we remove all edges (n, k) containing these n_i elements in $O(m_i)$ times where m_i is the number of edges between the levels $AL(i)$ and $AL(i+1)$.

But since n_i elements in a level i are distinct from the n_j elements in a level j and also the m_i covering edges related to the n_i elements are distinct from the m_j covering edges related to the n_j elements, it follows that, the total number of removal operations is of order $n_1 + n_2 \times m_2 + .. + n_i \times m_i < n\,(m_1 + m_2 + .. + m_i) < n \times m$.

Checking whether an element k is maximal is done in $O(n)$ times where n is the total number of elements, but since this operation is done at most m times where m is the total number of covering egdes, then the time complexity of the algorithm is $O(n \times m)$.

Important: The level decomposition algorithm is called only when there is a change in the 2D view, such as, adding or removing, vertices and edges. This process is independent from the 3D drawing algorithms, and not invoked automatically by these 3D drawing algorithms, meanwhile it provides the basic data, for instance the level decomposition, in order, to construct the 3D structure.

Layout of the uniform level decomposition algorithm

Algorithm: Uniform_LD (X,P)

For each AL(i)

- Draw a virtual 3D circle C on the plane xy, such that:

C→ center = (0,0,z) { where z is proportional to the index i of the current level},
and C→ radius = f (|AL(i)|).

- Distribute uniformly the elements in AL(i) as follows:

Let gap = 2π / |AL(i)|.
For each element E in AL(i)

Draw a vertex on the circle at location: zero + gap* (pos(E)-1)
{ where pos(E) is the horizontal position of the element E in the level i of the 2D representation, i.e.: The first vertex at location zero is the left-most vertex in the 2D representation }

End for.

End for.

End.

Analysis of the complexity

We distribute uniformly each elements of the order, level by level, on virtual circles, so it follows that the algorithm runs in $O(n)$.

We notice so far that the uniform level decomposition algorithm.

- Produces a relatively acceptable 3D representation of the 2D graph.
- Avoids vertex overlapping in 3D.
- Does not avoid edge crossing (see next section for how to minimize edge crossing in a plain 2D view).

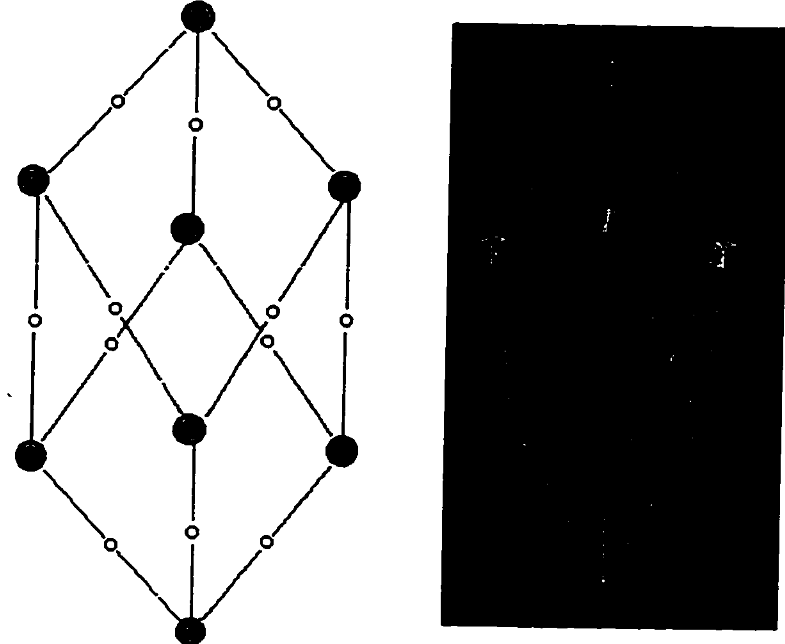


Figure 3.3[1]: Left: Snapshot of the hypercube (random labels) taken from the input view
Right: Snapshot of the 3D representation of the hypercube taken from the 3D view.

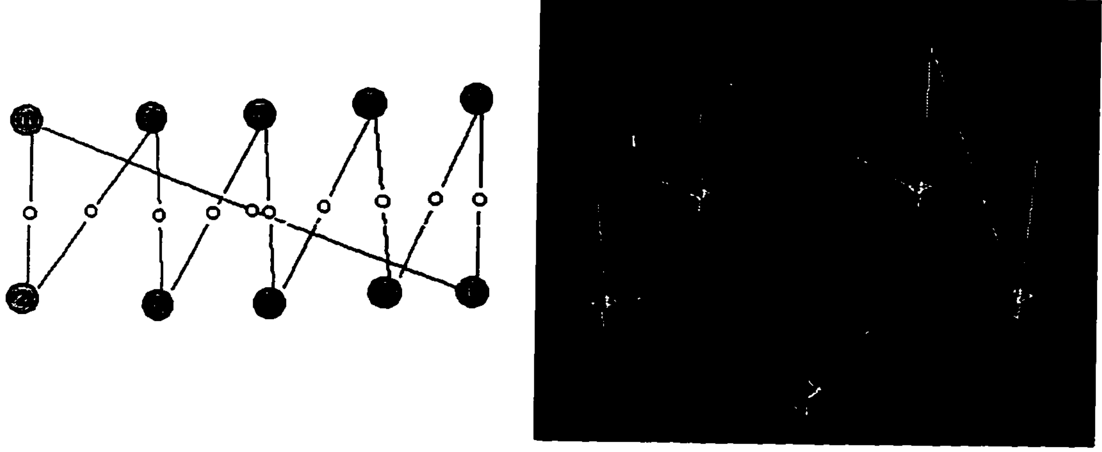


Figure 3.4: Another 3D representation of an order using Uniform_LD.

[1] Colors associated with the 3D vertices, denote the level of the vertex.
All views are moved up by $+\pi/4$ from the xy plane.

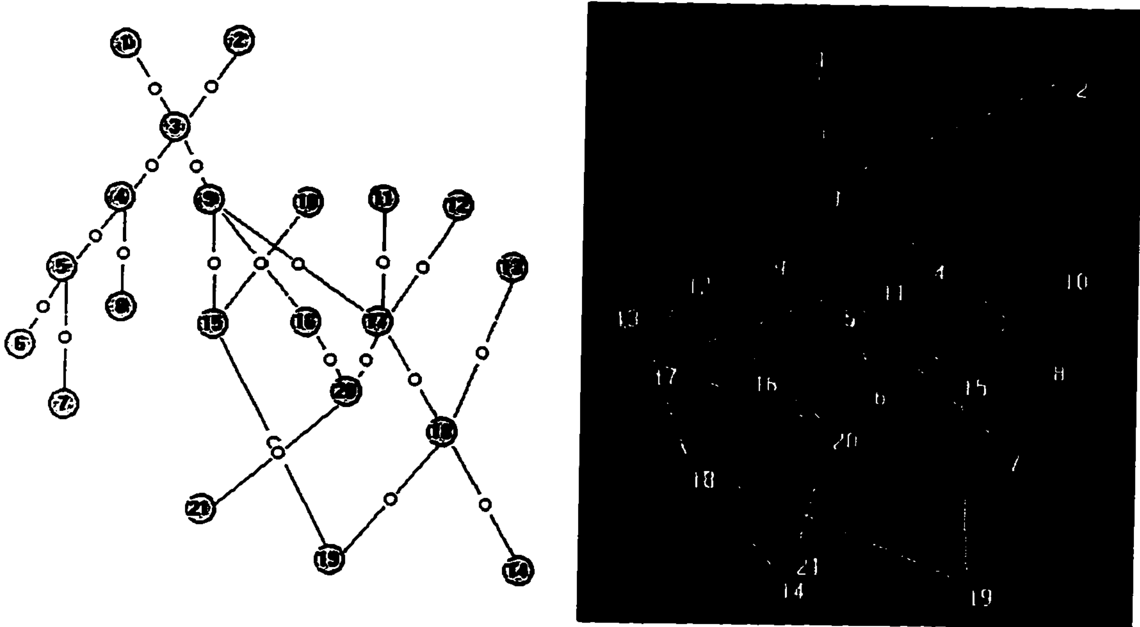


Figure 3.5: A more complex structure represented in 3D by the Uniform_LD method.

3.3.2 Method two: Barycentric Level Decomposition

The main idea conveyed by the use the barycentre concept is the grouping of elements which have common properties, around the barycentre of their uppercovers.

In our case, this method is somehow similar to the uniform level decomposition, but differs in the way elements are drawn on the virtual circles. In fact, we will show in the layout of the algorithm how the semantic grouping is relatively preserved through the use of the barycentre.

<u>Layout of our algorithm</u>

Data structures:

Let (X,P) be an ordered set, and C a linked list representing the set of covering edges.

Let AL(1..i) be an adjacency list corresponding to the level decomposition.

Let UCx, UCy be two variables: set containing the upper covers of a vertex.

Let LowerSet be a variable: set of elements which share exactly the same UC.

Let CP be a 3D point.

Algorithm: Barycentric_LD (X,P)

Step0: Call Level Decomposition Algorithm /* Initialization step */

Step1: Level 1:

- Draw a virtual 3D circle C on the plane xy, such that:

C → center = (0,0,z) *{ where z is proportional to the index of the current level and the screen resolution }*

C→ radius = f (| AL(1) |)

- Distribute uniformly the elements of AL(1) on C as follows:

Let gap = 2π / | AL(1) |

For each element E in AL(1)

Draw a vertex on C at location: zero + gap* (pos(E)-1)

{ where pos(E) is the ordinal position of the element E in AL(1) }

End for.

Step2: Level 2 and more

For each AL(i) where i > 1

Pick up a non marked element x in A(i)

LowerSet = { x }

Let UCx = UpperCovers (C,x)

- Look for elements in AL(i) who have the same UpperCovers as x

For each non marked element y in AL(i) where y ≠ x

UCy = UpperCovers (C,y)

if UCy equal to UCx then LowerSet = LowerSet + { y }

End for

Mark all elements in LowerSet

if not null(UCx) then CP = Barycentre (UCx) Else CP = (0,0,z)

{ Where z in both cases (UCx null or not null) is a function of the index of the current level and the screen resolution }

Step3: Drawing of LowerSet

- Draw a virtual circle C on the plane xy, such that:

C → center = CP and C→ radius = f (|LowerSet|) { f is a linear function }

- Distribute uniformly the elements of LowerSet on C as follows

Let gap = 2π / |LowerSet|

For each element E in LowerSet

Draw a vertex on the circle at location: zero + gap* (pos(E)-1)

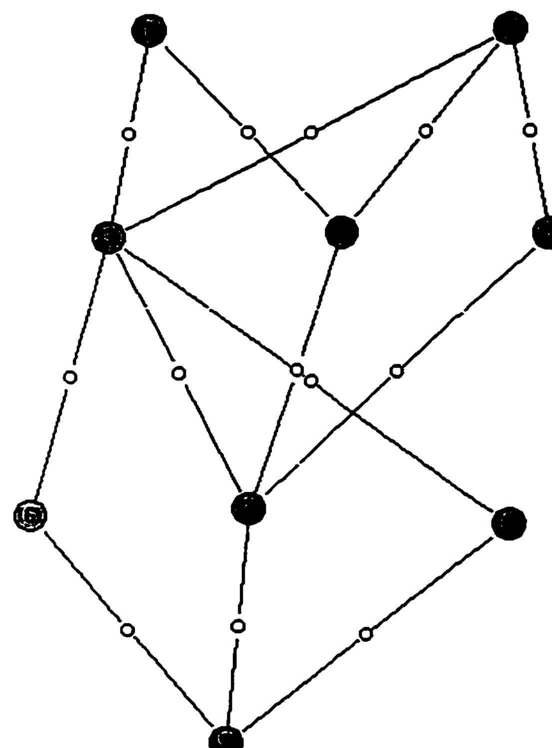
{ where pos(E) is the ordinal position of E in LowerSet }

End for

End for *{ levels }*.

Example of execution:

We are going through this example to describe all the steps of the algorithm.



Let be the following example:

X = { 1,2,3,4,5,6,7,8,9 } such that:

{ 9 } is below { 6 , 7 , 8 },

{ 6 } is below { 3 },

{ 7 } is below { 3 , 4 , 5 },

{ 8 } is below { 3 },

{ 3 } is below { 1 , 2 },

{ 4 } is below { 1 , 2 },

{ 5 } is below { 2 },

{ 1 } is below nothing,

{ 2 } is below nothing, and

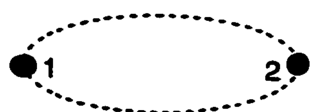
Number of Levels = 4.

We already have the level decompotion of the order as follows:

AL(1..4) = { { 1 , 2 } , { 3 , 4 , 5 } , { 6 , 7 , 8 } , { 9 } }

Step 1: Drawing elements of the first level.

Elements 1 and 2 are on the first level, so they will be distributed uniformly on a circle C whose center is (0,0,z) and radius = f(2).



Step 2: Level 2

Elements in level 2 are: 3,4, and 5.

LowerSet = { 3 }.

UC (3) = { 1 , 2 }.

UC (4) = { 1 , 2 }.

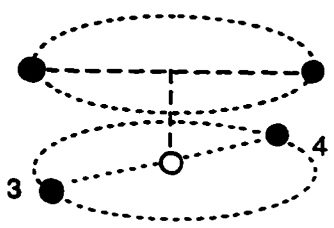
UC (5) = { 2 }.

So LowerSet = { 3 , 4 } and CP = Barycentre ({1, 2}).

3 and 4 are then marked.

Step 3: Drawing of LowerSet.

3 and 4 are put on a circle whose center is CP and radius = f(2).



Step 2: Level 2: Element 5.

UC (5) = { 2 }.

3 and 4 are already marked.

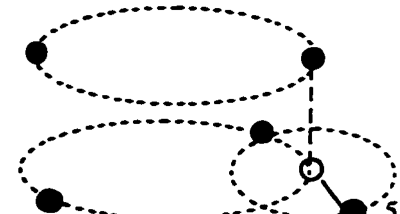
LowerSet = { 5 } and CP = Barycentre ({3}) = 3.

5 is then marked.

Step 3: Drawing of LowerSet

5 is put on a circle whose center is CP

and radius = f(1).



Step 2: Level 3:

Elements in level 3 are 6, 7, and 8.

LowerSet = { 6 }.

UC (6) = { 3 }.

UC (7) = { 3 , 5 }.

UC (8) = { 3 }.

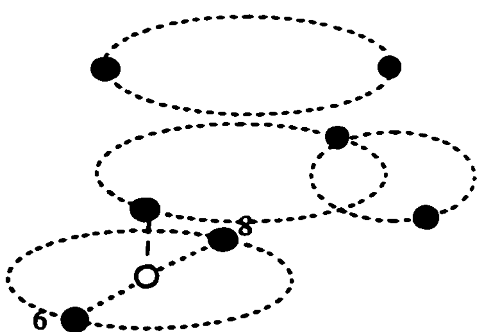
So LowerSet = { 6 , 8 } and CP = Barycentre ({ 3 }).

6 and 8 are then marked

Step 3: Drawing of LowerSet

6 and 8 are put on a circle whose center is CP

and radius = f(2).



Step 2: Level 3: Element 7.

UC (7) = { 3 , 4 , 5 }.

6 and 8 are already marked.

LowerSet = { 7 }.

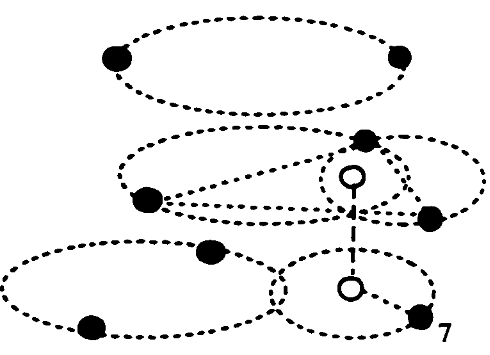
CP = Barycentre ({ 3, 4, 5}).

7 is then marked.

Step 3: Drawing of LowerSet

7 is put on a circle whose center is CP

and radius = f(1).



Step 2: Level 4: Element 9.

LowerSet = { 9 }.

UC (9) = { 6 , 7 , 8 }.

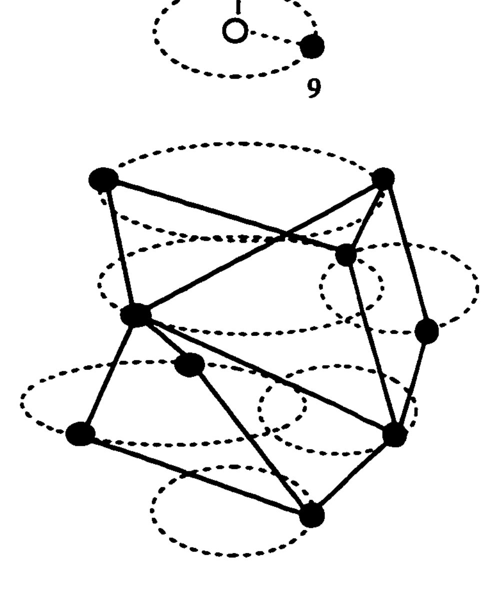
So LowerSet still = { 9 }.

CP = Barycentre ({ 6 , 7 , 8 }).

9 is then marked.

Step 3: Drawing of LowerSet

9 is put on a circle whose center is CP and radius = f(1).



A projection of the final 3D drawing is shown in front:

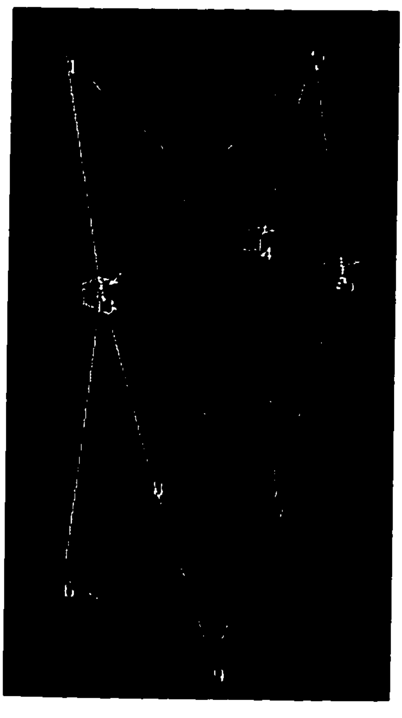


Figure 3.6: A snapshot of the Final 3D drawing taken from the 3D View of IAA/3D:

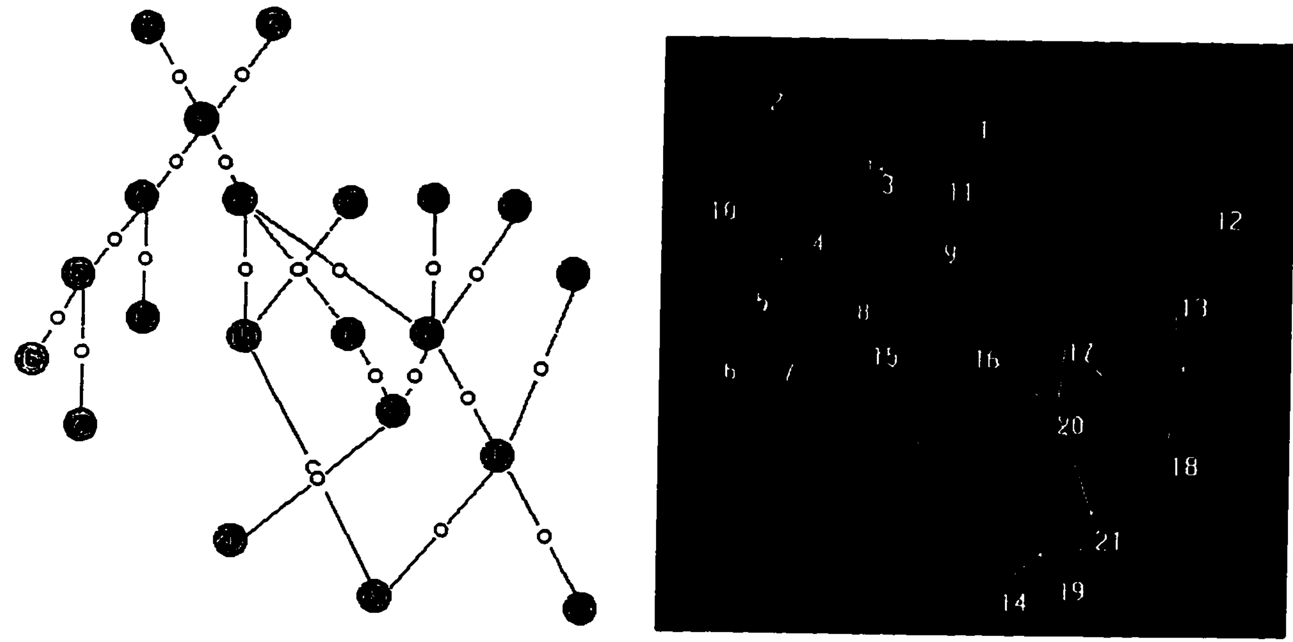


Figure 3.7: The same structure as in Figure 3.5 drawn using the barycentric method.

Analysis of the complexity

In the initialization step, we generate in $O(n \times m)$ time, the level decomposition of a given ordered set, such that: $AL = \{ AL_1, AL_2, .., AL_r \}$, where r is the number of levels.

For each level i such that $1 < i \leq r$, we have n_i vertices and m_i covering edges. In order to place a node on a virtual circle, we need to look at all its covering edge to get the uppercovers and then compare them with those of the neighboring vertices, so for all the nodes of a level i we will need $O(n_i \times m_i)$ operations.

But, since all the n_i vertices and the m_i covering edges of a level i are distinct from the n_j vertices and the m_j covering edges of a level j, so we will have for all the levels: $n_1 \times m_1 + n_2 \times m_2 + .. + n_r \times m_r < n\,(m_1 + m_2 + .. + m_r) < n \times m$ operations to place all the vertices. We conclude then, that the time complexity of the algorithm is $O(n \times m)$.

It follows that this algorithm:

- Produces a relatively acceptable drawing which preserves the grouping of elements according to their covers, in 3D.
- Also avoids vertex overlapping in 3D.

Finally, if we compare the 3D drawings in figure 3.5 and figure 3.7, we notice clearly the differences between the output of the uniform level decomposition and the barycentric method. Figure 3.7 best represents the graph, indeed, if we look at the chain 1,3,4,5,6, we see clearly that it is better represented in 3D and that, all covering relations; 4>8, 5>6 and 5>7 are also well perceived.

We remind our readers that the process of data structure visualization is not the fundamental constituent of our thesis, and should always be considered as a step toward our main objective which is an interactive algorithm animation. In fact, we do not tend to find the best solution or the method for 3D drawing. Meanwhile, we have to address the issue in a comprehensive manner, for testing purpose, to ensure that the semantics of the data structure is preserved and also in order to produce a relatively acceptable drawing.

3.4 Concern for Readability and Clarity

In this section we are going to focus on some problems generated all along the process of the 3D drawing of an order, and how we can deal with these problems by means of geometrical transformation and view rotation.

For our purpose, we consider a 3D structure readable if the information conveyed by the 2D structure is totally preserved, such as, the relationship between vertices and their corresponding levels, the homogenous representation of the vertices, their labels and the edges between them, as well as, minimum crossing.

3.4.1 Dummy nodes

We define dummy nodes as being the set of one or many virtual nodes between two nodes who does not reside on consecutive levels. A virtual node is treated by the drawing methods the same way as normal nodes but is not visualized on the screen neither taken in consideration by algorithms during running and animation.

let's consider the next figure 3.8. On the left we have the original order drawn in 2D. The image next is a 3D representation of the order using the uniform level

decomposition. The image on the right is also a 3D representation of the order using the same method, augmented with a dummy node (see broken edge).

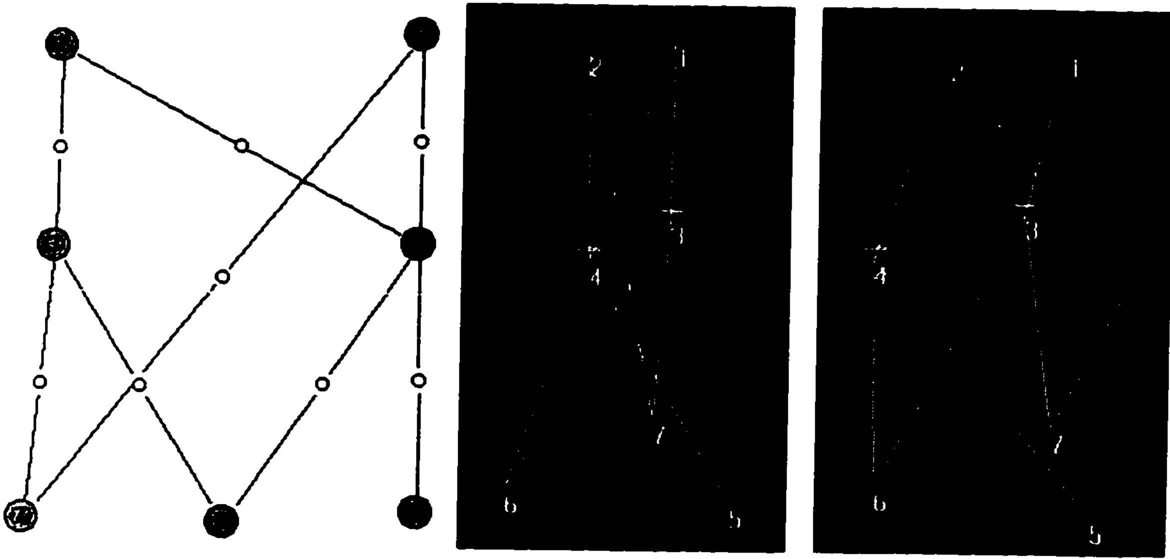


Figure 3.8: 3D representation of an order with dummy node (left) and without (middle).

We clearly see in the figure 3.8, that the left-most picture is more readable that the second, in fact we avoid two 3D real crossings between the edge 2-7 and the edge 1-4, and the edge 2-7 and the edge 3-6 by inserting an invisible dummy node. We emphasize that the resulting drawing does not necessarly use straight lines to represent a covering edge between two original vertices. In the next paragraph, we give a description of the algorithm.

Algorithm for adding dummy nodes

For each edge, we determine how many necessary dummy nodes to add. Then, we add these dummy nodes one by one. For each one, we perfom computations in order to decide whether to keep it in the drawing or not.

Deciding whether to keep or to remove a dummy node depends on two conditions: First, adding a dummy should not increase density of the structure, in other words, a dummy node should not be too close to an existing node. Second, a dummy node should always be within an acceptable range of its upper cover and lower cover.

Layout of the algorithm

Algorithm: Dummy_Node *(X,P)*

For all edges (x,y), such that:

Level (x) and Level (y) are not consecutive, and x, y are not dummy nodes.

For i = Level (x) + 1 .. Level (y) - 1

- Add a dummy node d on level i, so that we will have two new edges *xd* and *dy*.
- Let *e* be the original edge between *x* and *y*.
- Find a vertex $z \in \{x, y, d\}$ such that: *z* is the nearest vertex (including dummy nodes) to the edge *e* within the range of the edge *e*.
- We compute the distance between *z* and the edge *e* and denote it by *ez*.
- Let *iz = min (distance (z, xd) , distance (z, dy)).*
- **If *iz > ez* then keep *d* else remove *d* from the drawing. *{ First condition }***
- **If *angle (y,d,x)* > π / 2 then keep *d* else remove *d*. *{ Second condition }***

End for.

End for.

The next figure 3.9 gives an example of how to decide whether to keep or to remove a dummy node.

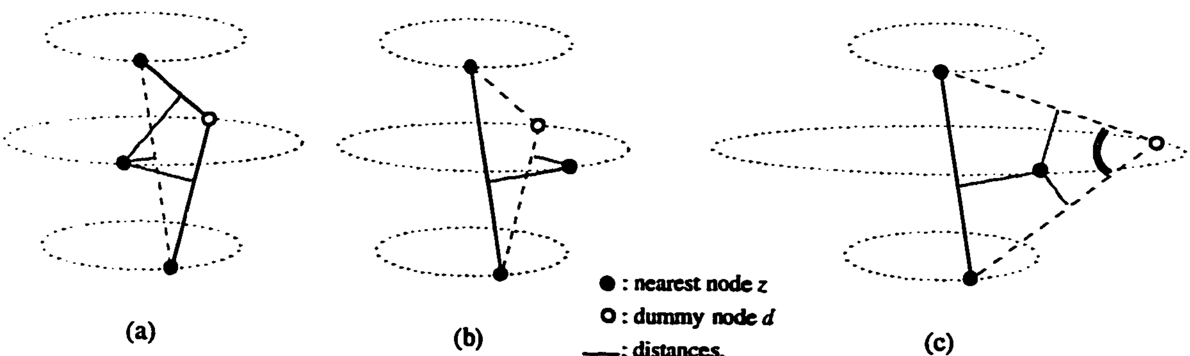


Figure 3.9: Decision making related to keeping or removing dummy nodes.

In figure (a), we keep the dummy node *d*, because it satisfies the two conditions. In figure (b), meanwhile, we remove the dummy node *d* because the first conditions is not satisfied (*iz* not higher than *ez*). In figure (c), we remove the dummy node *d* because the second condition is not satisfied.

Analysis of the complexity

First, we look for all edges whose nodes are not on consecutive levels. Then, for each edge we determine the number of dummy nodes to add between these levels, which is of $O(n^2)$. For each dummy node we look for the nearest vertex v to it, then compare the distances between, on one hand v and the newly added edges, and on the other hand v and the original edge, in order to decide whether to keep it or not, this process is done in $O(n)$, so the whole algorithm runs in $O(n^3)$.

3.4.2 Relative and absolute z-coordinate

By verifying on some samples, especially those whose height is elevated, we found that this feature could enhance the readability of the structure being drawn in three-dimension. It consists of: whether to keep the z coordinate as it was in the 2D representation (relative z), or bringing all nodes who reside on the same level l, on a same z-coordinate, which is calculated linearly based on the index of the level l. In this case, we call it the absolute z. The next figure shows the difference between relative z-coordinate and absolute z-coordinate.

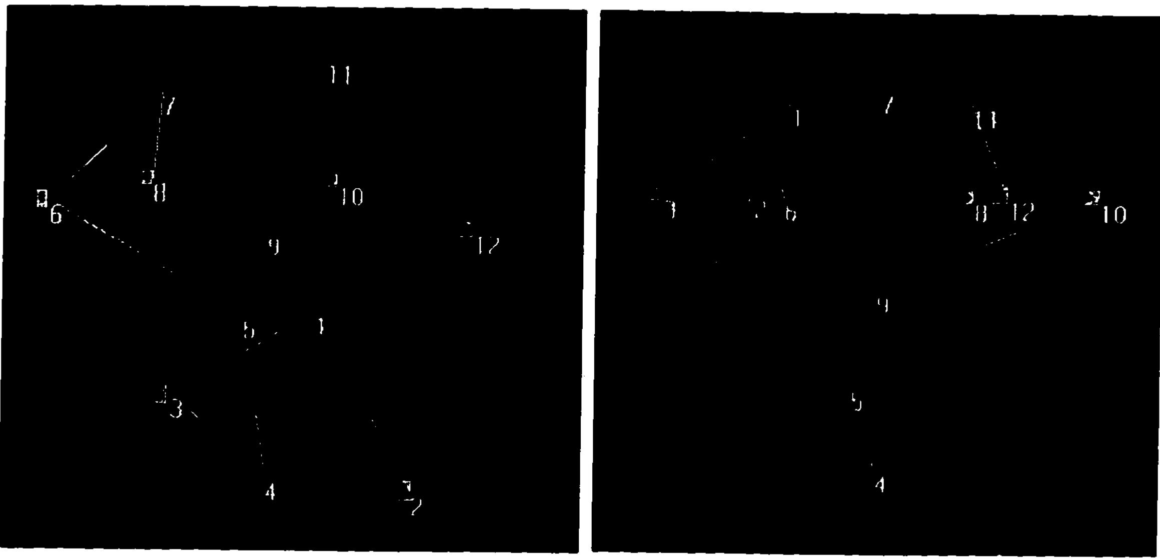


Figure 3.10: Two snapshots showing absolute z-coord (right) and relative z-coord. (left).

3.4.3 Level rotation and level symmetries

Sometimes, the distribution of the elements on the virtual circles are not appropriate and may generates some crossings or vertex overlapping in the 3D view. In order to avoid this, we provide two mechanisms for changing the disposition of the vertices on a level while keeping the node's uniform distribution.

First, we have level rotation which consists of whether to turn the virtual circle to the left or to right by a fixed angle of rotation.

Second, perform modifications on the layout of the virtual circle by means of symmetries according to the sym-axis (see figure 3.2).

These two features are far from being automated, and only the perception of the user can decide whether it is appropriate using them or not. We remind that it is possible to change the shape of the structure during both, the phase of the data structure visualization (3D mapping), as well as, during the algorithm visualization (animation and navigation). The next figure presents some examples of level rotation and level symmetries.

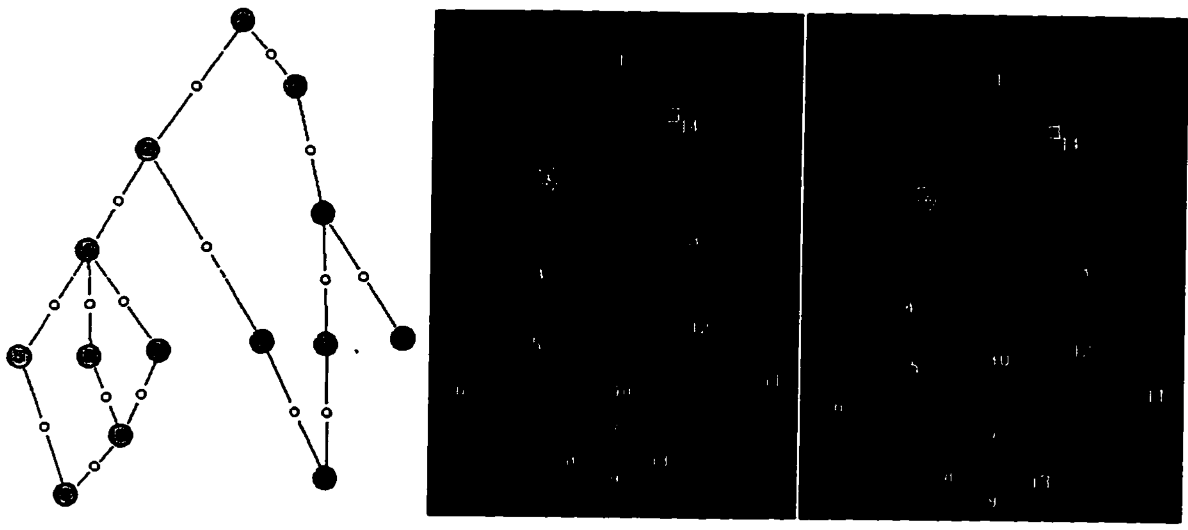


Figure 3.11: Level symmetries.

In this figure we have an order (left) and its 3D drawing using the uniform level decomposition method (middle). The right image shows the result of applying level symmetries to the third level which is composed of the elements 3,4,10.

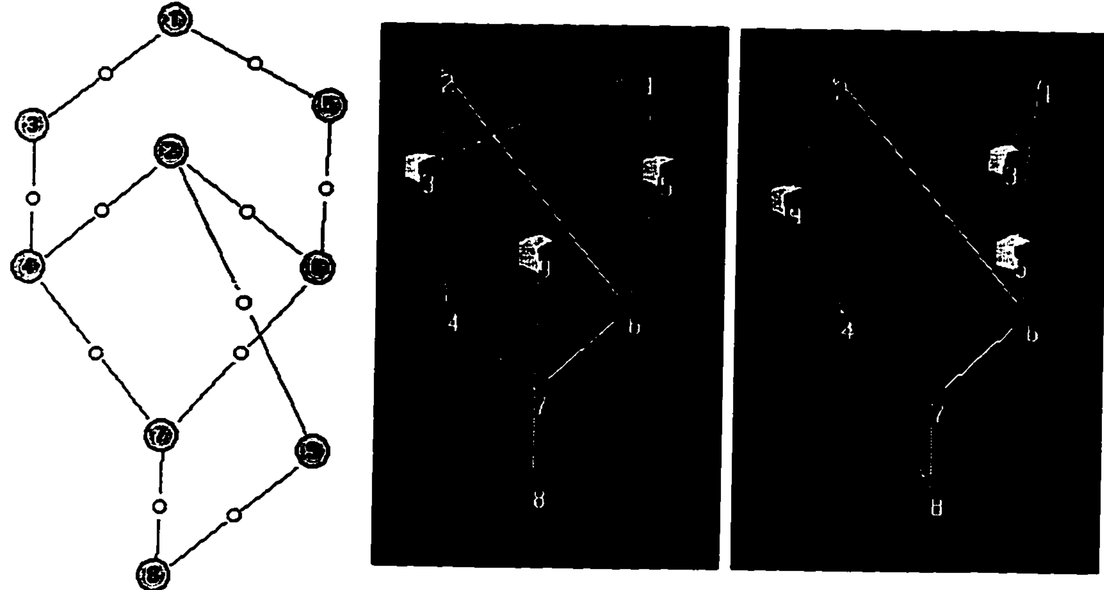


Figure 3.12: Level rotation.

However, we illustrate in the figure 3.12, the level rotation. We see clearly how rotation of level 2 (3,5,9) in the clockwise fashion, enhances the readability and minimizes crossing.

We have presented, so far, three techniques, for enhancing readability of our 3D drawing; the adding of dummy nodes, ways for computing the z-coordinate of 3D vertices, and level rotation and symmetries. Now, we are going to consider the issue of viewing the 3D representation as being a projection on the screen, and cope with the problems that may be encountered during the viewing process.

3.5 Use of Metrics

The question is: If we have a structure being drawn in 3D, how can we find the best 2D view considered as being the projection on the user screen of the 3D view ? The word best, means taking in consideration many aspects of the view, such as aesthetic criteria for visisblility and readilbility.

A lot of work has been done in this direction (see [Tun94], [Sea92], [DETT93]). In our case, we consider 3 concerns dealing with the clarity of the 2D view[2] : edge crossing, nodes coupling and nodes density, and we are going, all along this section to consider them as being our fundamental criteria.

3.5.1 Criteria

Edge crossing

...A common and traditional issue for all types of drawing. All researches tend to minimize the edge-crossings of a graph. We are going to provide a technique for searching and retrieving the best 2D view angle which has the least crossings between edges.

Coupling

The coupling of two nodes is the degree of correlation between their location in a drawing. In our case, a highly-coupled structure is a structure whose nodes, which are inter-related, are geographically located in the same region compared to the nodes which are not related, in contrary we say a low-coupled structure, when two related nodes are dispersed in space. Our objective, in this case, is to find the best 2D view which maximizes the coupling of the structure, because this usually preserves the readability of the poset.

Density

While density denotes the degree of compactness of the structure, regarding whether the nodes are related or not. A high density structure, is a structure where all nodes are located in a tiny regional space, while a low-density structure, is a structure where all nodes are well dispersed and distributed in space. Our objective tend to find the 2D view which minimizes the density.

---

[2] We are going to consider a 2D view as being a projection of the current 3D view on a Plane P (see Figure 3.2), according to a pre-defined angle.

3.5.2 2D Views

A 2D view is a projection on the user screen of the current 3D view, we can have as much 2D view as we want, and all projections are to be made on a plane defined by the user screen and a rotation angle.

In order to preserve the semantics or our structure, mainly the covering relation between any two nodes, we are going to consider only the projection according to the *xz* or *yz* plane, and discard those on the *xy* plane.

It is fundamental also to remind that, even if the navigation panel allows all directions, object rotation and camera rotation, we are going to restrict these features to only horizontal object rotation and camera rotation, in order to preserve, the semantics of the 3D ordered set.

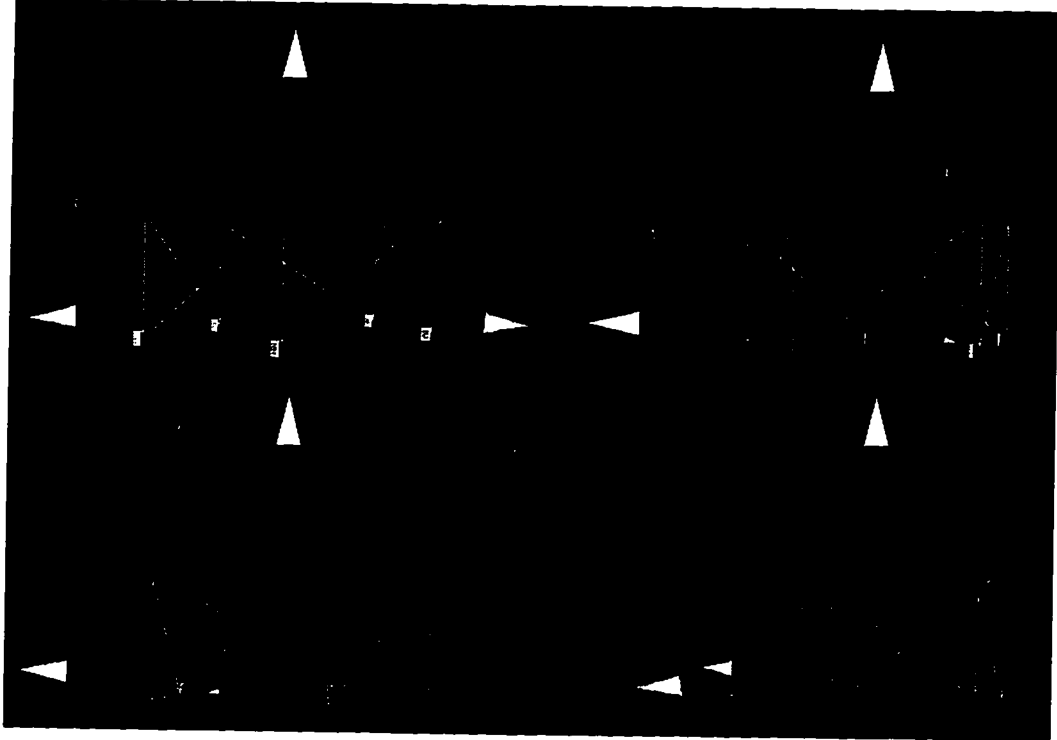


Figure 3.13: Three projections of a 3D structure (top-right, and bottom)

3.5.3 Metrics formulation

The word metric denotes the process of quantifying a well-defined criteria by means of conjunction of mathematical formula.

In this paragraph, we are going to investigate formally the means of these criteria by mathematical formulation.

- Let $Cross(e_i, e_j)$ be a function which returns a positive value (1) if e_i and e_j cross with each other in the 2D view or null else. We define E as follows:

  $$\text{E (edge-crossing)} = 1 - \frac{\sum Cross(e_i, e_j)}{m}$$; where m is the number of edges.

  We say a 2D view has a minimal number of edge-crossings if $E \approx 1$.

- In order to compute the density, we should first consider a 2D plane grid, whose height and width are equal to $\lceil\sqrt{n}\rceil$. A grid divides the plane into at most n cells, let *cell(x,y)* denotes the number of vertices in the cell at location (x,y), and let μ the average number of vertices per cell. It follows that:

  $$\text{D (density)} = 1 - \frac{\sum (Cell(x,y) - \mu)^2}{n};$$

  D is the standard deviation of the number of cells being occupied.

  A density is considered to be low if $D \approx 1$.

- For all edges, let *edge(u,l)* be the distance between the vertices u and l who are linked together by an edge. Let *dist(u,l)* be the distance between any two nodes (linked or not linked). Let m be the number of edges and n the number of nodes. It follows that:

  $$\text{C (coupling)} = 1 - \frac{\frac{\sum edge(u,l)^2}{m}}{\frac{\sum dist(u,l)^2}{n(n-1)/2}};$$

C is the average square distance of the linked nodes compared to the average square distance of two any nodes. (this metric was inspired from the theory of mechanics which

states that the energy of a structure is strongly related to the degree of coupling of its components). A high-coupled structure is a structure whose $C \approx 1$.

3.5.4 Measurements

In this section, we are going to present our methodology for finding the best 2D view based on the pre-defined criteria and their associated metrics, and also on the user desires, expressed by its choice concerning the number of projections.

3.5.4.1 Methodology

Our methodology is based on statistics and probabilities. A layout of it is described below:

1. First, we consider the set of our samples, which is a - user defined - fixed number of projections. (we note that the projection angle is generated randomly).
2. Second, we gather the necessary data, based on our metrics by means of graphical measurement.
3. Third, we analyze the measurements.
4. Finally, we choose the best 2D view.

Step1: Samples

Let *np* be the number of desired projection(s) defined by the user, we then have $2\pi / np$ projections of the 3D on the plane P (see Figure 3.2) formed by the *sym-axis* and the *z-axis*. Examples of projections are show in figure 3.14.

Step2: Information gathering

We apply our metrics on all projections and store them in a repository for further analysis. All the values gathered are normalized and between 0 and 1.

Step3: Synthesis

Let *metric(p,m)* be a matrix whose rows are projections and columns are metrics; entries then, denote the value gathered in step2 for the metric $m \in \{E, C, D\}$ and a projection p. We first compute the mean value μ_m and the variance $\sigma_m{}^2$ for all projections, per metric.

as follows: $\mu_m = \frac{\sum_p metric(p,m)}{np}$, $\sigma_m{}^2 = \frac{\sum_p (metric(p,m) - \mu_m))^2}{np}$

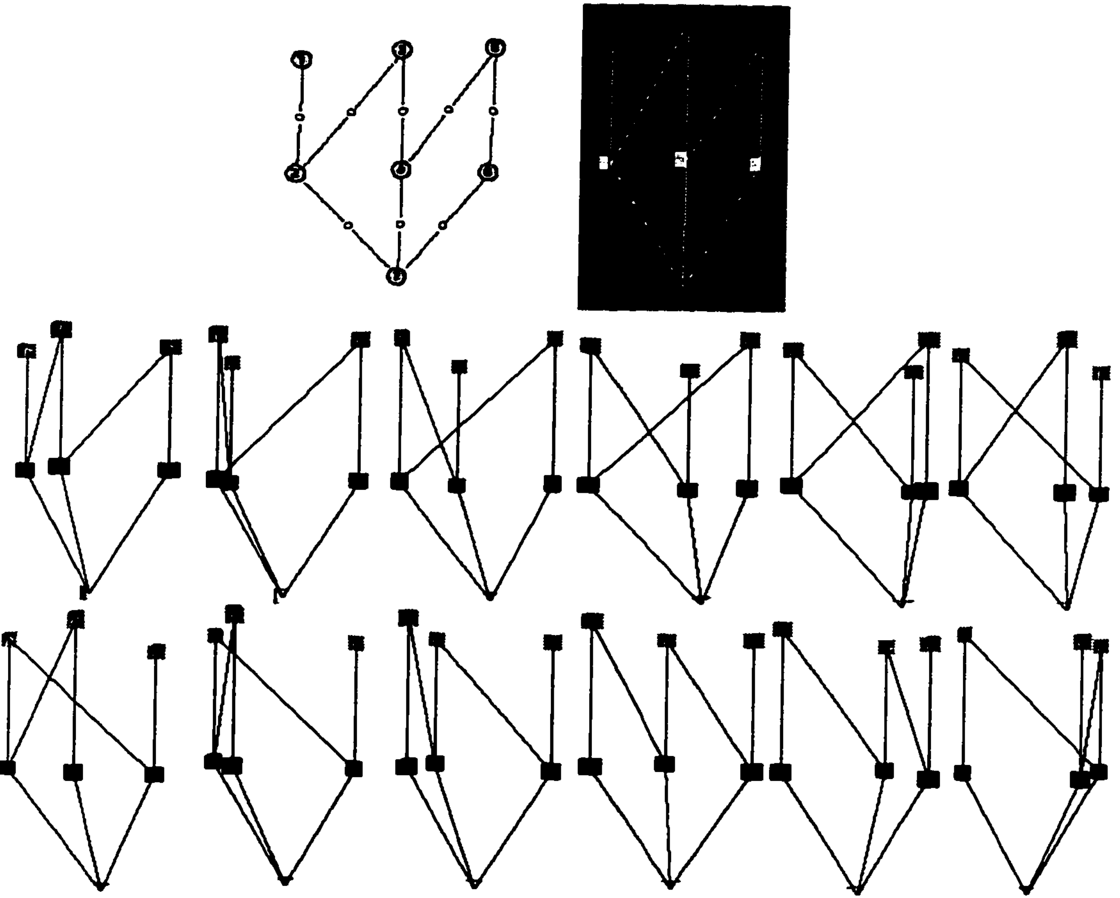
Then we update entries in the matrix, for each metric, as follows:

$metric(p,m) = (metric\ (p,m) - \mu_m) / \sigma_m$

As a consequence, all values in the matrix will be of the same order of magnitude, so that we can compare them.

Step4: Decision-making

For all projections p, let *metric(p)* = $\sum_m metric(m,p)$. Normally, the highest value among *metric(p)* should be retained and thus we get the best 2D view.



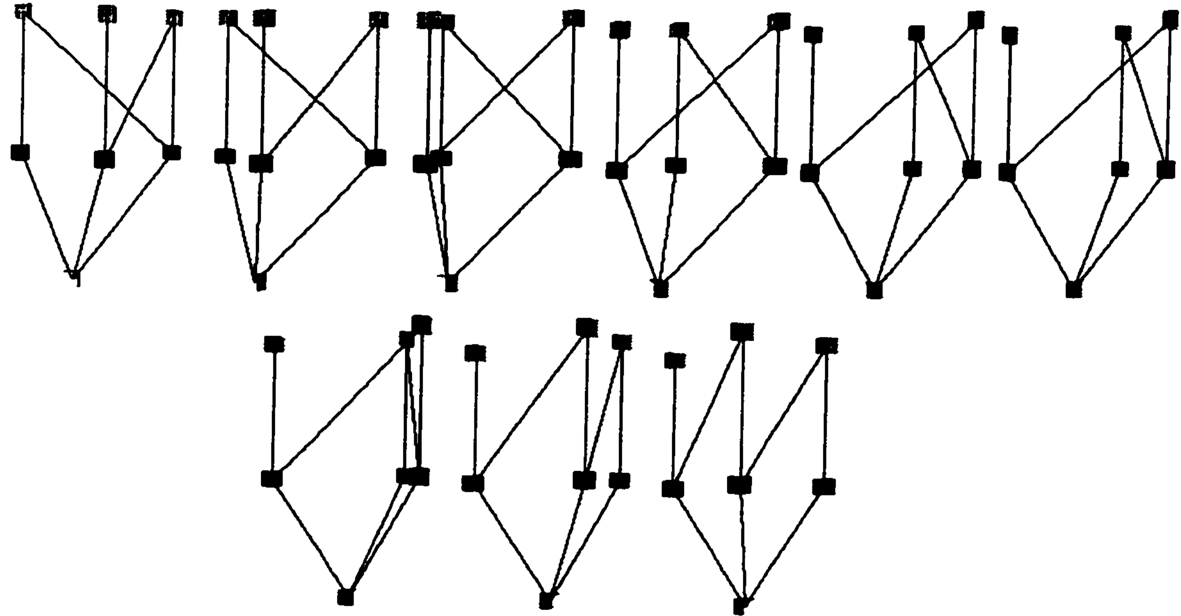


Figure 3.14: An ordered set represented in 2D (top-left), its 3D drawing (top-right) and 22 of its projections.

But, what if the user prefer one metric to another, or may be he/she wants to give much more importance to one metric and less to another, or may be a combination of them. In this case we associate weights w_i to the criteria, so that the user can express in a quantitative format its preferences, and then a decision would be like:

$$\text{For all } p,\ best_p = \max\left(w_1 \times metric(m_3, p) + w_2 \times metric(m_2, p) + w_3 \times metric(m_3, p) \right)$$

3.5.4.2 Case Study

Example #1

In figure 3.15, we have on the left-most picture, an order represented in 2D, in the middle, its 3D drawing using uniform level decomposition, and on the right-most, the 2D view which minimizes crossings and density.

We clearly see a major decrease in screen density and a slight decrease in edge crossing. In fact, by testing on many samples we concluded that our methodology associated with the choices of the user can improve the readability and the clarity of the structure been drawn in 3D.

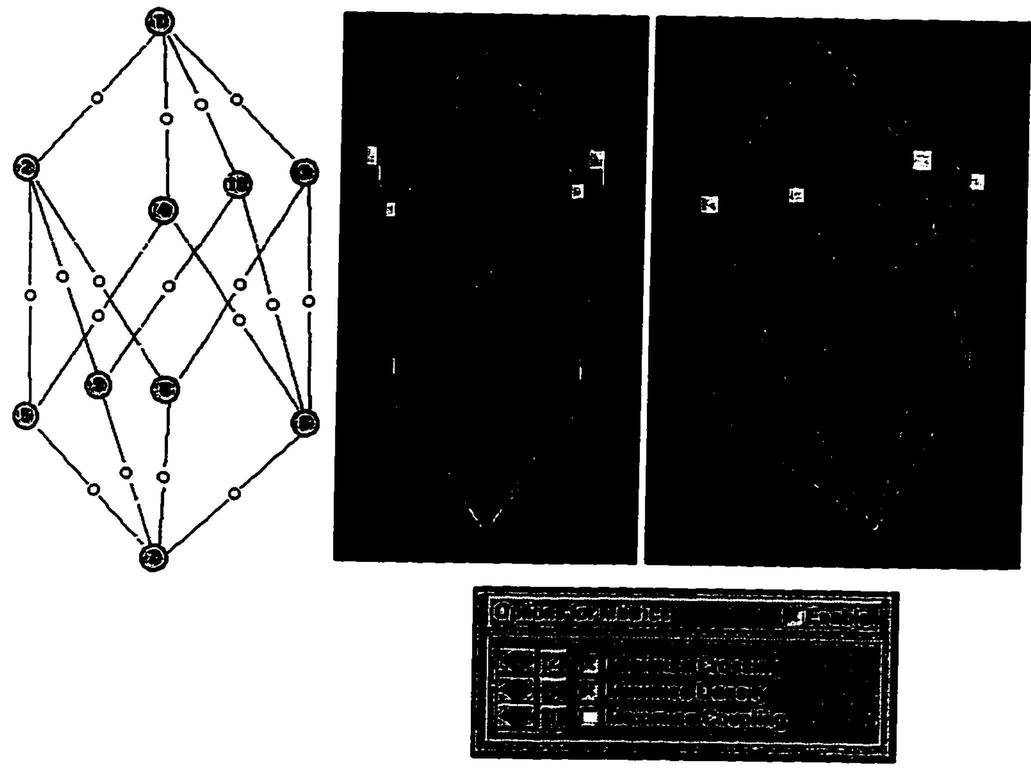


Figure 3.15: An example of using metrics and measurements.

The small window below both 3D representations in figure 3.15, contains weights associated with criteria's, in this example, the user prefers to minimize crossing more than minimizing density, and doesn't care about the coupling.

The following table shows the results for the matrix *metric(m,p)* at the end of the step4.

metric(p,m)	Crossings	Density	2xC + 1xD + 0xC
view p=1	-1.603567	.6859943	-2.52114
view p=2	-.2672612	-1.886485	-2.421007
view p=3	-.2672612	.6859943	.1514719
view p=4	1.069045	.6859943	2.824084
view p=5	1.069045	-.1714987	1.966591
view p=6	-1.603567	.6859943	-2.52114
view p=7	-.2672612	-1.886485	-2.421007
view p=8	-.2672612	.6859943	.1514719
view p=9	1.069045	.6859943	2.824084
view p=10	1.069045	-.1714987	1.966591

We see that the best view is the 2D view at the angle $2\pi/4$ or $2\pi/9$. We emphasize the fact that all entries in the matrix are centered, in order to make them comparable and homogenous, since they are going to be weighted. Lower negative values mean worst cases and higher positive values denote best cases.

Example #2:

In the following example (figure 3.16), we notice that giving a high weight to a criteria can prevail on other criteria. In fact, in the top-right 3D representation, we have the uniform level decomposition of an order (top-left), we see that the coupling between nodes is weak.

In the bottom-right image, we notice a very high degree of coupling, since we asked the system to compute the metrics while giving a high priority to the coupling. Meanwhile, density increases and edge-crossing remains as it was in the original representation.

3.5.4.3 Comparative study

An automated comparative study on 200 ordered sets, collected from different sources, such as; order series, order related books, and other papers, has revealed that both methods for representing ordered sets in three-dimension, had impacts on the three criteria. We present in the following table a summary of our observations:

Criteria / Method	*Uniform_LD*	*Barycentric_LD*
C - Coupling	Low	High
D - Density	Low	High
E - Crossings	Relatively low	Average

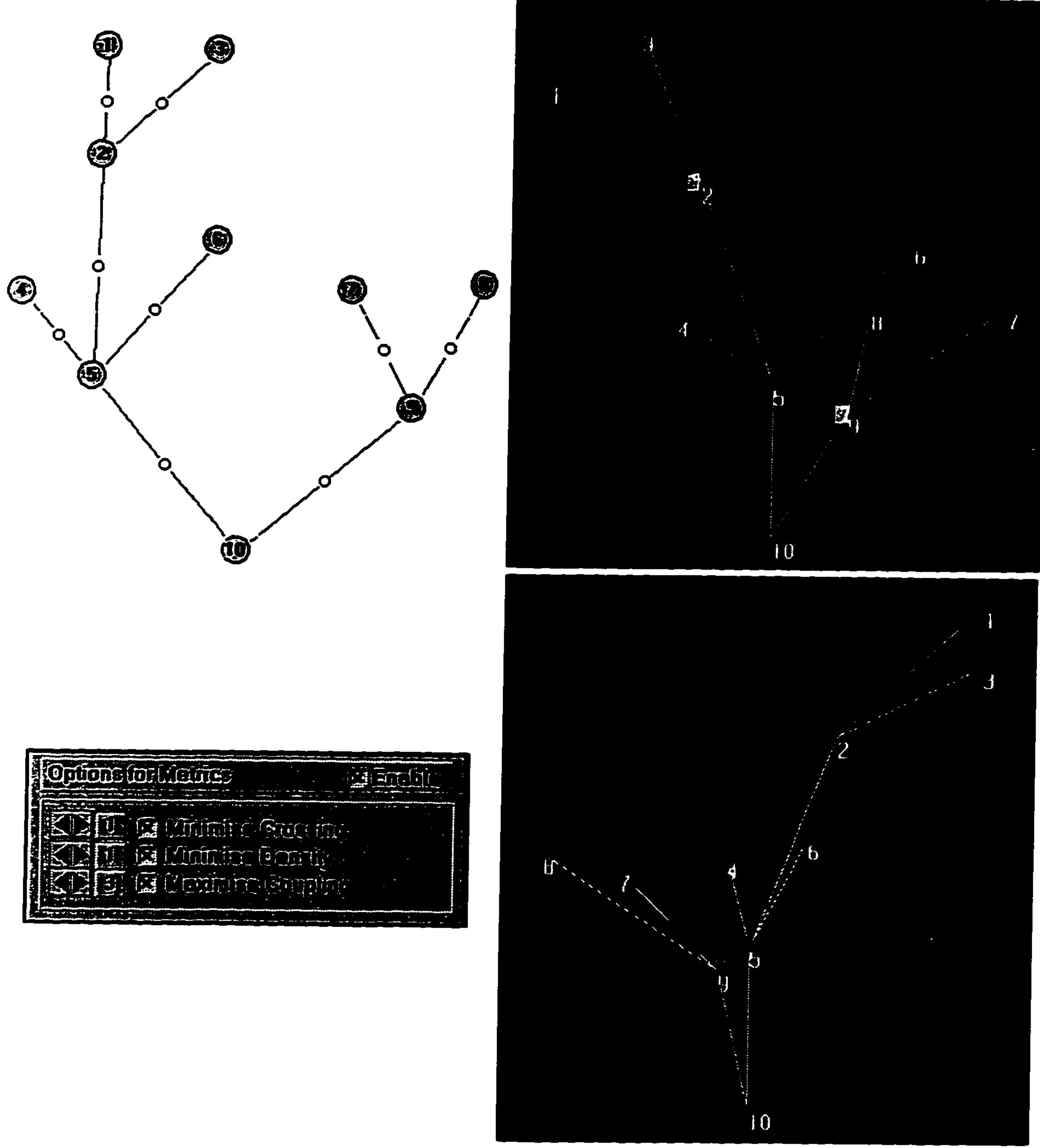


Figure 3.16: Maximizing the coupling between nodes verses optimizing other criteria.

3.6 Chapter summary

All along this chapter we talked about the data structure visualization, in other words, how to draw a structure from 2D, to 3D, and also how to find the best 2D view from the new 3D drawing.

We presented our main data structure which its ordered set, its definition, its notation and its representation in 2D.

Then we described two methods, based on level decomposition. Briefly, the first: *"uniform level decomposition"*, places the vertices on virtual circles uniformly, however the second, *"barycentric level decomposition"*, places the vertices according to the barycentre of their covers.

And then, in order to enhance and improve the readability of the drawing in 3D, we introduced some user-controlled features, such as adding dummy nodes, level rotation, level symmetries, and z-coordinate adjustment.

Next, we addressed the issue of 2D views, which is a projection of the 3D drawing on the user screen, since we believe that these views are the real interface between the user perception and the information conveyed by the 3D structure. We defined three basic criteria, and we associated precise metrics to them, then, based on the set of 2D projections, we took up measurements of these metrics and found the best view which optimizes these criteria. We illustrated all these computations by comprehensive examples.

Finally, we discussed the impact of the 3D drawing methods on these metrics, and presented in a table, the synthesis of our observations.

In the next chapter, we are going to talk about the second part of our dissertation, which is the paradigm of algorithm animation, the art of mapping and association, the exploration of the dynamic behavior of an algorithm while running, as well as, the concept of rendering and animation.

4. Algorithm Visualization

In this chapter we are going to discover and describe the second main part of our dissertation which deals with algorithms and their visualization and/or animation. We remind that data structure visualization (discussed in the previous chapter) is the first step for building an algorithm animation environment.

4.1 Abstractions

Algorithm visualization is the graphical presentation, monitoring, and exploration of computations. More precisely, it is considered to be as a mapping from some aspect of an algorithm (or execution of a program) to a final sequence of images. This leads to the paradigm which suggests that a natural implementation of a visualization consists of three "computations" working concurrently. These computations will form our interactive model.

The first of the computations is the algorithm whose behavior is to be visualized, or the *underlying computation*. The second, *visualization process*, implements the mapping by examining the dynamics of the underlying computation and transforming it into some graphical form. Finally, the *rendering process* displays the images generated by the visualization process, including straightforward graphical object motion and animation, while providing some simple viewer interactions.

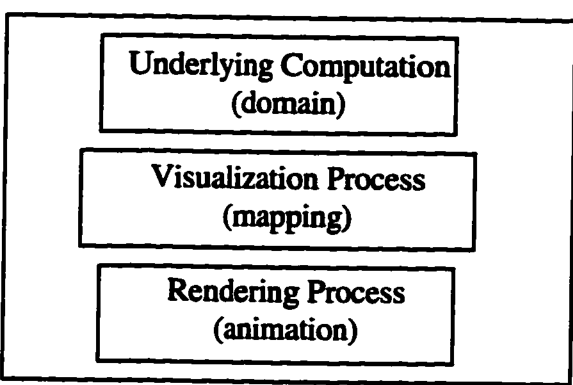


Figure 4.1: Interactive model.

We remind that, the word "computation" is used in its most general sense here, to encompass all the many ways in which these functions can be performed. All three computations may be part of a single process, or each computation might itself consist of a large number of closely - or loosely - coupled processes, or even be partly implemented in hardware (particularly in the rendering component).

This tripartite division is also reflected in the roles involved in a visualization. The programmer implements the algorithm, the animator constructs the visualization, and the viewer or the end-user examines the final images. We see clearly here, that too many participants take place in the whole process, this has a negative effect, since one participant is highly dependent on the other. In the next paragraph we will discuss another approach related to this matter.

4.1.1 Participants

We believe that at most two participants should take place in the process of animating an algorithm, the programmer, who is entirely responsible of designing the algorithms, and implementing them. The animator-user, a second participant should be capable of carrying out the following actions.

1. Defines and modifies the underlying computation.
2. Sets the rules for the visualization process.
3. Interacts with the rendering process.

Although, some requirements have to be fulfilled such as, a minimum knowledge of algorithmic, the data structure's problem domain , and visual interaction paradigm.

But, in order, for the animator-user, to be able to carry out his mission, the system should provide high level of abstractions and fulfill all the requirements stated in chapter 1, ranging from pertinence to informativeness.

4.2 Underlying computation

Here, the question is: what is the concept to be animated and in which format it is represented ?

The answer can be found in the title of the chapter, in fact, the concept is that of algorithms.

4.2.1 Algorithms

The term algorithm denotes..."*a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end*".

This definition is rather vague (step-by-step) because an algorithm can be more difficult to design and implement when the problem is some how tricky, and then be more complex and hard to understand, especially when it involves recursivity and external calls.

4.2.2 Focus

As stated in problem domain section in the third chapter, our basic data structure will be ordered sets, and a natural consequence to this, will be to focus on algorithms related to ordered sets (see section 3.1.1).

4.2.2.1 Classification

We are interested in three categories of algorithms on ordered sets, which are:

1. Algorithms for computing parameters.
2. Recognition algorithms.
3. Construction algorithms.

The first category is intended for computing common parameters of ordered sets, such as the height of an order, the width, the number of covers, etc., ...

The second includes algorithms for finding sub-structures within the order. Examples of these algorithms are: X-Free recognition algorithms, where X can be N, K or another interesting structures, series-parallel algorithms, etc.,...

Construction algorithms include those who generate new structure from the original one, such as linear extensions algorithms, chain decomposition, etc.,...

4.2.3 Algorithm Layout

Our first step toward animating an algorithm involves making it appear to be more interactive and easy to understand to end-users. To do this, we introduce two notion:

1. Structured algorithm, which is the algorithm we wish to animate, modified in certain ways.
2. Pseudo-code representation, which is a copy of the algorithm code, but statements are modified, in certain way, in order to gain a better perception and understanding of the actions within the algorithm.

4.2.3.1 Structured algorithm

We define a structured algorithm as the following: A structured algorithm is an algorithm that has been separated into three major parts, to handle the algorithm code, the algorithm parameters, and the algorithm interface. See Figure 4.2.

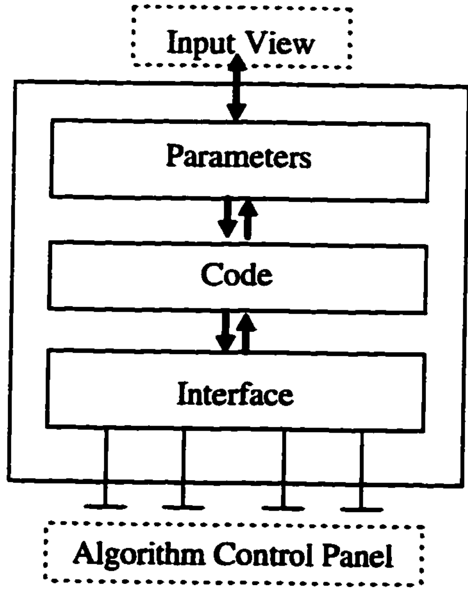


Figure 4.2: Structured Algorithm.

In this figure we see how the algorithm interacts with the other components within the algorithm animation system.

First, the algorithm parameters part is responsible of communicating the data from the input view (section 5.4.1) to the algorithm code. The parameters part has to find the

adequate and optimized format for the input data, in order to be delivered to the code. Consideration for space occupancy and organization of the data structure are crucial for the running of the algorithm and especially for its animation.

Second, the algorithm code part, waits for incoming messages from the algorithm interface to run and launch the many subroutines of the code, in order to process the data. We will see in the next paragraph how the code is organized, to achieve this purpose.

The interface is the dash board of the algorithm, it receives signals from the algorithm control panel, - a part of the algorithm view (section 5.4.2) - mostly controlled by the user, and then transforms them to messages to the algorithm code.

4.2.3.2 Pseudo-Code

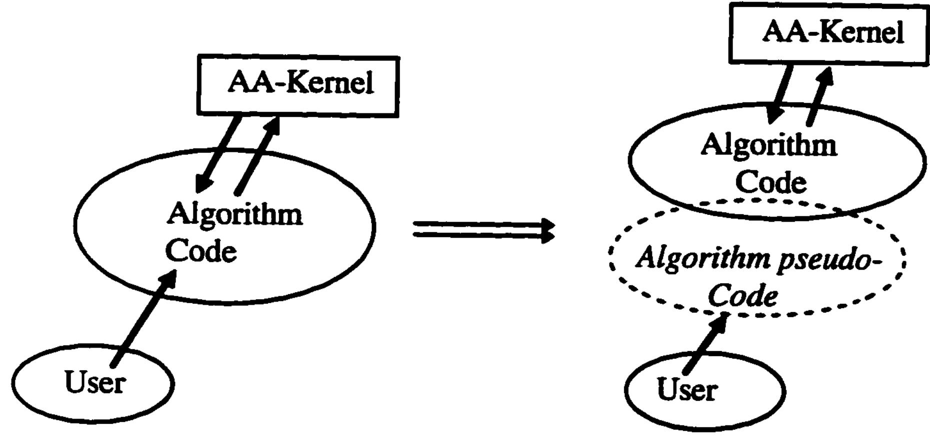


Figure 4.3: Algorithm code duplication.

Most of the algorithm animation systems did not care about the way the algorithm code is written, indeed they all provided a raw version written in low-level language. We find this to be inappropriate for the algorithm animation paradigm and contrary to our requirements, because it does not help, in any case, the user to comprehend the underlying computation and thus to build a correct animation.

To cope with this, we thought about duplicating the internal code of the algorithm and thus get a new aesthetic format while keeping the functionalities of the original algorithm. In other words, statements in the internal code are re-written in better format.

Generally, internal code is written in low level language, such as PASCAL or C and addressed to the computer to be interpreted/compiled (section 1.2.4) and then executed. Meanwhile, pseudo code could be much easier for the user to understand and to manipulate.

In the following paragraph, we show how internal code is transformed into pseudo-code.

1. Statement format:

   All expressions will be re-written in pre-order notation, i.e:

   a = b + c → Set (A, Plus (b , c))

   a ≤ b → Condition ("≤", a , b)

   This has proven to be very important because it highlights the operations of the algorithm.

2. Organization:

- Every statement or expression should be on a separate line, i.e.:

  theta = pi/2: alpha = cos (theta), should be re-written as follows

  1. Set (theta , divide (pi , 2))

  2. Set (a , cos (theta))

- Semantic grouping: Divide the pseudo-code into 3 different sections reflecting the components of the internal codes which are: Types, Variables, Body.

3. Expressiveness: Most of the algorithms taken from the literature or already coded lack considerable on-line documentation and expressiveness. Pseudo-code should provide details (of the function) for each statement, and give approriate significant names specifically to variables to match the context of the algorithm.

Because an algorithm has different sources: papers, books, as well as, electronic format in a given programming language, we think that only the programmer is able to transform this raw version into a pseudo-code.

In the next two fragment of code, we illustrate the difference between an algorithm written in BALSA-II and an algorithm written in pseudo-code. Even if the principle of the algorithm is quite simple, like in our case, the insertion sort, the way we present it has lots of impact on the way we read it. Especially, when the animator-user is going to focus on the algorithm and capture its essential operations as we are going to discuss it in the following section.

```
Procedure Alg.InsertionSort
        var i,j,v: Integer
        a: array [1..MaxN] of integer

Begin
        for I:= 2 to N do
        begin
                v:=a[i]; j:=i;
                while a[j-1] > v do
                begin
                  a[j]: = a[j-1]; j:= j - 1;
                end;
                a[j] : = v
        end;
end.
```

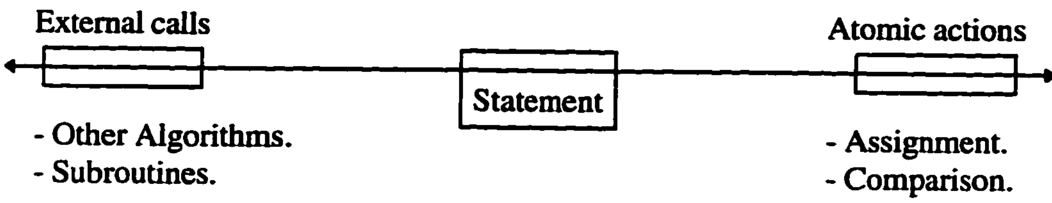
```
Types:

Variables:
        i,j,v : integer
        a : array of integer

Algorithm.InsertionSort
        For i  from 2 to N
          set (v,a[j])
          set (j,i)
          while condition (">",a[j-1],v) is True
                set (a[j], a[j-1])
                dec (j)
          loop
          set (a[j],v)
        next
end.
```

Figure 4.4: On the left, an algorithm taken from the BALSA-II system. On the right, its pseudo-code representation.

4.2.4 Algorithm Annotation

We have seen in the previous section a structured layout of the algorithm, and now we're going to investigate deeply the main component of an algorithm which is the code.

The algorithm code is a collection of blocks and control structures, for instance, statements within conditions and iterations. The word statement ranges from a low level (atomic) statement to a high level (external) statement as follows:



The next classification presents an encapsulation of statements according to their level of complexity.

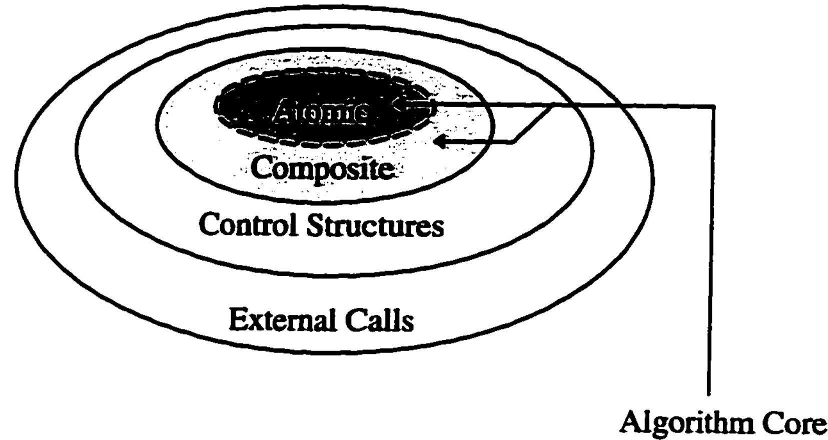


Figure 4.5: Range of statement in an algorithm.

We investigate in details the mean of each category:

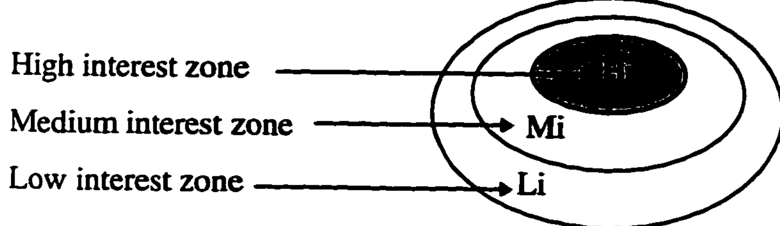
- Atomic: An atomic action is a basic algorithmic operation which can not be divided further more and runs in constant time, i.e., an assignment operation, a direct access comparison.
- Composite: A composite statement is a combination of atomic actions, i.e., embedded actions like: *dec (j)* which is really *Set (j, minus (j,1))*. It could also be a logical or a mathematical function.
- Control structures: Such as; iteration, condition or block of statements.
- External Calls: Could be, external algorithms, or external complex procedures whose main function is to achieve a specific task in the algorithm, without investigating how it is achieved.

We note that, for a given algorithm, we can have different representations depending on the level of complexity of each algorithm operation. In fact, one representation could be constructed by decomposing each composite into its corresponding atomic actions, and thus obtaining a low level version of the algorithm. Meanwhile a high level representation could be viewed as grouping atomic actions which share a common context to one composite actions. In our thesis, we will not represent

low-level algorithm, nor high-level algorithms, but we tend to find a middle ground between the two options, in order to express, what we consider only the fundamental operations.

PS: We did not address the issue of animating external calls (such as the optimal matching call in the width algorithm) and recursive calls. We will assume also that composite statements and atomic actions both form the algorithm core, that will be mapped and animated later.

4.2.4.1 Algorithm Core



Let C denotes the algorithm core which is the set of all atomic actions and composite in the code of an algorithm. We associate with each element c - by means of a function Fi - of the set C, one and only one value *i* which expresses the level of interest of an action.

4.2.4.2 Annotation

The annotation process [NB94], is done by both the programmer and the animator-user.

First, the programmer identifies and ensure that all actions in C are annotated with empty markers, then reduce the set C by eliminating all the algorithm actions which could not be animated in any way (called also non-animable actions, i.e., a_2 in figure 4.6). Then, the animator-user, defines its C' by capturing the desired operations, as described in the next section

4.2.4.3 Capture operation

We define the capture operation as being the act of choosing a subset C' from the set C, such that, for all c in C', Fi(c) is maximum.

This operation is highly influenced by the animator-user, since, he is the unique participant who will build the animation according to his needs and also to his understanding of the underlying algorithm.

Increasing the size of the Set C, implies increasing the possibilities of perceiving the algorithm from different points of view, since the animator-user can focus on certain parts of an algorithm and ignore the others.

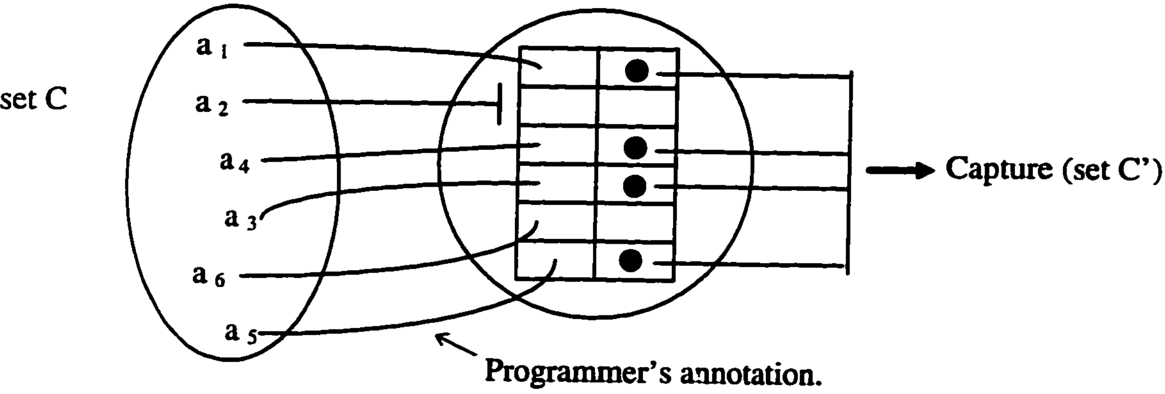


Figure 4.6: Annotation and Capture.

Here is an annotated program of the algorithm which finds the maximal elements in an order. We remind that this algorithm has been transformed form the original one, in order to be animated. The algorithm is written in VB. and its pseudo-code is shown in Annex I.

```
Sub Run_Max ()

'define AA-POSET variables
        Dim i as Integer
        Dim maximal as String
        Dim Chosen as T_Node

'process step by step

'STEP 1: Init maximals to null
        maximals = ""

For i = 1 To totalnodes
```

```
'STEP 2: choose randomly a vertex which is not marked

        Do
                Chosen = Int(Rnd * totalnodes) + 1
        Loop Until Not Node(Chosen).marked

        'annotation of operation select/choose
        animate op_select, Chosen

'STEP 3: Look for an uppercover of Chosen, if it exists animate both elements

        If Node(Chosen).UpperCovers <> "" Then
                UpperCover = get_upper(Chosen)
                ' annotation of operation subset/element++
                animate op_subset, Chosen + UpperCover
        end if

'STEP 4: keep chosen if it has no upper covers

        If Node(Chosen).UpperCovers = "" Then
                maximal = maximal + Chosen + ","
                ' annotation of operation keep
                animate op_keep, Chosen
        End If

        'mark the node as being treated.
        Node(Chosen).marked = True

Next
```

4.3 Visualization process

This section has proven to be very important because in it we present two key features of an algorithm animation system, which are graphical objects, object transformation, and then the important concept of mapping.

4.3.1 Graphical objects

We are going to introduce, in this paragraph, our main graphical objects which will be considered as the graphic repository for the graphical transformation.

Our classification, defines, types of objects, their attributes, and their specification(s), [MS93], we include also a snapshot of the 3D view and their context in the underlying data structure. We will denote the ensemble of object types as the set OT.

Object	Context	Example / View	Attribute	Specification(s)
3D Line	Edge		Position	Predetermined [1] Relative [2] Algorithm dependent [3]
			Size	Predetermined Relative
3D cube	Vertex		Position	Predetermined Relative Algorithm dependent
			Size	Predetermined
2D Label	Label	10	Position	Predetermined Relative
			String	Predetermined Relative
3D Moon	Marker		Position	Relative Algorithm dependent
			Size	Predetermined
Object Array	Subset		Position	Relative Algorithm dependent
			Number	Algorithm dependent
			Size	Relative Algorithm dependent

Figure 4.7: Graphical objects.

[1] Static: Determined by data structure visualization
[2] Static & Dynamic: Relative to other object, i.e, the position of an edge is relative to its tails.
[3] Dynamic: Value depends on the state of the algorithm during execution.

4.3.2 Object Transformation

A graphical object is defined by its existence and its essence. The existence is both construction and destruction, while the essence is composed by object motion, and object metamorphosis.

4.3.2.1 Object motion and metamorphosis

Object motion and metamorphosis are not yet considered as real-time animation, because of their primitive character. Motion is related to change in position and location, rotation and translation. For example, in a linear extension algorithm, a vertex is moved from its original location to the vertical linear extension by means of translation.

Meanwhile, metamorphosis denotes changes in shape, color and size. Such as highlighting, enlarging and reducing graphical objects.

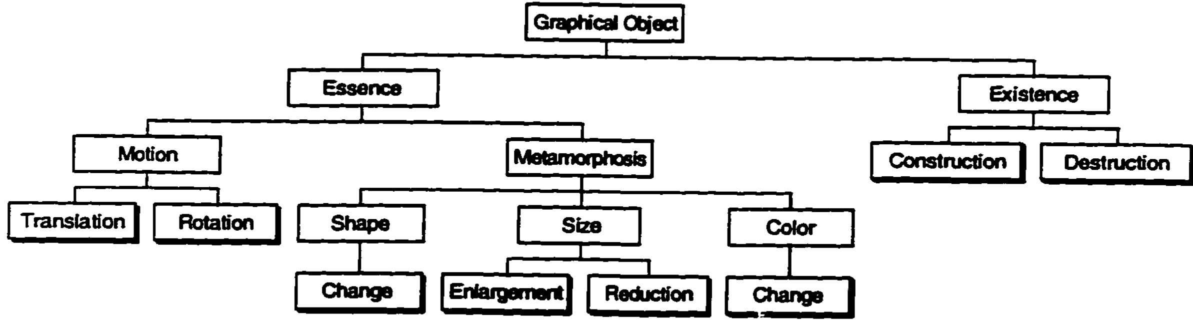


Figure 4.8: Object Motion and metamorphosis.

We define, Elementary Transformations, the set ET, as being the fundamental basic actions that can occur on objects from OT. (in figure 4.8, the shadowed boxes)

4.3.2.2 Combined Transformations

A combined transformation is a combination of element(s) from ET. In other words, we can have a complex graphical transformation by means of a sequence of elementary graphical transformation.

We denote the set of all combined transformation by CT. In the following table we present some of the combined transformation from the system.

Name	Class	Scope	ET Decomposition	Description
none	ET	Null	Null	Idle
show	ET	One / Array	OT.Color.Change	Highlight
show_grad	ET	Array	OT.Color.Change Wait	Highlight
shape	ET	One / Array	OT.Shape.Change	Change shape
vibrate	CT	One / Array	OT.Translation +[4]	Vibrate
flash	CT	One / Array	OT.Color.Change +	Flash
satellite	CT	One	OT.Construction OT.Size.Reduction OT.Shape.Change OT.Translation OT.Rotation OT.Destruction	Object Focus: A moon is created to rotates around a specific object
tracker	CT	Two	OT.Construction OT.Size.Reduction OT.Shape.Change (OT.Translation & OT.Rotation) + OT.Destruction	Object Tracker: A moon is created to fly from a source object to a destination object.
flash_edge	CT	One	OT.Color.Change +	Flash
color_edge	ET	One	OT.Color.Change	Change Color
remove	ET	One / Array	OT.Destruction	Remove object
disappear	CT	One / Array	OT.Translation OT. Destruction	Make object disappear
move_apart	ET	One / Array	OT.Translation	Move apart object
construct	CT		OT.Construction OT. Translation	Construct new object
drag	ET	One	OT.Translation OT.Shape.Change	drag object to a new location
show_comp	CT	Array	OT.Construction OT.Color.Change OT.Destruction	Show complement graph

[4] the + symbol means a repetitive action.

move_camera	CT	One	Cam.Motion + (See Section 4.4)	Change Camera Position

Figure 4.9: Combined graphical transformations.

4.3.3 Mapping

4.3.3.1 Process of Mapping

The next figure shows the process of the mapping during the visualization process. We see two different starting points. The first one is the algorithm core, which is subject to the capture operation by the user, in order, to generate the set C'.

The second is OT, the set of the pre-defined graphical objets. Based on OT, we construct a set ET, which represents the elementary graphical transformation routines, then combine them to get a more complex set denoted by: Composed graphical transformations or also CT.

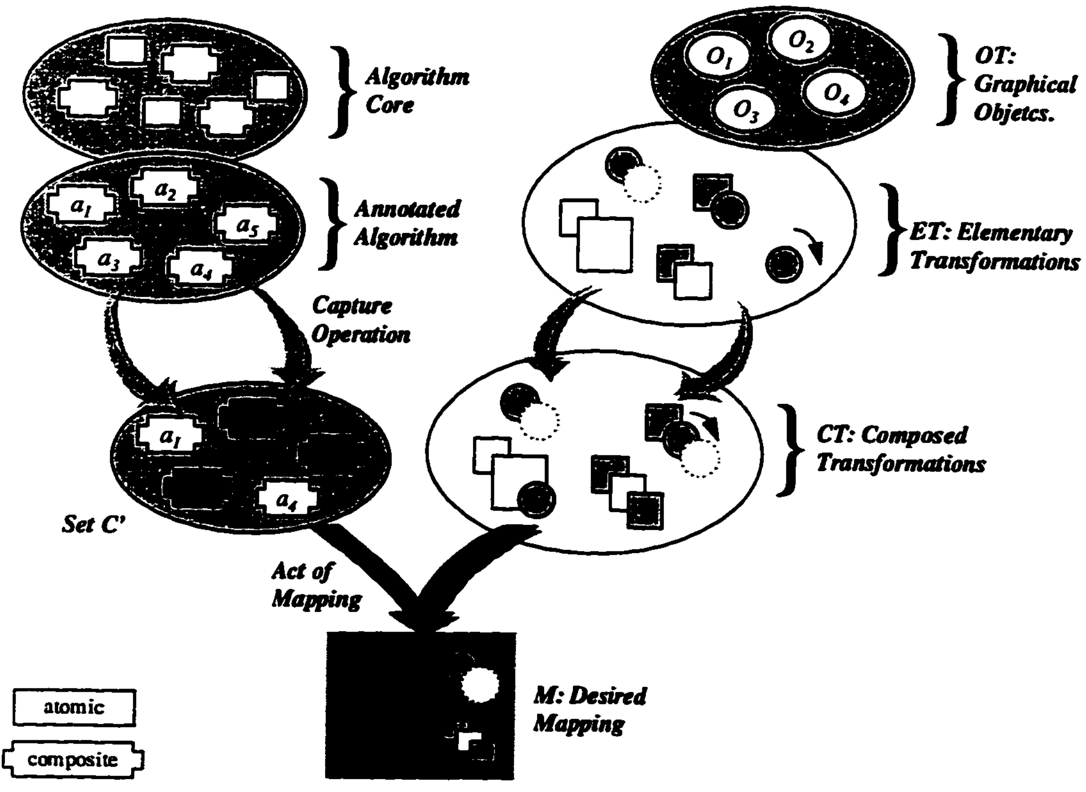


Figure 4.10: Act of Mapping

The act of mapping, consists then of associating one element from C' to another element from CT.

4.3.3.2 Pre-Mapping

Pre-mapping is the set of all possible mappings from the element of C' to those of CT. Pre-mapping is a key issue in mapping, because it preserves the semantic meaning of algorithm actions, and prevent any hazardous association.

For example, the result will be inconsistent, if we let the user to associate the algorithm action: *select,* to the composed graphical transformation: *Object construction.*

Pre-mapping is done by the programmer after both, the annotation phase and the design and implementation of ET and CT.

4.3.3.3 Free-Mapping

The Free-Mapping consists of choosing an appropriate desired mapping from the set of possible mappings. We use the world free here, to express a high level of interactivity between the user and the system, since, the user can design as much as possible mappings depending from which point of view he wants to animate the algorithm, and which part of the algorithm should have the focus.

We realized through our study of the previous work, that most of the algorithm animation systems have unfortunately ignored the free-mapping paradigm. In fact, they were limited to unique pre-mappings, and didn't gave the user the flexibility to express his desires.

Another matter not addressed in previous work also, is that, existent systems were not capable of changing the user-defined mapping during algorithm execution and animation, an innovation to this is to design a mapping mechanism which can be set in static time (visualization process) and changed in dynamic time (rendering process or animation)

4.3.3.4 Informatory mapping

We believe that, in order, to produce a highly interactive system, the user should be able to understand what's going on. Informatory visualization is a new way to present all steps of the visualization process. In other words, every action or invitation by the system should be fully detailed, as well as, captured algorithmic actions and composite actions (C'), and their counter-parts; the graphical objects (OT), the elementary graphical transformations (ET) and the composed graphical transformations (CT).

4.4 Rendering process

In this section, we will describe our methodology for generating appropriate animation of algorithms which have been previously described in the underlying computation section.

4.4.1 Event generation

In order, to run an algorithm, we need to have either an interpreter or a compiler. The advantage of having an interpreter is that the program is executed line by line, therefore the user can have much control during interpreting rather than executing the whole program one at a time.

Our **interpreter** is the same as the programming language's interpreter. In fact, interpreting a program involves traditional mechanisms, such as: run, suspend, resume, halt, step-by-step, and variable watch. In this paragraph, we are not going to investigate these operations, since they are detailed in section 5.4.2.

We will consider all captured algorithm actions and composite, as being algorithmic **events** during the execution. Let c' be a captured algorithm action or an algorithm composite from C', and t a given instant, we define then an algorithmic event e as being the pair (c',t).

It follows that the interpreter will generate all along the execution, a finite set of algorithm events, denoted by E, which in reality represents all the captured algorithmic actions and composite, from the algorithm core, executed at different instant of the algorithm execution.

4.4.2 Graphical Specification

When an event is generated by the interpreter, then it is submitted to the adapter which is a part of the graphic view. An **adapter** is responsible for building graphical specifications, based on the set of events *E*, and the user-defined mapping *M*.

We define a graphical specification as being a tupelo S_t (*ct, param, post, t*) consisting of 4 elements: *ct* denotes a composed graphical transformation from CT, *param* is a set of parameters related to the event *e*, *post* is the set of elementary graphical transformations needed to implement and achieve the specification, and *time* is an index of the temporal dimension (when to start).

For each event received from the interpreter during the execution, the adapter generates a unique specification S_t as follows: $S_t = f_M(e)$ where f_M is a the adapting function.

4.4.3 Animation Sequence

In this paragraph, we will introduce the last component of the Rendering process. The renderer receives from the adapter a graphical specification and transforms it to what we call sequences of animation. A sequence consists of a defined number of frames. but before going into detail let us define these technical words.

- A frame *fm*, is the low level element of an animation, it is a snapshot of the screen at a given time. Comparatively, in a movie-making, a frame is the atomic element, in fact, we can consider a 10 sec movie, as being, 250 consecutive frames, at a rate of 25 frames per sec.
- Time gap *u* between two consecutive frames. For example. if we have n frames, the duration of the sequence will be of order $(n-1) \times u$. We remind that the *u* is pre-defined and also user-defined.
- An animation sequence *Seq(n)* as being the set of *n* frames $(fm_1, fm_2, \dots, fm_n)$ such that $fm_1.t = S.t$ and $fm_n.t \leq (n-1) \times u$, where $S.t$ and $fm_1.t$ are both respectively the starting times of the graphical specification *S* and the frame fm_1.

We define then a final animation as being the continuous play of all the animation sequences. Finally, an algorithm animation is the conjunction of the two simultaneous actions, the algorithm execution (interpreting) on one hand and its related animation (adapting and rendering) on the other hand.

4.4.4 Overview of the Rendering Process

Figure 4.11: Rendering process.

4.4.5 Off-line animation

We looked forward to the case where the animator-user does not want to explore a particular behavior of an algorithm by entirely going into its animation. Sometimes, the algorithm is tricky and building its animation (mapping and visualization) can take much effort and lots of time.

Off-line animation is a the algorithm visualization process, without going into algorithm execution. In other words, we can animate specific algorithm actions, by means of direct manipulation, which is associating pre-defined graphical specifications to them, without passing by the interpreter. The renderer take in charge the generation of animation sequences and the final animation based on these specifications. Off-line animation is detailed in section 5.4.4.

4.5 Chapter summary

This chapter was fully dedicated for algorithm visualization. So far, we have discussed three major topics.

The first topic, consisted of presenting the underlying computation, which is in our case, algorithms on ordered sets, then defining what is an algorithm, its structured layout, its presentation based on pseudo-code, and finally its decomposition.

We discussed in the second topic, methods for visualizing the algorithm core by means of graphical objects and graphical transformation, we introduced the concept of mapping which we consider, the fundamental procedure for adding animations to algorithms.

The third and last topic was intended to detail the rendering process, which, based on the previous defined mapping and the underlying computation, generates appropriate graphical specifications, in order to build animation sequences.

5. Interactive Environment

A substantial portion of the research reported here concerns the nature of the interactive environment for exploring animated algorithms. In this chapter, we present a "tour" of the system from an end-user's perspective.

In the first part, we are going to provide a comprehensive case study, in which we are going to illustrate in details all methods and features discussed in chapter 3 and chapter 4.

After that, we will present our system's architecture and then describe its three components or views, for instance, the algorithm view, the input view and the graphic view.

Our system currently runs on IBM-PCs and is a Microsoft Windows based application. This chapter assumes some familiarity with the basic windows interface, including pull-down menus, icons, display boxes, as well as general GUI terminology, such as double-clicking and dragging.

5.1 Case study #1

We are going in this case study, to consider an algorithm, and to develop all the steps for generating one of the many possible animations of that algorithm.

5.1.1 Underlying computation

5.1.1.1 Algorithm for Linear Extension

Let P = (X,P) be a poset with $|X| = n$ and let $L = [x_1, x_2, , \dots , x_n]$ be a linear extension of P. We find it useful to view L as being constructed by the following algorithm. We emphasize the fact that this algorithm is one of the many algorithms used to construct a linear extension.

Linear Extension Algorithm.

Set $X_0 = X$, $P_0 = P$, $\mathbf{P}_0 = (X_0, P_0)$, and $M_0 = min(\mathbf{P}_0)$
For $i = 0,1,2,\dots, |X| - 1$
  CHOOSE $x_{i+1} \in M_i$ */* choose randomly an element from the minimal */*
  SET $X_{i+1} = X_i - \{x_{i+1}\}$ */* construct a new set X by removing the chosen element from the old set */*

$P_{i+1} = P(X_{i+1})$ /* *re-define the ordering function on the new set* */
$P_{i+1} = (X_{i+1}, P_{i+1})$
$M_{i+1} = min(P_{i+1})$ /* *get the new minimal elements from the new order* */
End

5.1.1.2 Pseudo-Code

Let's consider the following pseudo-code representation of the linear extension algorithm. Underlined statements denotes elements from C (candidate atomic actions and composite actions, also known as the algorithm core).

The underlined and bold statements are elements of the reduced C, which is the set C minus the non-animable actions such as *Card, Set, and Sub.*

```
Types:
        Type_P: Poset
        Type_SV: Set of vertices
Variables:
        X: Type_P
        M,Lx: Type_SV
        i: Counter,  v: Vertex

Algorithm LinearExtension
        Init (M, Min (X))
        For i = 0 to Sub(Card(X),1)
                Select (M,v)
                Add (Lx,v)
                Minus (X,v)
                Set (M, Min (X))
        Next
end.
```

Our algorithm core C is composed of the following atomic and composite actions: Atomic actions: *Select*; composite actions: *Add, Minus, and Min (which is a function).*

We will assume that the animator-user will consider all actions of C necessary for the animation process, so that the capture operation will regard all element of C, so C'= C.

5.1.2 Visualization process

Before going into the mapping process, we should state the pre-mapping for each algorithm action, especially those of C. We remind that the pre-mapping process is a strictly programmer task.

5.1.2.1 Pre-Mapping

Algorithm [illegible]	[illegible]
Min	none, show, shape, vibrate, flash
Select	none, show, shape, satellite, move_camera, flash
Add	none, construct
Minus	none, remove, disappear

5.1.2.2 Free-Mapping

In our case study, we will consider one mapping which involves lots of graphical transformations, and in our opinion, this mapping presents a good animation of the algorithm. Meanwhile we do not claim that our proposed mapping is the best, since, each user has a different perception of the animation of an algorithm. Let M be one of the possible mappings, illustrated in the following table

Min	none, show, **shape**, vibrate, flash
Select	none, show, shape. satellite, move_camera, **flash**
Add	none, **construct**
Minus	none, remove, **disappear**

5.1.3 Rendering Process

In our case study we are going to consider the hypercube and its 3D representation using the uniform level decomposition. (see figure 3.3).

5.1.3.1 Interpreting

In this phase, we see how events are generated all along the running of the algorithm. Let's consider the first iteration of the algorithm, when i = 0. All along the iteration, four events will be generated, receptively e_1, e_2, e_3, and e_4. i.e.: $e_1 = (Select, t_1)$ where t_1 denotes the first instant when the event was generated.

In all, we will have $|C'| \times (|X|-1)$ events, since we have $|C'|$ actions and $(|X|-1)$ iterations.

5.1.3.2 Adapting

As described in section 4.4.2 and section 4.4.5, the adapter will take in charge the events, look for corresponding graphical transformations *Cts*, in the mapping set *M*, and then generates a graphical specification.

In our case, event e_1 will have the following specification:

$S_1 = (ct = M (Select), param = \{ X, X \rightarrow v \}, post = \{ OT.Color.Change + \}, t = t_1)$

Where: X is the underlying data structure, $X \rightarrow v$ is the focus element, $M (Select)$ is the *CT* associated with *Select*, which is in our case: ***flash***, and t_1 is the specification time.

5.1.3.3 Rendering

According to the specification, the renderer will generate primary frames and then if necessary animation sequences. (We note that sometimes an animation sequence can be only one frame).

In our case, the renderer will look at the specification of ***flash***, which graphical transformation is described in the post condition field of S_1, in our case: OT.Color.Change+.

Then, n frames will be produced, such that each frame will consist of a *DS*, with $DS \rightarrow v$ in a different color. snapshots of these frames are shown below (see vertex 7).

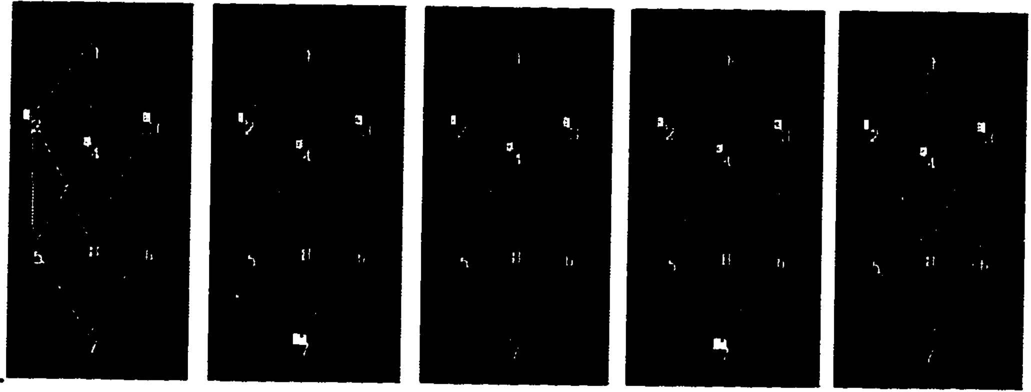


Figure 5.1: Animation of a graphical transformation.

Let's consider another specification, for example, of the event e_6 which is the operation add of the second iteration. $e_6 = (Add , t_6)$.

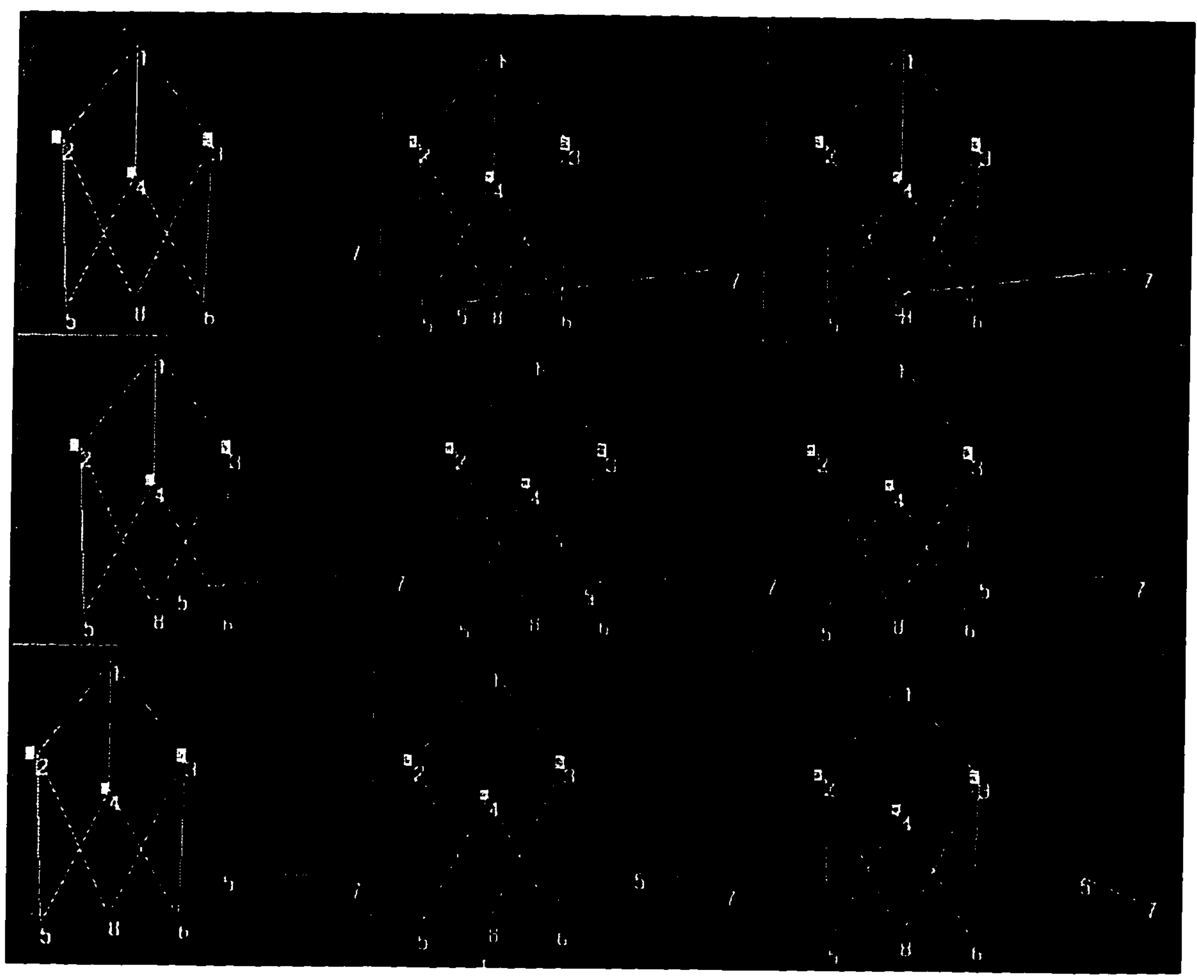
$S_2 = ($ $ct = M\,(\,Add\,),$

$\quad param = \{\,X,\ X\rightarrow v,\ Lx\,\},$

$\quad post = \{\,OT.Construction,\ OT.Translation\,\},\ t = t_6\,)$

Where, X is the underlying data structure, $X\rightarrow v$ is the element to add in Lx, $M\,(\,Add\,)$ is the *CT* associated to *Add*, in our case: *construct*, and t_6 is the specification time.

The renderer will then produce n frames, such that: The first one, will be the construction of a vertical linear structure (if not already build), the followings will be the dragging of the element v from its original position to its new position in Lx. A snapshot of theses frames is shown below.



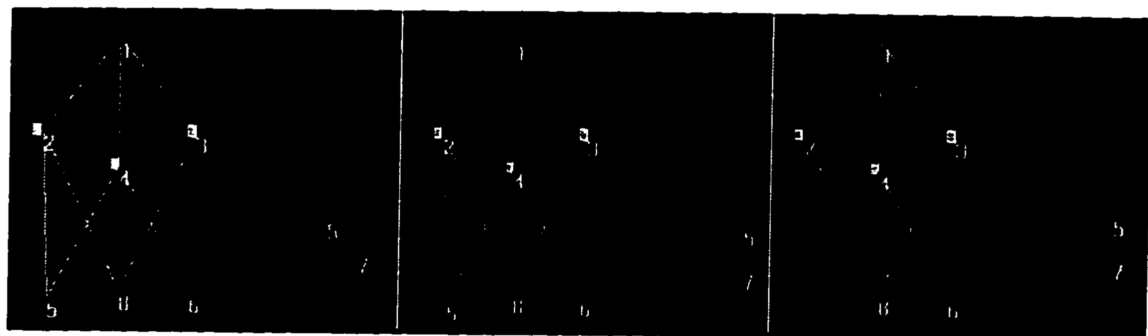


Figure 5.2: Animation sequence of an add operation.

We conclude this section by showing the last frames of the animation of the linear extension algorithm.

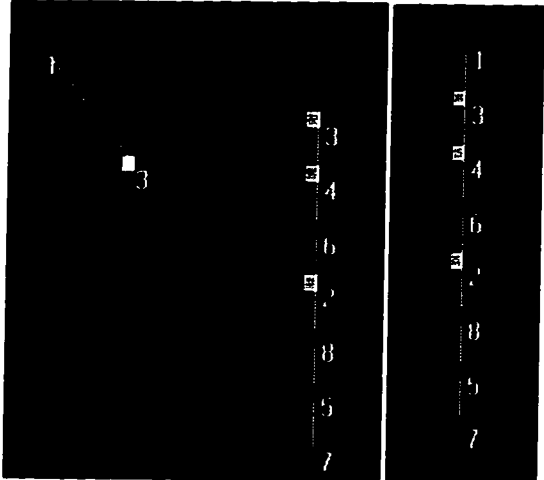


Figure 5.3: Snapshot of the final frames.

5.2 Case study #2

In this section, we are going to show few screenshots from the animation of the series-parallel recognition algorithm (see Annex I).

First we consider an ordered set composed of the three distinct components represented in 2D (figure 5.4 - left) and its 3D representation using the uniform level decompostion. (figure 5.4 - right).

As a preprocessing, the algorithm will generate an emtpy cotree for the corresponding order and transform the order to a comparability graph CG (Fig. S/S.1). In the following, we are going to detail the process which leads to each of the screenshots.

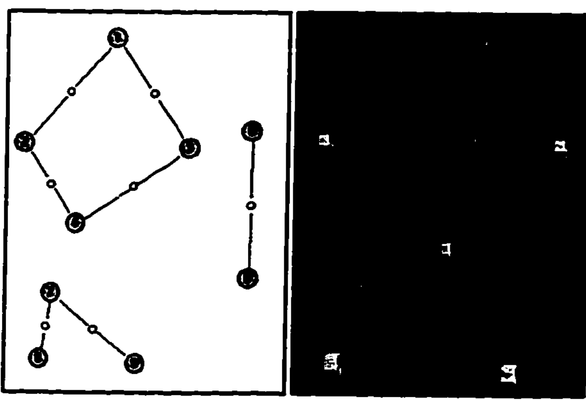
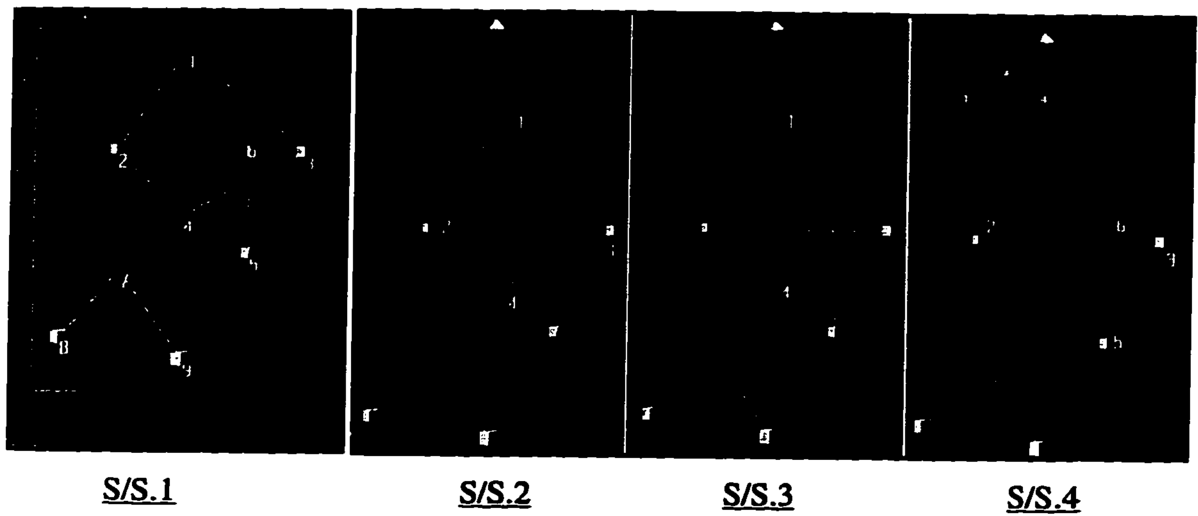
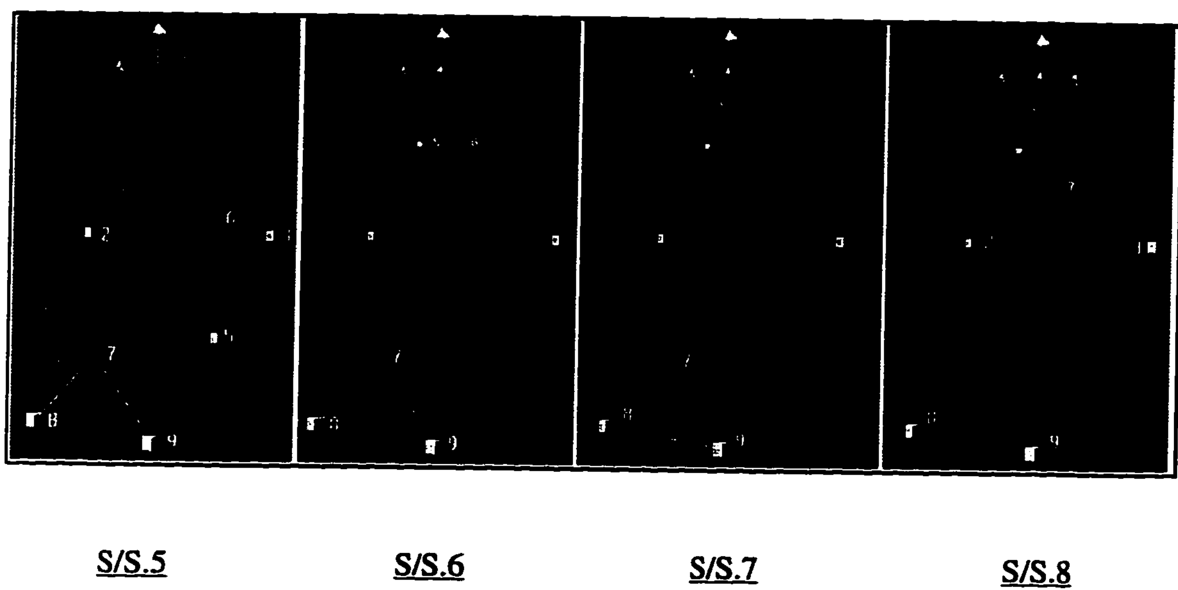


Figure 5.4: An order (left: 2D, Right: 3D) for the animation of SP-algorithm.



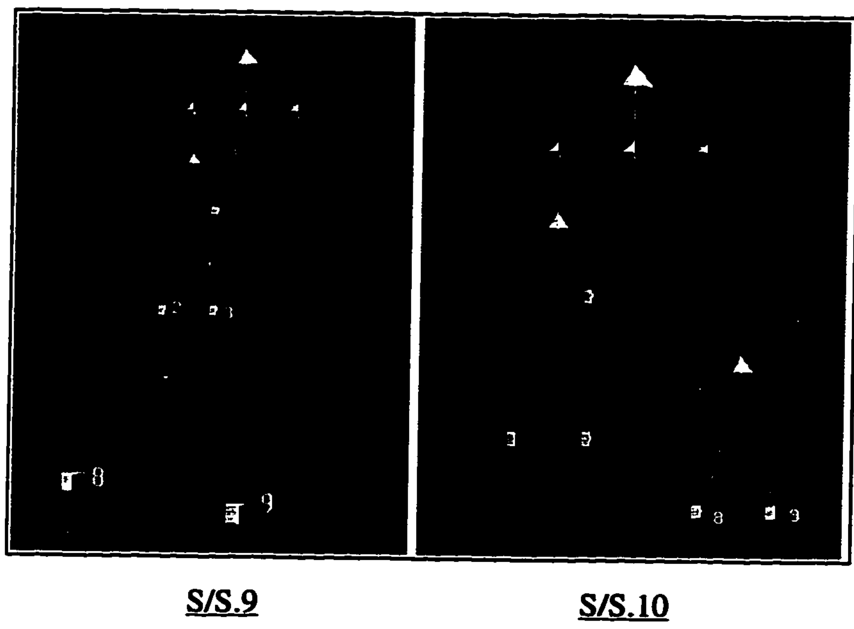
S/S.1 S/S.2 S/S.3 S/S.4

- S/S.1: the ordred set have the focus.
- S/S.2: First, generation of the comparability graph *CG* of the order, so that 3 connected components are detected $\{C_1=(1,2,3,4),\ C_2=(7,8,9),\ C_3=(5,6)\}$, second, creation of a tree, which root represents a parallel construction node and which has *n* children, where *n* is the number of connected components (in our case, three), and then focus on the first connected component C_1.
- S/S.3: generation of the complement graph of C_1, so that four connected components are distinguished: $\{C_{11}=(1),\ C_{12}=(4),\ C_{13}=(2,3)\}$, then focus on the two connected components C_{11} and C_{12}
- S/S.4: vertices *1* and *4* are added to the cotree under a series construction node.



S/S.5 S/S.6 S/S.7 S/S.8

- S/S.5: focus on the connected components C_3.
- S/S.6: vertices *5* and *6* are added to the cotree under a series construction node, and focus on the the connected components $C_2=(7,8,9)$.
- S/S.7: generation of the complementary graph of C_2, so that two connected components are distinguished: $\{C_{21}=(7), C_{22}=(8,9)\}$, then focus on the connected components C_{21}.
- S/S.8: vertices *7* is added to the cotree under a series construction node, then focus on the component C_{32}.
- S/S.9: vertices *2* and *3* are added to the cotree under a parallel construction node, then focus on the component C_{22}.
- S/S.10: vertices *8* and *9* are added to the cotree under a parallel construction node.



S/S.9 S/S.10

Figure 5.5: Snapshots from the animation of SP[1].

5.3 System's Architecture

Figure 5.6 shows the three main components of the whole system, the connections between them, the data flow control, the interaction points, and the data transducers.

Data transducers (boxes with thick margins), are black boxes, whose function, is to transform an information from a given format to another (i.e.: an interpreted algorithm action into an event, an event into a graphical transformation and a graphical specification into an animation sequence).

Interaction points (grey boxes), are location where the user interacts with the system. So far, we notice three interactions points: The first, is the editor, through which the user creates or loads the 2D drawing, then we have the algorithm control panel, responsible of the algorithm execution, suspension and termination, finally, the navigation panel, where the user can manipulate and explore the 3D structure.

Gray arrows denote a control over, such as algorithm control panel, controlling the algorithm execution, meanwhile black arrows, denote, a transfer of data.

[1] Symbols in the pictures include: Pyramid: for parallel construction, Octahedron: for series construction, black cubes: not yet determined.

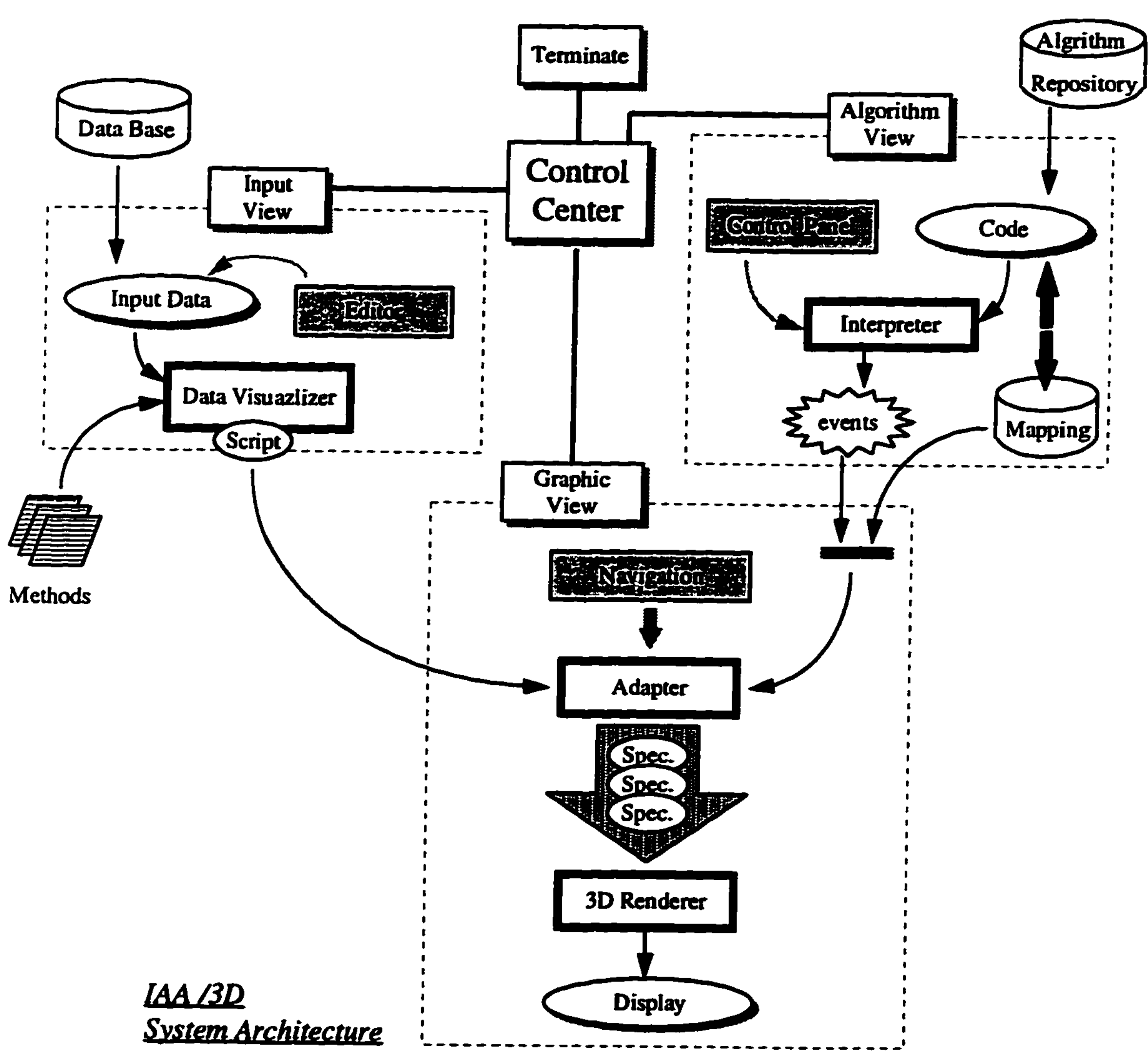


Figure 5.6: System's Architecture.

5.4 System's Components

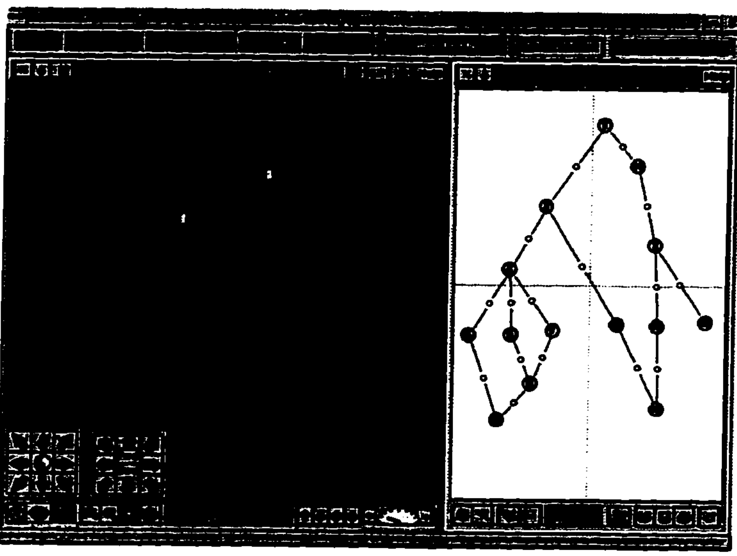


Figure 5.7: System screenshot.

5.4.1 Input View

The input view is an independent view within the algorithm animation system. It consist of three components:

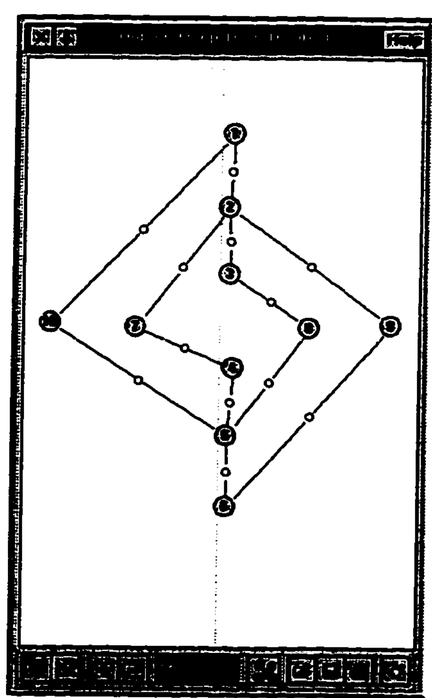


Figure 5.8: Input view.

1. *Editor (middle)*: In the editor we create 2D graphical objects, such as nodes and edges. We make the necessary arrangements in order to preserve the information conveyed by the structure.
2. *Control Panel (bottom)*: used to load a new structure, save the current drawing, clear the editor, add and remove instances of nodes or edges.
3. Control Bar (up): used to get help, to minimize or to dismiss the input view.

5.4.2 Algorithm View

The Algorithm view regroups the algorithms that are subject to animation. We can see clearly five sections within the view:

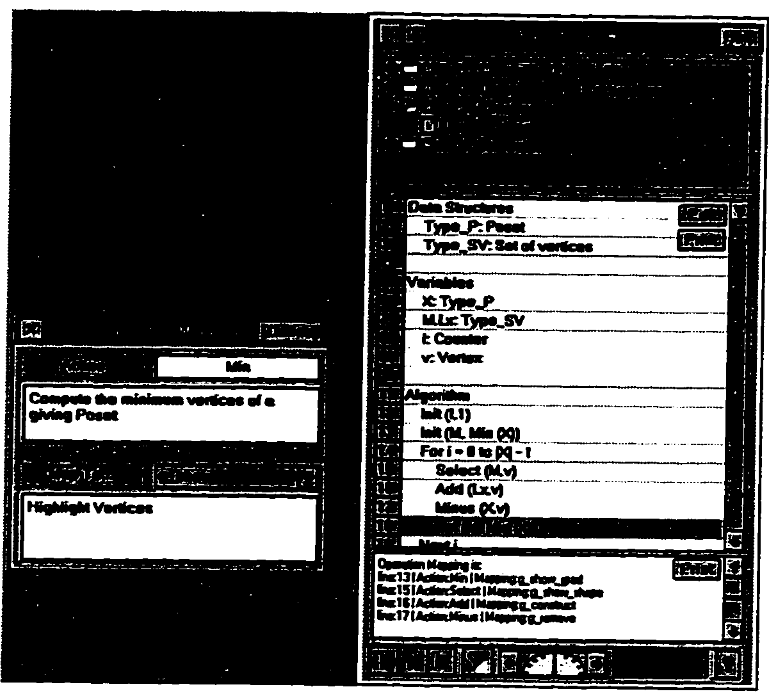


Figure 5.9: Algorithm view.

1. *Algorithm list*: where algorithms are classified by their types.
2. *Algorithm Pseudo-code:* with editing and printing features.
3. *Algorithm Trace:* To be most effective, algorithm animations must be accompanied by comprehensive motivational instruction. The quality of these teacher-provided explanations is perhaps even more important than the animation itself. To make algorithm animation simulate this instruction, the animation displays should be augmented by textual descriptions of the ongoing operations. As even better alternative would be a multimedia display with an audio-video window showing an instructor describing the algorithm.
4. *Control Panel:* Includes, run, suspend, stop, control speed, and an on-line help.
5. *Operation Mapping View:* Clicking on a specific line containing a mappable action or composite, in the algorithm pseudo-code, displays a new form, inviting the user to

associate (map to) a graphical transformation, from a pre-defined list (pre-mapping) to the current algorithm action.

5.4.3 Graphic View

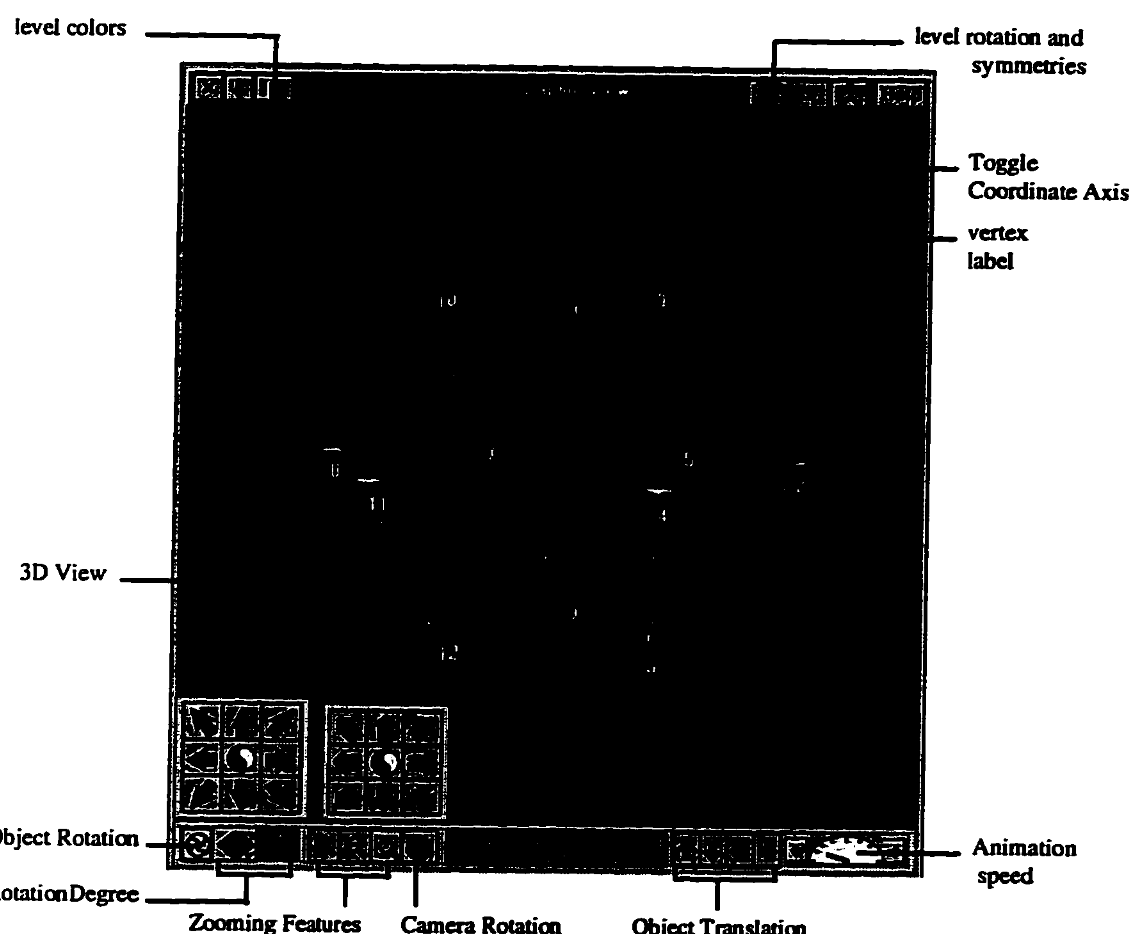


Figure 5.10: Graphic view.

5.4.4 Off-line Animation

Let's consider a 3D representation of an order (Figure 5.11) using the barycentric level decomposition. Off-line animation (the window inside the algorithm view) consists of animating algorithm actions whitout interpreting it.

The example in figure 5.11, shows the animation of the *Minus* operation mapped to the graphical transformation *disappear*. And some of the frames of this animation is presented below. (vertices 1, 2, 4, 5, and 9 are removed from the structure).

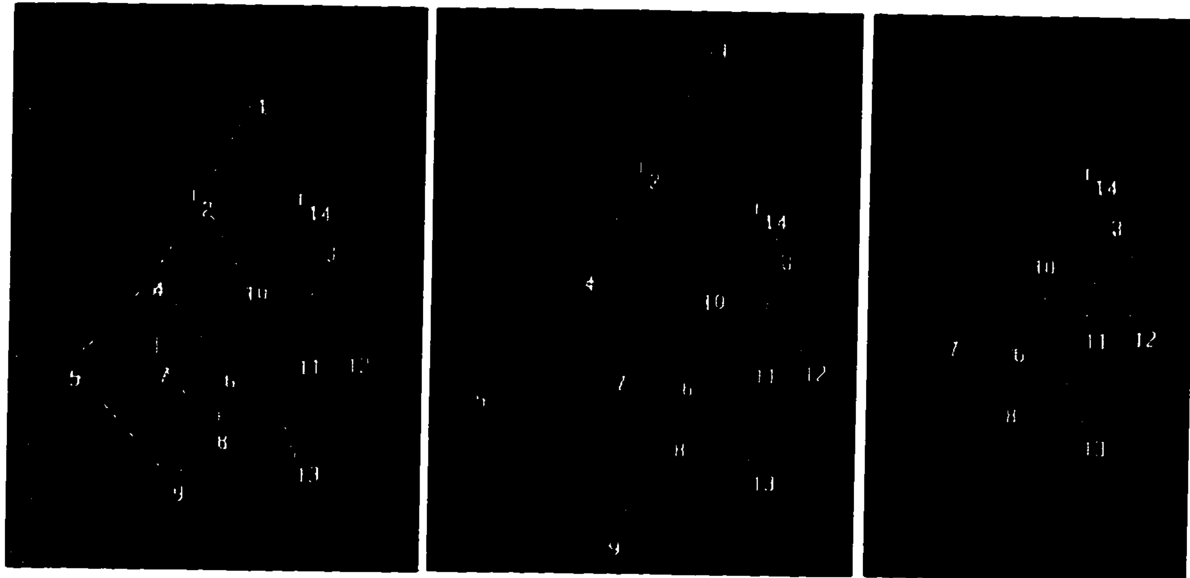


Figure 5.11: Off-line animation (removal of vertices)

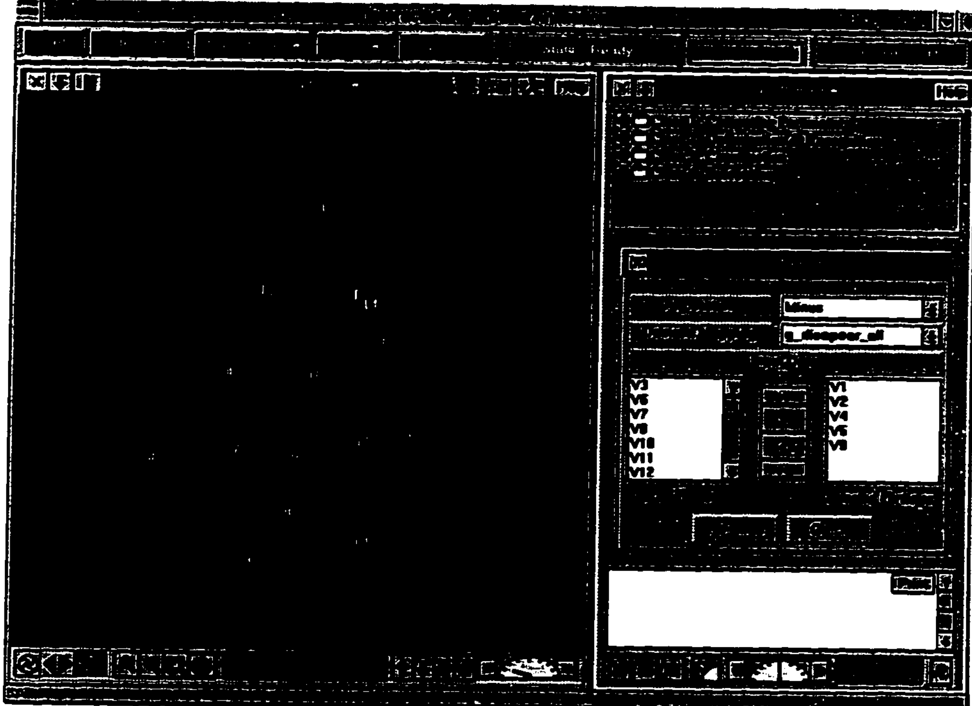


Figure 5.12: Off-line animation (removal of vertices)

It follows that the user is free to select any algorithm operation and to map it to any graphical transformation within the pre-mapping space, in order, to explore and to get an independent perception of each algorithm operation, without need to know which algorithm to choose to be animated.

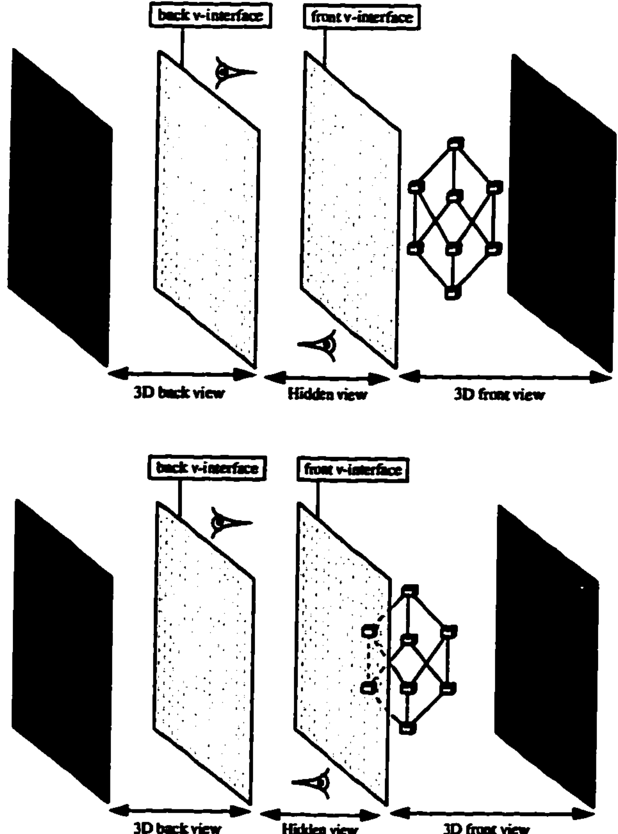
This - unexplored yet - step toward adding more effectiveness and interactivity to the algorithm animation process, has proven to be very successful especially, for relatively large data and long-time running algorithm.

5.5 3D Navigation

In this section we are going to describe two main features of the 3D navigation. The first consists of the in-depth exploration of a 3D structure, and the second is the Focus view point. We note that there are other standard and common 3D features, like camera motion and object motion, as well as, object translation and reduction.

5.5.1 3D In-Depth Exploration

We are going to describe this feature by the following pictures. We consider an ordered set in 3D (HyperCube) and we are going to navigate into the structure. We note that during all the following steps, all other navigation features are available, in design time, as well as, run-time.



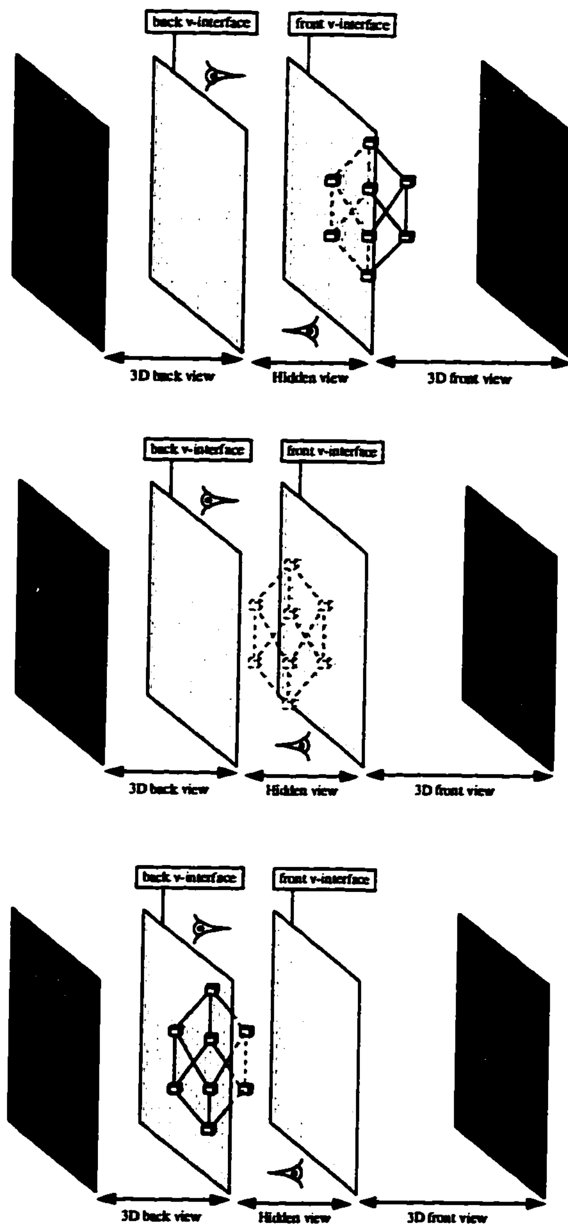
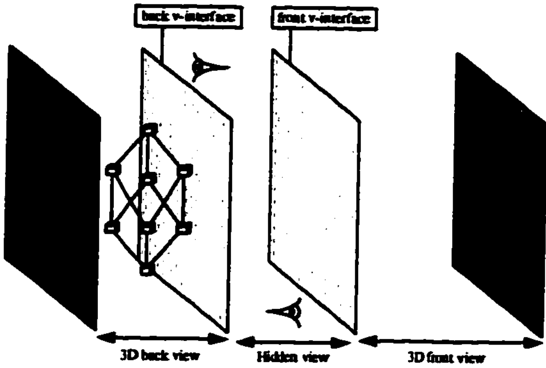


Figure 5.13: 3D exploration.

In-depth exploration, as shown is the pictures above, is the translation of the structure toward the front visual interface (user screen), until it disappears in the hidden view, then reappears in the 3D back view. We define 3D back as being the 3D front view observed from the back. In the third snapshot, the user has the perception that he starts to navigate into the structure, since only half of it is in the visible view, for instance the 3D front view, meanwhile in the fourth snapshot, the user has the perception that he's

completely in the structure, and since the distance between the visual interface and the components are too short, so it is impossible to have them within the visual range. We remind that a visual range is defined by a minimal distance between the visual interface of a 3D object and the infinite as a maximal range.

The next snapshot shows the last step of an in-depth exploration, for instance, the view of the structure from the back.



5.5.2 Focus Point

In order to have the focus on a certain location of a 3D structure, the user bring his focus first to an acceptable visual range, then, the system will relocate the all 3D components, into layers, defined by the distance of these components to the user screen. Each object in a layer will then have its size and appearance altered (augmented or reduced) according to that distance.

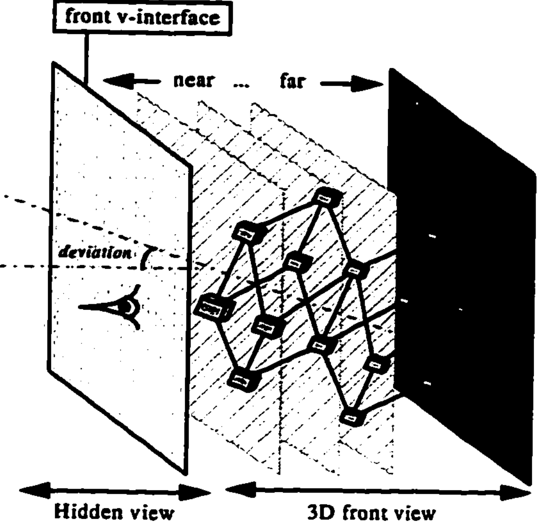


Figure 5.14: Focus Point.

6. Conclusion

6.1 Evaluation

In our thesis, we have presented two main parts: In The first part, we presented methods for data structure visualization in three-dimension, and an approach for best-viewing the structure in space. In the second part, we announced our methodology for algorithm visualization, by going through the underlying computation (algorithm), the visualization process (mapping) and the rendering process (animation).

The last part was mainly dedicated to a technical description of the system, by presenting its architecture and discussing its main components.

Unfortunately, algorithm animation is still a very labor intensive process. Lack of time and resources forbids us from implementing animations of all algorithms on ordered sets we would like. Even, with IDE & RAD (Integrated Development Environment) and (Rapid Development Method), it still takes days or even weeks to animate an algorithm of intermediate complexity.

One of our goals is to develop a sophisticated version of the current prototype allowing more productive work and a shorter development cycle. In the next section we will present means for achieving this goal.

6.2 Research Directions

Much more work remains to enable animated displays of algorithms and of other complex or abstract phenomena to become a mainstream component in computing environments. This section outlines areas for further research to help make this happen.

Generalization

By designing and implementing an advanced specific interpreter, we can animate all sorts of structured algorithms, especially those still unexplored algorithm related to ordered sets. This step could be very productive in means of time, flexibility and interactivity.

Graphics

In our prototype, the graphics used are acceptable but not perfect for representing both static three-dimensional structures and graphical transformations. Improvements could be: The use of another advanced 3D rendering library, the use of sophisticated programming language, such as VC++, and why not, changing from platform, to work on a silicon graphics station, and user their powerful OpenGL animation package.

Multimedia

In the next decade, multimedia will be a standard for all all-purpose and specific systems, a main research direction, could be integrating multimedia capabilities, like sounds and movies into the animation process, such as: adding voice to algorithm tracing and system messages, adding quicktime movies showing an instructor describing the algorithm during its execution, and enabling the recording of ongoing animations.

Scripting

Mapping scripting is the fact of designing pre-defined mappings for each algorithms, this can be useful, in the case where the user does not want to go into mapping details, and just want to explore the animation based on some pre-defined views of the algorithms.

Multiple-Views and use of 3D.

This feature worth mentioning, consist of implementing multiple-views for both graphic view and algorithm view. For the graphic view, it is possible to have many views showing a different shot in space, or also expressing the third dimension, such as: history capture, or simply aesthetics. Meanwhile multiple-algorithm views could be the way of animating embedded algorithms calls, or independent algorithms manipulating the same structure.

6.3 Final thought

Every reader will remember something different about this thesis. Some might remember a handful of screen images portraying the visualization of ordered sets in three-dimension, others might remember our straightforward 3D navigation capabilities and the automated computations for finding the best view. still others might remember our

interactive model for adding animation to algorithms. While these pictures, 3D structures, and conceptual ideas, are, perhaps, interesting and innovative, they are not the essence of this dissertation.

A picture is worth a thousand words, a movie even more. But interactive animation, available on workstations to virtually everybody, are even more valuable. The difficulty in future will not be a lack of hardware, it will be lack of software, and the lack of creative visual thinking to drive new software. This thesis will contribute to its being looked upon in years to come as but a primitive attempt to harness computer hardware as an aid for understanding and thinking about complex phenomena.

Annex I

A.1 Algorithms

Here is the pseudo-code for the algorithms that have been animated by the system.

We note that: <u>action</u> → Algorithm Core and **<u>action</u>** → Captured Algorithm operation.

Computing Parameter Height [Tro92 pp7-9]

```
Data Structures
        Type_P: Poset
        Type_SV: Set of vertices
Variables
        X: Type_P
        A: Array of Type_SV
        I: Counter
Algorithm
        Init (I,1)
        Init (A,{})
        While Condition (X,{},"=")
                Inc (I)
                Set (A[i], max (X))
                Minus (X,A[i]))
        Loop
        output(I)
End
```

Compting Parameter Width [Tro92 pp4-5]

```
Data Structures
        Type_P: Poset
        ype_SV: Set of vertices
Variables
        X: Type_P
        C: Array of Type_SV
        I: Counter
Algorithm
        Call Matching [Lau89 pp189]
        Init (C,{})
        Init (I,0)
        While Condition (X,{},"=")
                Inc (I)
                Set (C[i], Subset("chain",X))
                Minus (X,C[i]))
        Loop
        output(I)
End.
```

N-Free Recognition algorithm [Riv87 pp140-141] (slightly modified).

```
Data Structures
        Type_P: Poset
        Type_SV: Set of vertices
Variables
        V: Type_P
        N: Type_SV
        Mark: Array of Verticies
Algorithm
        Init (Mark,0)
        While Condition (V,<>,{})
                Select (v)
                If Conditon(Subset("Impred",v)),<>,{})
                        For all u in Impred(v)
                                Select (u)
                                If Condition(mark(u),<>,0)
                                        Mark(u,v)
                                Else
                                        Set(w,mark(u))
                                        Set(U,Union(Impred(v),Impred(w)))
                                        Set(I,Intersect(Impred(v),Impred(w)))
                                        Set(E,Minus(U,I))
                                        If Condition(E,<>,{})
                                                Subset("N",{v,w,u,first(E)})
```

```
                                        Output("The Order is not N Free")
                                        Terminate
                                    End if
                                End if
                            Loop
                        End if
        Loop
        Output("The Order is N Free")
End
```

Series-Parallel Recognition algorithm [Riv87 pp121]

```
Data Structures
        Type_P: Poset
        Type_SV: Set of vertices
Variables
        X: Type_P
        G,Gc: Graph
Algorithm
        Set (G,CompGraph (X))  /* Comparability Graph */
        Set (N,Vertex(G))
12:     Set (CC(),ConnComp(G|N)) /* Connected Components */
        if Condition (CC,<>,{})
                Label (T,N,"P")
                AddSons (T,N,CC)
                Goto 25
        End if
        Set (Gc,Complement(G|N))  /* Complement Graph */
        Set (CC(),ConnComp(Gc|N))
        if Condition (CC,<>,{})
                Label (T,N,"S")
                AddSons (T,N,CC)
                Goto 25
        End if
        Output ("X is not S-P")
        Terminate
25:     For All p in Nodes(T)
                if Condition (|p|,>=,2)
                        Set (N,Select(p))
                        Goto 12
                Else
                        Output ("X is S-P, T is the CoTree")
                        Terminate
                Endif
        Loop
End.
```

Algorithm for finding the maximals

```
Data Structures
        Type_P: Poset
        Type_SV: Set of vertices
Variables
        X: Type_P
        A: Array of Type_SV
        maximals: Type_SV
Algorithm
        Init (A,{})
        For all nodes v
                Select (v)
                Set (A, Subset("OnePred",v))
                if Condition (A,=,{}) keep(v)
                Add (maximals,v)
        Loop
        output(maximals)
End.
```

Algorithm for finding the minimals

```
Data Structures
        Type_P: Poset
        Type_SV: Set of vertices
Variables
        X: Type_P
        A: Array of Type_SV
        minimals: Type_SV
Algorithm
        Init (A,{})
        For all nodes v
                Select (v)
                Set (A, Subset("OneSucc", v))
                if Condition (A,=,{}) keep(v)
                Add (minimals,v)
        Loop
        output(minimals)
End.
```

Linear Extension construction algorithm: See section 5.1

A.2 3D Library

Visualib version 2 is a comprehensive state-of-the-art graphics library for the Microsoft Windows environment. It contains powerful and efficient functions for rendering both 2D and 3D graphic objects. Visualib 2.0 consists of several DLLs which can be used with any Microsoft Windows develop environments such as Microsoft C/C++, Microsoft Visual Basic, and Borland C++ version 2.0 and up.

Complete 2D and 3D viewing systems allows flexible view settings. Sophisticated transformation mechanism supports virtually all types of graphics transformations.

Visualib provides many different lighting, shading, material, and other rendering options. Lights can be created individually with different characteristics.

For smooth animation effects, **double buffering** is supported for both 2D and 3D viewers. z-buffer is also available to handle complex backface elimination. Visualib supports a full set of common 2D and 3D drawing functions and the powerful curve and surface drawing functions such as Bezier, Hermit curves, B-Spline, NURBS curves and surfaces. Visualib also includes a large collection of graphics primitives.

VisualLib is registred to **Visual Tech Co.**
P.O. Box 8735
Fort Wayne, IN 46898-8735
(219) 489-0235

References

[Baker] M. Pauline Baker. "After the Storm: Considerations For Information Visualization", NCSA, University of Illinois.

[Bas85] David B. Baskerville. "Graphics Presentation of Data Structures in the DBX Debugger", Report No. UCB/CSD 86/260, University of California at Berkeley, Berkeley, CA, October 1985.

[BK91] Bently, J.L, Kernighan, B.W. " A System for Algorithm Animation", Computing Systems, vol. 4, no 1, Winter 1991.

[Bro88] Marc H. Brown. "Algorithm Animation", MIT Press, Cambridge MA, 1988.

[Bro91] Marc H. Brown. "Zeus: A System for Algorithm Animation and Multi-View Editing", DEC Systems Research Center, CA, 1991

[Bro94] Marc H. Brown. "The 1993 SRC Algorithm Animation Festival", DEC Systems Research Center, CA, July 29, 1994.

[CR92] Kenneth C. Cox, Grucia-Catalin Roman, "Experiences with Pavane Program Visualization Environment", Tech-Rep WUCS-92-90, Dept. of CS, Washington University, Saint-Louis, MO, , October 1992.

[DETT93] G. DiBattista, P.Eades, R.Tamassia, and I. Tollis. "Algorithms for drawing graphs: an annotated bibliography", Tech-Rep, Brown University, June 1993.

[Dui86] Robert A. Duisberd. "Animated Graphical Interfaces Using Temporal Constraints", Proc. ACM SIGCHI'86 Conference on Human Factors in Computing Systems, April 1986, 131-136.

[Hop74] F. Robert A. Hopgood. "Computer Animation Used as a Toll in Teaching Computer Science", Proc. 1974 IFIP Congress, 1974, 889-892.

[Lau89] Lau, H.T. "Algorithms on Graph", TAB BOOKS Inc, Blue Ridge Summit, PA, 1989.

[LD85] Ralph L. London, Robert A. Duisberg. " Animating Programs Using Smalltalk", IEEE Computer, 18,8 (August 1985), 61-71.

[Mod92] Mitchel L. Model. "Monitoring System Behavior In a Complex Computational Environment", CSL-79-1, Xerox PARC, Palo alto, CA 1979.

[MS93] Sougata Mukherjea, John T. Stasko. "Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding.", GVU-GIT, Atlanta, GA, 1993.

[Mye90] Brad A. Myers. "Taxonomies of Visual Languages and Program Visualization", Journal of Visual Languages and Computing, (1990), 1, 97-123.

[NB94] Marc A. Najork and Marc H. Brown. "Building Algorithm Animations with Zeus", DEC Systems Research Center, CA, June 1994.

[Riv84] Ivan Rival. "Graphs and Order", Proceedings of the NATO ASI Series, Banff, Canada, May 84.

[Riv87] Ivan Rival. "Algorithms and Order", Proceedings of the NATO ASI Series, Ottawa, Canada, May-June 1987.

[Riv94] Ivan Rival. "Introducing Ordered Sets", Lectures on Ordered Sets, Dept. of Computer Science, University of Ottawa, Canada, 1994.

[Sea92] Andrew Sears. "Layout Appropriateness: A metric for evaluating user interface widget layout", Computer Science Dept., University of Maryland, December 1992.

[SP91] John T. Stasko, Charles Patterson. "Understanding and Characterizing Program Visualization Systems", Tech-Rep GIT-GVU-91-17, Georgia Institute of Technology (October 1993).

[Sta89] John T. Stasko. "TANGO: A Framework and System for algorithm animation", Ph.D thesis and Tech-Rep No. CS-89-30, Brown University, Providence, RI, May 1989.

[Sta92] John T. Stasko. "Three-Dimensional Computation Visualization". Tech-Rep GIT-GVU-92-30, Georgia Institute of Technology (1992).

[Tei84] Warren Teitelman. "A Tour Through Cedar", IEEE software, 1, 2 (April 1984), 44-73.

[Tes81] Larry Tesler. "The Smalltalk Environment", BYTE, 6, 8 (August 1981), 90-147.

[Tro92] William T. Trotter. "Combinatorics and Partially Ordered Sets", Johns Hopkins University Press, Maryland, 1992.

[Tun94] Daniel Tunkelang. "A Practical Approach to Drawing Undirected Graphs", CMU-CS-94-161, School of Computer Science, Canegie Mellon University, Pittsburgh, PA, June 1994.