To the Graduate Council:

I am submitting herewith a thesis written by Debra Lynn Jones entitled "Algorithm Animation By Demonstration". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Brad Vander Zanden

Bradley Vander Zanden, Major Professor

We have read this thesis
and recommend its acceptance:

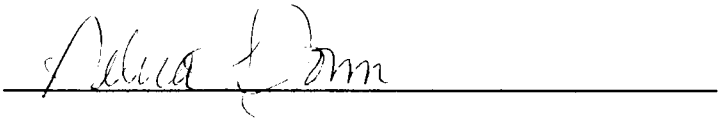
Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature 

Date July 19, 1994

Algorithm Animation by Demonstration

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Debra Lynn Jones

August 1994

Acknowledgements

I would like to start by thanking the University of Tennessee Computer Science Department, for providing me with opportunity for a graduate degree. I enjoyed the three years I spent as a teaching assistant. I would also like to thank my committee for their time spent on my thesis, as well as their helpful comments. But most of all, I would like to thank my advisor, Brad Vander Zanden. His patience with me was amazing and his insistence on doing things the right way has taught me a lot these last four years.

My family has also played an important role in pushing me along to finally finishing. It was a little unbelievable to them that I could be 98% done for such a long time. I'm no longer 98% done. Their support and interest in what I was doing meant a lot to me.

I would also like to thank all of my friends who have provided encouragement when I needed it and who have understood when I couldn't do interesting things because of my thesis. Their help and support of my and my sometimes seemingly impossible goal is greatly appreciated.

Lastly, I would like to thank Oracle Corporation and my managers for giving me the time that I needed when I needed it and for never assuming that I wouldn't finish.

Abstract

The growing trend towards graphical user interfaces and graphical representations for computer applications has created an interest in the animation of algorithms. Many environments provide the means for the creation of graphical data structures through the use of an interface or through procedure calls to graphical functions. Animation is then possible by manipulating the positions of these objects. This can be a time-consuming process involving a great amount of work to produce the animation. This project attempts to alleviate the burden of manipulating the objects in the graphical environment by providing a graphical interface to create the necessary elements for an animation, as well as the resulting animation. Through the creation of these elements, usually a data structure and transitional paths for its elements to follow, a generic animator is produced. The animator encapsulates the behavior demonstrated by the creation of the animation sequence, for example, the swapping of two objects. An algorithm can therefore be represented graphically by defining the data structure and the transitions that its elements make. These transitions are specified through the creation of visual paths which designate an object's starting and ending positions. Since an animator can handle the movement of several objects at once, the swapping of two objects is possible. Because an algorithm often performs the same types of actions on objects repeatedly, the same animator can be used to perform these actions by merely changing the objects that the animator is to move.

The methodology used for creating this animation package includes descriptions of the creation of data structures, their transitional paths, and how they are tied into one object, the animator.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Background 4

1.3 Overview of Examples 7

1.4 Goals 7

1.5 How the Goals Were Achieved 8

2 Related Work **10**

2.1 Animus 11

2.2 BALSA 12

2.3 TANGO 14

2.4 ZEUS 15

3 Examples **17**

3.1 Insertion Sort 17

3.2 Depth First Search 22

3.3 2-3 Tree Insertion 29

4 Programming Layer **34**

4.1 Internal Creation of Paths 35
4.1.1 Path Slots 35
4.1.2 Polyline and Line Paths 36
4.1.3 Arc Paths 36
4.2 The Animator Interactor 39
4.2.1 Object Related Slots 40
4.2.2 Path Related Slots 43
4.2.3 Path Transformations 46
4.2.4 Other Slots 48
4.2.5 Programmer Setable Slots 49
4.2.6 The Timing Function 49
4.3 Using an Animator 50

5 Visual Design Environment **53**
5.1 Paths 53
5.2 Timing Information 55
5.3 Data Structures 56
5.3.1 Trees/Graphs 56
5.3.2 Lists/Arrays 61
5.4 Creating An Animation 67
5.5 Save Options 67
5.5.1 Saving an Animator 67
5.5.2 Saving a Session 69
5.6 Loading a previous session 70

6 Conclusions **72**

Bibliography 74

Vita 77

List of Figures

1.1 Main menu of animation designer. 3

3.1 Design of swap animator. 18
3.2 List before elements begin swapping. 20
3.3 Beginning of insertion sort animation. 20
3.4 Insertion sort animation continued. 21
3.5 Insertion sort animation continued. 21
3.6 After the first two elements are swapped. 22
3.7 Example list to be used for bubble sort. 23
3.8 Creating the Depth First Search Graph. 25
3.9 Design of depth first search animation. 26
3.10 Traversal to the first adjacent vertex. 27
3.11 Traversal to the first adjacent vertex continued. 27
3.12 Traversal to the first adjacent vertex continued. 28
3.13 Traversal to the first adjacent vertex complete. 28
3.14 The 2-3 tree before node "17" is added. 30
3.15 Insertion of node "17". 30
3.16 Creation of a new parent node. 31

3.17 Moving second child node to the new parent node. 31
3.18 Both children have been moved to the new parent node. 32
3.19 Creation of a new root. 32
3.20 Insertion complete. 33

4.1 Sample Paths. 37
4.2 a and b of Arc Path. 38
4.3 ϕ of Arc Path. 38
4.4 Illustrate obj-to-change slot with data structures. 41
4.5 Illustrate obj-to-change slot with unrelated objects. 42

5.1 Editing an Arc Path. 54
5.2 Tree/Graph Menu. 57
5.3 Creation of a tree. 58
5.4 Tree/Graph Properties. 58
5.5 Adding a node to an already formatted tree. 60
5.6 Formatted tree. 61
5.7 Creation of a list. 62
5.8 Formatted list. 63
5.9 List/Array Editor. 63
5.10 Items Editor 64
5.11 Rank Margin 65
5.12 Traversal of a Tree. 70

Chapter 1

Introduction

1.1 Overview

In the field of computer science, many concepts are easier to understand when accompanied with visual imagery. Data structures, such as linked lists or graphs, can be easily shown pictorally. It follows that algorithms performed on these structures can often be demonstrated by showing the structures in stages as they are affected by the operations, such as a swap of two elements, performed on them. Even more helpful than viewing an algorithm in stages is being able to view the data structures as they are changed by the algorithm, including the transitions between the changes. This facilitates quicker understanding of a sometimes difficult concept. Additionally, it can prove to play an important role in the debugging of an algorithm. If a user can actually watch the data structures move as programmed, he can understand how the algorithm works and at the same time detect any errors. Thus, animating an algorithm is beneficial to a student as well as a teacher or programmer.

This animation toolkit was created to fill the growing need for a tool that allows

for the animation of algorithms by providing the user with the means to create the animation by demonstrating the desired actions. Several algorithm animation toolkits currently exist but do not provide the same ease of use as ours does.

Most of the algorithm animation systems reviewed in Chapter 2 require a significant amount of extra code and understanding of the underlying package from the user. Animating an algorithm with these packages requires numerous procedure calls in order to place the objects in the correct position and to actually do the desired animation. TANGO [Sta91] comes closest to allowing the user to design an animation by demonstrating it. This system provides the user with a visual design environment which allows him to create objects and the paths they are to follow. However, it also requires the user to define the transitions the objects make as well as requiring the paths to be associated with the objects.

Other packages, such as Animus [Dui86], by Robert Duisberg, provide the user with the ability to create objects which already have behaviors associated with them. This is a benefit, but as with TANGO, the user has to make numerous other menu choices in order to perform the actual animation.

By providing the user the ability to create animation sequences by demonstrating the animation, we can be sure that the animation produced is exactly what the user wants. The user can specify a transitional change in a data structure graphically by creating the data structure and the path required for the change. To create this transitional movement, the user need only know what data structure to use and what type of path it is to follow. The designer provides the user with all of the tools required to demonstrate this movement (Figure 1.1).

Two distinct phases are required to animate an algorithm with this package. First, the sample, or prototype, animator must be created and saved in the design

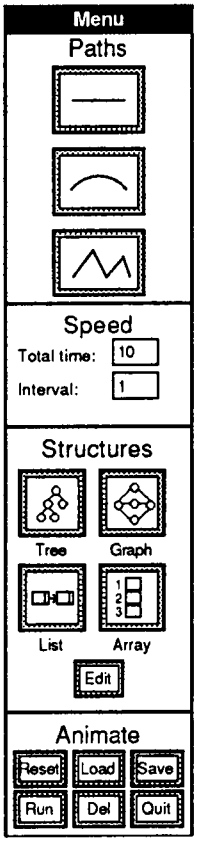


Figure 1.1: Main menu of animation designer. Includes data structures, paths and speed information.

environment (called the designer). The prototype animator is an encapsulation of the data structure and the movement. It contains information about the object to be moved and the transition it is to make. Second, the animator is manipulated by a program in a run-time environment. The user determines the actions of the animation when creating the original animation in the design environment.

All animators are created by the designer and can then be tested and when desired, saved to be used in the run-time environment. The designer provides the user with the elements required to create an animator: data structures and paths. Also provided are features, such as speed, which enable the user to edit the timing options of the animator as well as numerous editing options for data structures. The user, therefore, has the ability to create the structure exactly as desired. For example, if a sorting routine is demonstrated on an array, the user can edit the array so that it is horizontal or vertical and change the horizontal and vertical spacing if desired.

The data structures and paths created in the designer are used to build the animator and the way in which they are created plays an integral role in the design of the animator and the resulting animation. Once saved, the animator can be further edited in the run-time environment by changing the timing requirements or the objects to be moved (or both). Use of the animator in the run-time environment will be discussed in Chapter 4, The Programming Layer.

1.2 Background

This system was built with Garnet, which is a user interface development environment that provides a comprehensive set of tools allowing the creation of highly

interactive graphical interfaces [MDZ90]. Garnet is implemented in LISP and uses the X11 window manager. Of Garnet's numerous interesting features, most important to this animation system are its object-oriented programming system, its graphical-object system, its input handling, and its constraint system. All data and methods of Garnet objects are stored in "slots". These slots represent different properties of an object and can grow dynamically if so desired. For example, if a rectangle is created, the user can also specify the color of the rectangle by assigning a value to the :filling-style slot. Or, if desired, the user can create custom slots to store values that may be particular to an application. Slots can store either data values or formulas.

Constraints are relationships that are defined once and maintained by the system. Since any slot can contain a formula, the value of one slot of an object can be constrained to another. For example, if an object consists of a box and text, the text can be centered in the box by constraining its :left slot to the value in the :left slot of the box. Constraints can contain any LISP expression. The previous example would require the :left slot of the text to contain a formula which references the :left slot of the box using a combination of Garnet and LISP syntax. Because the :left slot of the text is constrained to the :left slot of the box, whenever the box is moved, the text is also moved.

Garnet's graphical-object system allows graphical objects to be easily created and edited. It also provides for drawing and erasing of objects. This system also ensures that when the property of one object is changed, any other objects that depend on it are refreshed to reflect that change.

All input-handling is handled through the use of Garnet interactors which encapsulate input device behaviors and allow the details of these behaviors to be separated from the application and from the graphics [MGD+92] . Interactors handle the events

that occur, such as a mouse click, and determine what types of behaviors are associated with those events. By putting restrictions on interactors, such as where they can start, or where they can be active, many different interactors can be used to handle separate input events. For example, when one item from the menu is chosen, a different action may be required, such as drawing an arc path instead of a line path. By manipulating the slots of the interactors, only the interactor that allows for an arc path to be drawn will be active, thus allowing for an arc path to be drawn. All interactors provided by Garnet come with their own requirements, such as the Move-Grow-Interactor which provides for a running-action, which determines what happens while that interactor is active. Other interactors may also have a final-function which determines what happens when that interactor becomes inactive after being active.

The animator that is produced as the result of the creation of an animation sequence by the user is an interactor. It encapsulates the behavior that the user has demonstrated in the animation sequence. It has many interesting slots and requirements which will be discussed further in Chapter 4. The use of constraints allows the user to create animators when knowing very little information about how the animator will later be used. The animator is created as a prototype that is specified by its original requirements (i.e. an object starts at the first object in a list and ends at the second one). Expensive calculations of some path information is also saved by using constraints. Constraints play an integral role in providing an environment which allows for the simple creation of algorithm animations.

1.3 Overview of Examples

Three example animations were designed in order to demonstate the usefulness of this package. They are: sorting a linked list by using insertion sort, depth first search of a graph, and insertion, deletion and traversal of a 2-3 tree.

The insertion sort example demonstrates a common algorithm and takes advantage of the Garnet aggrelist structure which helps to lay out list and array structures.

The depth-first search example shows how easy the animation of an algorithm can be. It shows that an algorithm can be animated with very little knowledge of Garnet and very few extra lines of code required to do the actual animation. It also demonstrates using an object to follow the paths of the data structure instead of the data structure itself.

The 2-3 tree manipulations represent a very complex algorithm. The algorithm containing the animation code requires knowledge of Garnet aggregraphs which help to lay out tree and graph data structures. This example exploits all features of animations created with the system, and demonstrates moving objects, as well as elements of the data structure, along designated paths.

1.4 Goals

The main goal of this animation package is to allow the user to create animations easily with little to no knowledge of the underlying system requirements. The user should be able to create an animation without complex calls or knowledge of how the animation package relates to the code begin written, thereby requiring extra code from the user. We also aim to provide an easy to use tool which allow the user to see the animation as it is being created, thus providing the option to change the

animation as needed. Another aim is to provide the user with the freedom to design the paths that objects are to follow, as well as the corresponding data structures that contain the objects.

The animations created with the design package should be usable in independent Garnet applications. Because the creation of a data structure in the design environment does not affect the run-time version of the animator, the user should not be dependant upon the design environment to use animations. As a result, the entire data structure need not be created, as long as the movement required is demonstrated.

In summary, the goals are:

- Create an animation by demonstrating the objects and transitions.
- Create portable animations.
- Allow flexible design of data structures and paths.

1.5 How the Goals Were Achieved

The many editing options provided by the visual editor allow the user to create data structures and paths to fit his needs. A transition of an object is demonstrated by creating an object and creating a path that the object is to follow. This path, as well as the object, is editable. As a result of creating the animation sequence, the transition of the object defines the behavior of the animator. Additionally, an animation can be tested before it is saved to be used in an algorithm. This allows the user to fine-tune the animation to make sure that it is exactly what is needed.

The actual animator interactor created through the use of the options provided by the visual editor relies on the availability of constraints provided by Garnet. By using

constraints, the animator can adhere to the original requirements even when the objects operated upon change. Because animators can be used repeatedly by changing certain properties, such as the object to be moved, they are not only flexible, but also portable. The animator's behaviors define actions to be performed but do not restrict the objects these actions can be performed upon. This allows any animator to be used with any of the data structures provided and in any Garnet environment.

Interactors provided by Garnet were the other important elements in fulfilling the goal of creating a flexible design environment. The flexibility of these interactors provide the user with the capability to create and edit the paths and data structures necessary for any algorithm animation.

Chapter 2

Related Work

As time goes on, the amount of work required from the user by algorithm animation systems decreases and the systems become easier to use. The methods by which the most recent algorithm animation packages work vary, as does the amount of work required from the user. One important difference with this system is the flexibility provided by the drawing editor. Previous algorithm animation packages provide for the creation of paths but not the necessary data structures. This package provides for the creation of both the paths and data structures, as well as extensive editing options, thus giving the user the ability to create the data structure which is appropriate for the algorithm.

Another major difference between our system and other existing systems is the amount of work required in order to create an animation. The steps required for swapping two objects with our system involve creating the object, such as a list, and creating paths that the elements of the object are to follow. To swap two elements, two paths would be created, one for each element. In the design environment, the animation can be directly run. In the run-time environment, additional lines of code

need to be added to the algorithm to indicate the objects to be moved and to actually start the animator. To demonstrate the difference between our system and other existing animation systems, the steps required to swap two elements are outlined for each of the following algorithm animation packages.

2.1 Animus

The Animus system [Dui85] provides the user with three different ways to create an animation. One of the ways is through a visual editor in which the user makes choices from a menu in order to create all of the parts of the animation. The other two methods involve coding in Smalltalk using ThingLab and adhering to the Animus constraint syntax.

With its visual editor, the user can create an animation without writing any code. However, if the user requires an object or behavior that is not supplied by the designer, this is not possible. The visual editor supplies the user with numerous objects, which have behaviors already attached to them through the use of constraints. In order to perform a sorting algorithm on a queue, the user must begin by choosing the SortQueue object from the animation toolkit. This object can be edited by adding new elements with varying magnitudes, thus changing the value of that element in the queue. The queue is sorted by sending messages to it. This is done by making several menu choices which involve selecting the type of object to receive the message as well as the object itself. From there, the system checks to see what types of messages the queue object can receive and prompts the user to select the message desired from the list.

The code which does the actual sorting contains no animation information whatsoever; it is exactly the same as if the code was not being animated. This is accomplished by requiring the user to supply additional constraints dealing with the comparison and swap of the objects. The animation is therefore built by stating constraints outside of the algorithmic code and the tasks of algorithm design and animation are separated [Dui86]. Because Animus provides objects with behaviors already associated with them, the transitions between objects and paths never need to be specified. The constraint mechanism provided by the system handles all of the transitions required for an animation. However, this limits the user to paths created by the system. There is no freedom in designing the path that the objects are to follow. Additionally, because the objects are provided without learning the underlying system, the user is limited to those objects.

Since Animus, Duisberg has created an experimental algorithm animation system aimed at allowing the user to create animations 'without ever having to write graphics code' [Dui87]. Creating an animation involves creating objects and paths and associating paths with objects by graphical gestures. This makes an animation easier than his previous system but still requires many menu selections as well as interactions with several editing facilities, thus adding some complexity.

2.2 BALSA

The BALSA-II system [Bro88] provides the end-user with a vast variety of algorithms and data structures upon which to perform them. The user can choose an algorithm and data structure and execute the animation, then change the data

structure and execute the animation again. It is also possible to have two views of the same algorithm animated upon different data structures at the same time. This flexible environment is incredibly useful for the end-user but involves a great deal of coding for the algorithm animation programmer.

Creating a sorting routine involves several steps: preparing the algorithm, writing input generators, then creating the views to be used with the algorithm. Preparing the algorithm requires annotating, which involves 'inserting interesting event markers', and structuring which handles all interactions between BALSA-II and the algorithm, such as initialization [Bro88]. The input generator handles the input for the algorithm and must be able to handle changing the input at runtime. A view consists of a modeler, a renderer and an adapter. The modeler 'creates an abstract representation that the renderer draws in the view window' while the adapter is used to transform the data so that the output events correspond to the information the modeler and renderer need.

All components of the algorithm animation become part of a library so that they can be reused. This provides for an algorithm animation to be easier to create the next time. While the end product is quite impressive to the user, getting there involves a steep learning curve for the programmer. Creating an algorithm animation with BALSA-II requires code only without the help of a graphical menu to allow a new user to create an animation easily.

2.3 TANGO

The TANGO system [Sta91] provides the user with two different ways to create animations: through an animation designer (DANCE - Demonstration ANimation CrEation) and by using only code. Animations created through DANCE can be used as input by the TANGO algorithm animation system [Sta91]. Both ways require three steps: annotate the program with a set of abstract algorithm operations, design specific image display and animation scenes and map the algorithm operations to the animation scenes [Sta89]. With the visual designer, these steps are made a lot easier through menu choices. The main difference between our visual editor and TANGO's is the amount and kind of choices required to create an animation.

TANGO requires that the structure, the transition, the path and the associations between objects be made before an animation can be created. With our visual editor, objects, paths and the associations, or relationships, between them are implied by their location in relation to each other. For example, to animate the swapping of two objects with TANGO's visual editor, the user creates the objects, chooses the transition type (move for example) and creates the transition. The path must then be created and associated with the transition. With our visual editor, the user creates the object and attaches the path to it. The path is therefore associated with the object and no further menu choices need to be made.

By using associations in TANGO, the user is allowed to generalize an animation. This allows previously created animations to be used, regardless of the current objects in the animation scene. This is another aspect of TANGO that is similar to our package. The idea of making an animation an entity independent of the current state of the visual editor, or run-time environment, gives the user a much greater amount

of freedom in the use of the animations.

The same three steps need to be followed to create an animation without the aid of the visual designer. However, this requires coding in C and TANGO package calls. Every object involved, as well as all paths, transitions and associations, need to be created by the user through calls to the appropriate functions. This allows the more experienced programmer to create animations without the visual designer but it leaves the novice programmer or user with the difficult task of learning the requirements of the TANGO package calls as well as the C code required to embed these calls.

2.4 ZEUS

The ZEUS system [Bro91] also uses the 'interesting event' approach and requires the user to use function calls in order to create an animation. Each of the function calls, or events, require parameters, such as the object and the number of elements. ZEUS provides the user with a control panel from which the algorithm, view and the data for the algorithm are chosen. Additionally, a general purpose editor for editing objects as well as 'some library procedures to interpolate changes of object parameters over time' are provided [Bro91] . One similarity between ZEUS and this system is the ability to write current states to a file and restore the state by loading the previously saved file.

Animating a sorting algorithm with ZEUS requires that the user include numerous event calls in the algorithm. These calls include the code required to create the data structure, initialize the values and do all of the swapping. Another similarity to ZEUS is that it also allows the user to test animations as they are being developed, thereby

developing an animation without actually implementing the algorithm.

Chapter 3

Examples

Three examples were created to demonstrate the utility and flexibility of this package. Each example demonstrates a useful algorithm. The three examples are: insertion sort, depth first search and manipulations of a 2-3 tree. For each of the examples, a data structure, the appropriate path(s) and any other objects necessary for the animation were created in order to create the prototype animation.

3.1 Insertion Sort

Insertion sort is a sorting technique where the *ith* element of a list is put in its correct place on the *ith* pass through the list. For example, the third element of any list will be in its proper place after the third pass through the list. For this animation example, the swapping of two objects in a list was first created as an animator and saved. This prototype swap consisted of a list and two arc paths (Figure 3.1). The actual specifics of the list are not important to the creation of the animation. The

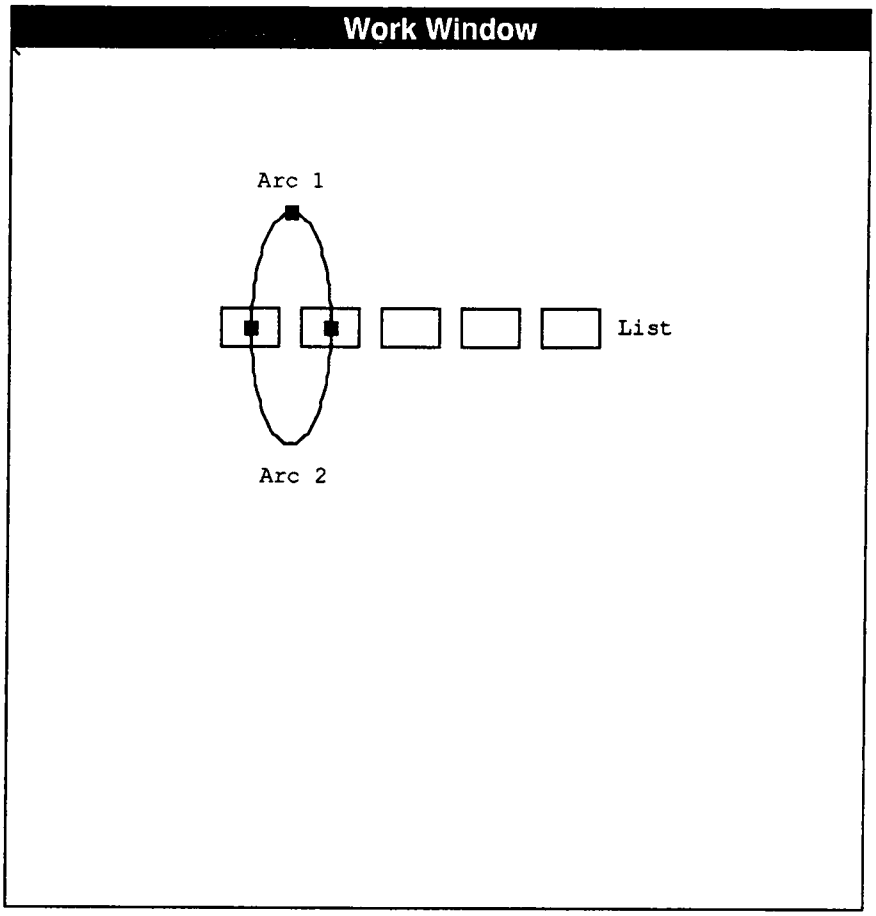


Figure 3.1: Design of swap animation. Includes one list and two arc paths.

demonstration of the exchange of the two objects, regardless of the number of elements in a list, etc., is what is important to the animation creation.

The user demonstrates the exchange of the objects by drawing paths between them. Figure 3.1 shows the design of the swap animation. There are two ways that an object to be moved may be associated with a path: 1) the object does not belong to the data structure but physically intersects the path (i.e. a box that moves between two nodes of a graph in depth-first search); and 2) the object belongs to the data structure and the path originates either in or on the object. When there are two or more conflicting objects that meet the criteria, the designer chooses the object that intersects the path that does not belong to the data structure. In Figure 3.1, Arc 1 originates in the first box in the list and even though the path is also connected to the second box, it is the first box that is moved along the path. By including Arc 2, which originates in the second box, it will also be moved. Because each path ends where the other originates, the visual effect is that of swapping the two elements. When the animation is executed, the objects move smoothly along the designated paths. Because the animator created is a prototype, it can be used repeatedly for the swapping of all necessary items by merely manipulating the values of the animator.

Figures 3.2 - 3.6 demonstrate that the objects follow paths similar to the paths created by the designer. The objects being swapped do not have to be the same distance apart as the objects originally created by the designer. The objects can be closer or farther apart and the associated paths will be scaled to achieve the shape of the paths created by the designer.

The animator created for this algorithm can be used for any sorting algorithms. For example, it can be used with bubble sort, which sorts by placing the lowest key value in the first position by swapping it with elements above it until it is in the

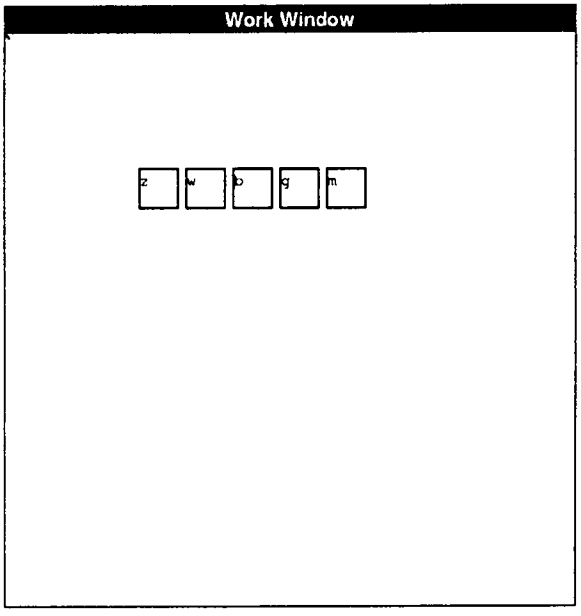


Figure 3.2: The list before any of the elements are sorted.

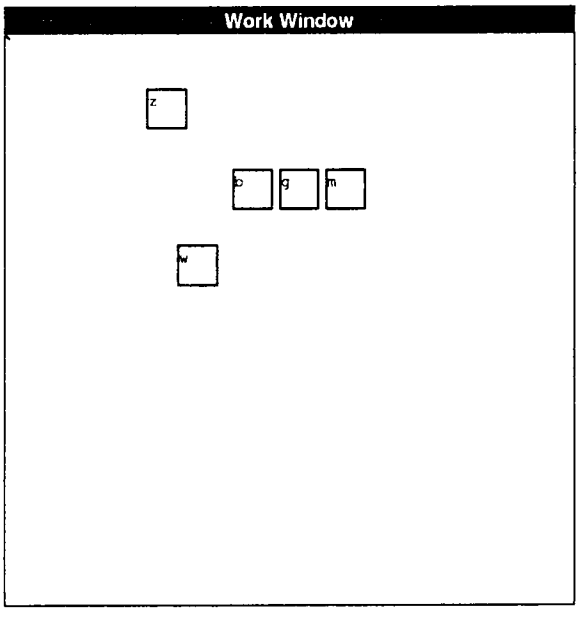


Figure 3.3: Swapping "z" and "w" in order to put "z" in its correct position.

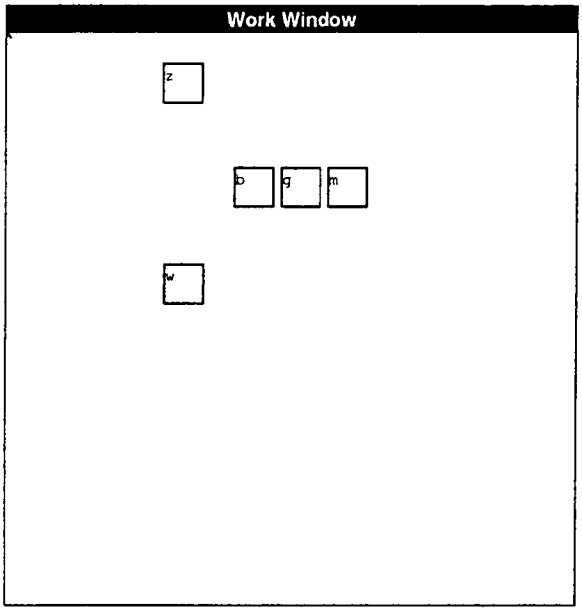


Figure 3.4: Swapping elements "z" and "w".

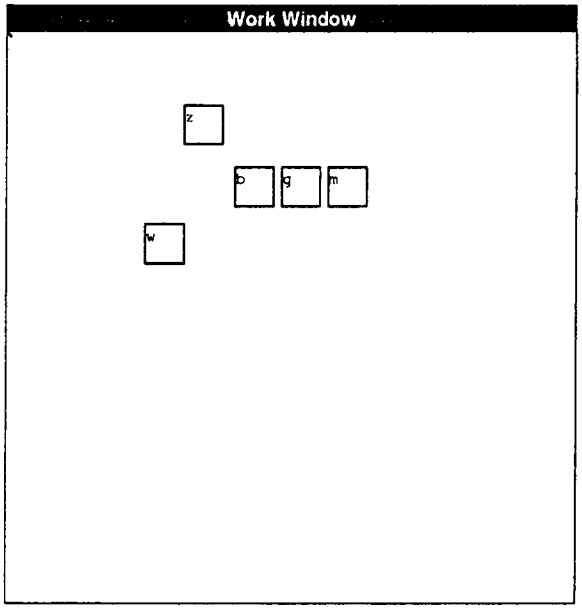


Figure 3.5: Swapping elements "z" and "w".

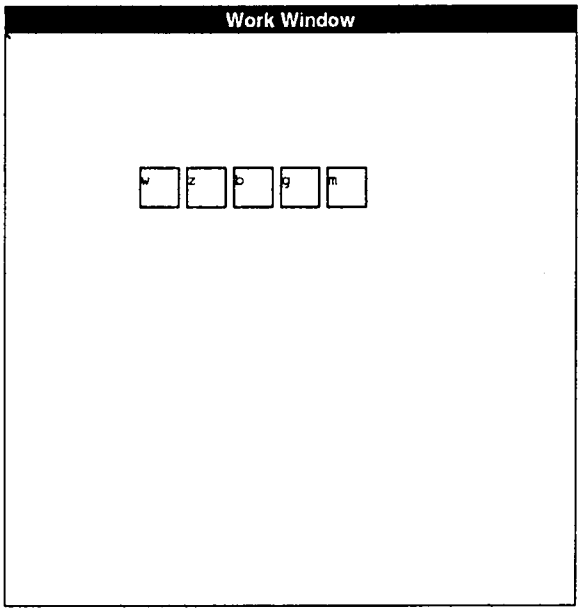


Figure 3.6: The list as it appears after the first swap has completed.

first position. Repeated passes are made through the list in this fashion, gradually "bubbling" up into thier proper position [AHU83]. Thus for each element, many swaps may be required.

For example, element "3" of the list in Figure 3.7 would be swapped with "17", then "22", then "10" to put it in its proper position.

3.2 Depth First Search

A depth first search of a graph begins with each vertex marked unvisited. A vertex is chosen as the starting vertex and is marked as visited. Each of its vertices are in turn searched using depth first search recursively and marked as visited as the search progresses. Once all vertices adjacent to the starting vertex are marked as visited,

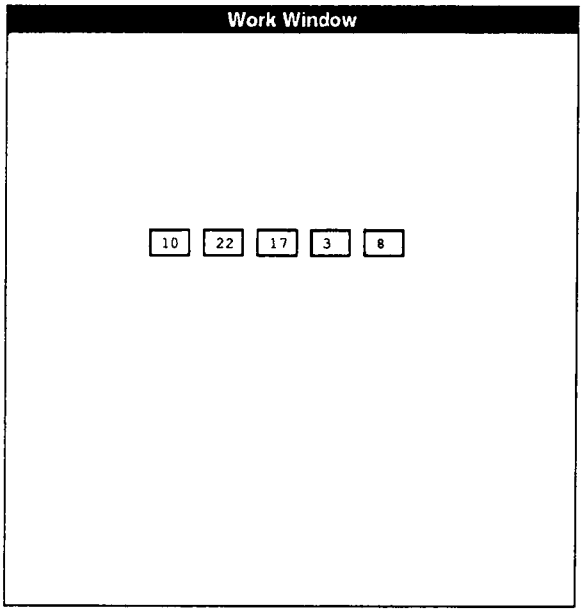


Figure 3.7: List to be used for bubble sort.

the depth first search for that vertex is complete. If any vertices of the graph are left as unvisited, then an unvisited vertex is marked as the new starting vertex and a depth first search begins from there. Because the depth first search is performed recursively on each vertex as it is visited, the search continues deeper in the graph as long as possible, hence the name depth first search [AHU83].

The depth first search algorithm animation proved to be the most simple to develop with this package. The animation that was created by the designer consisted of a graph, a polyline path that connects the node that is currently being visited with the next node to be visited and an object to move along the path. Since a depth first search merely involves movement from one node to the next, the designer needed only to demonstrate movement from one node to the next. This allowed for the graph created by the designer to be very simple. The layout or complexity of the graph is not important since the graph used in the design session does not have to be the

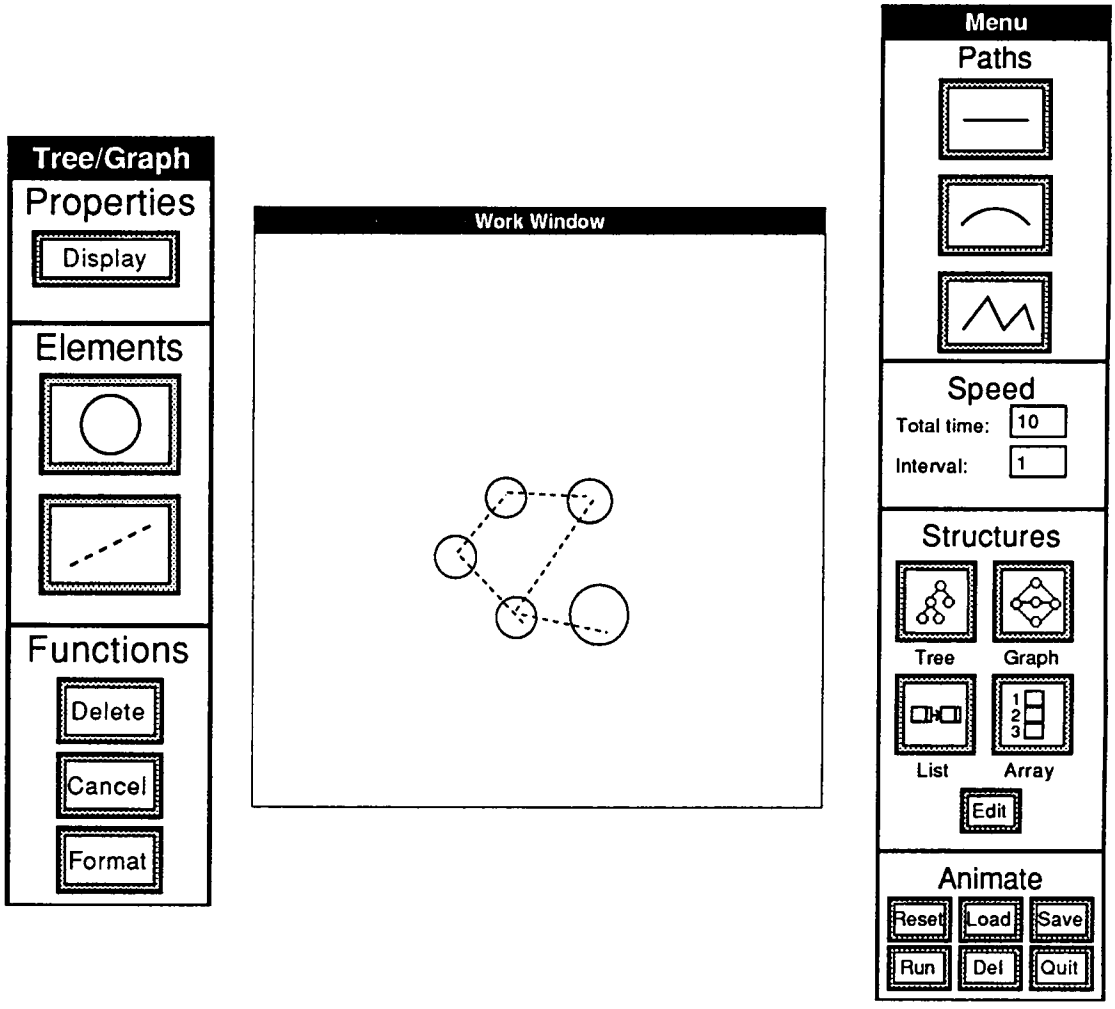
graph used in the algorithm animation.

To demonstrate the depth first search, the graph was first created. By selecting the graph from the Structures portion of the main menu (Figure 3.8 (c)), the tree/graph editor is displayed (Figure 3.8 (a)). This editor allows creation of nodes and edges by drawing circles and lines that link them together (Figure 3.8 (b)).

Once the format function is chosen, the circles and lines are replaced by nodes and edges and the graph is drawn. The path can then be drawn by selecting the polyline from the main menu and using the nodes of the graph as origination and ending points. The object to move between the nodes can be any Garnet object and can be loaded through a file or created interactively. When the object is positioned to intersect with the path, the animator will use that as the object to move instead of the element of the data structure from which the path originates. Figure 3.9 shows the complete design of the animation.

The nodes are colored as they are visited. The code that animates this searching algorithm includes the traversal back to the starting vertex once all adjacent nodes have been visited. This is done so that the object moves smoothly between the nodes of the graph. Figures 3.10 - 3.13 show traversal to the first node.

Any type of tree traversal can also be animated. Again, it is up to the user to determine the type of animator actions desired. The user has the freedom to determine the paths, therefore determining the actions of the objects to be animated.



(a) (b) (c)

Figure 3.8: Creating the Depth First Search Graph.
(a) The tree/graph editor, node and edge options are used to design the graph.
(b) The design of the graph.
(c) The main menu, graph option used to display the tree/graph editor.

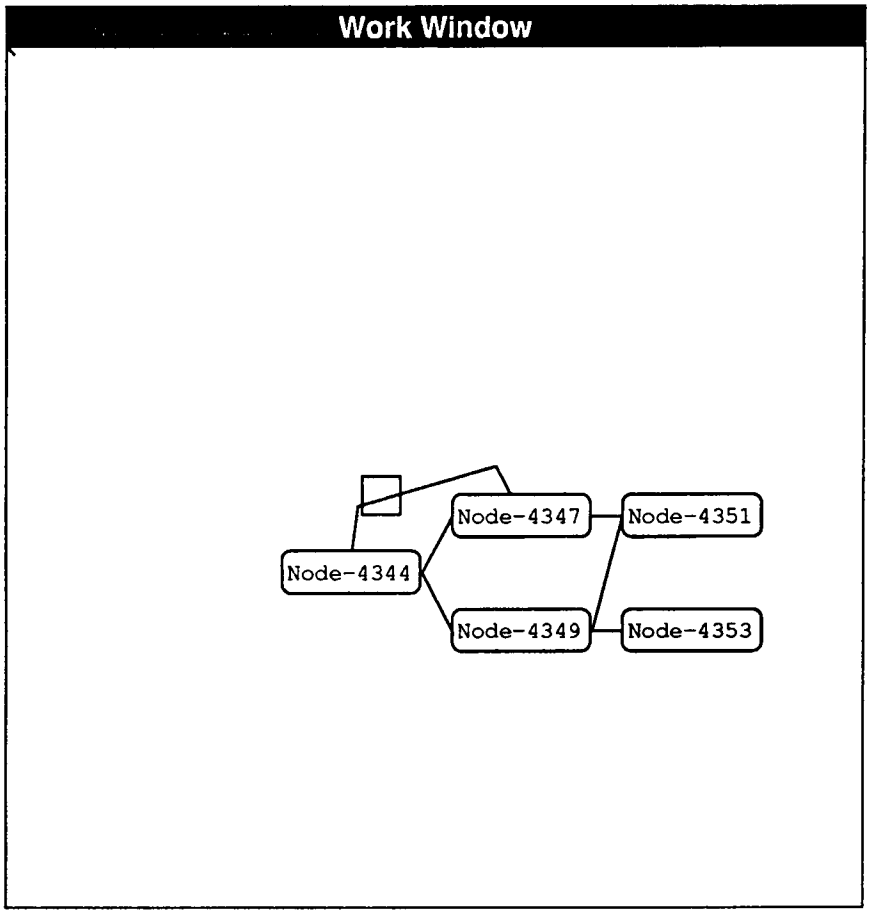


Figure 3.9: The design of the traverse animation includes an object to move along the path, the graph and one polyline path.

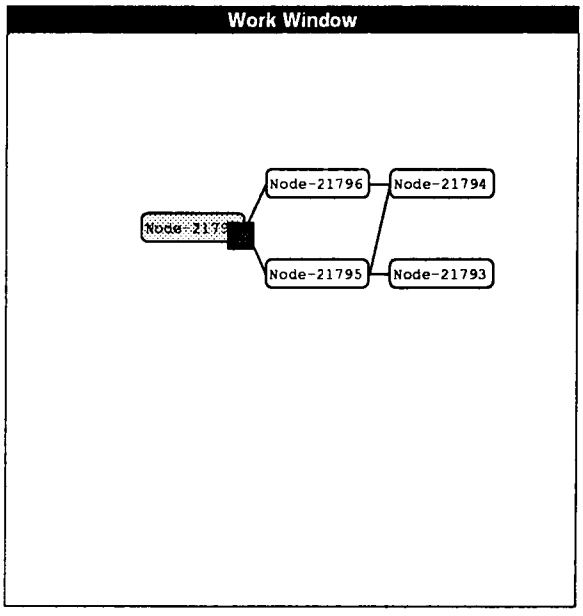


Figure 3.10: The root node is marked as visited and the object begins to move to the first adjacent vertex.

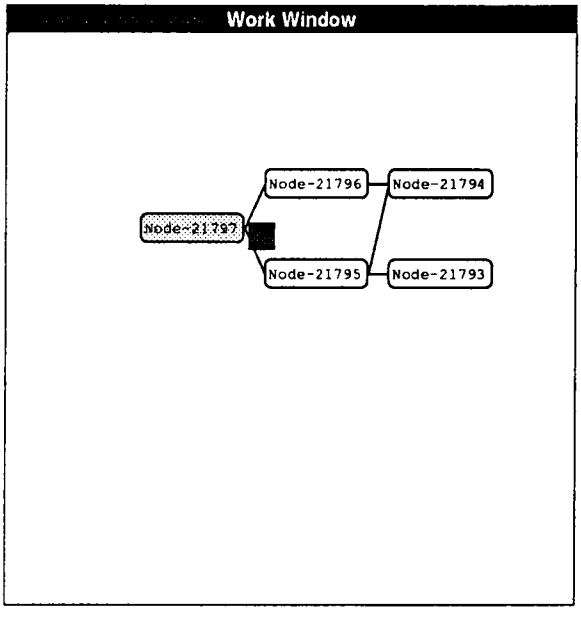


Figure 3.11: Object continues moving toward first adjacent matrix.

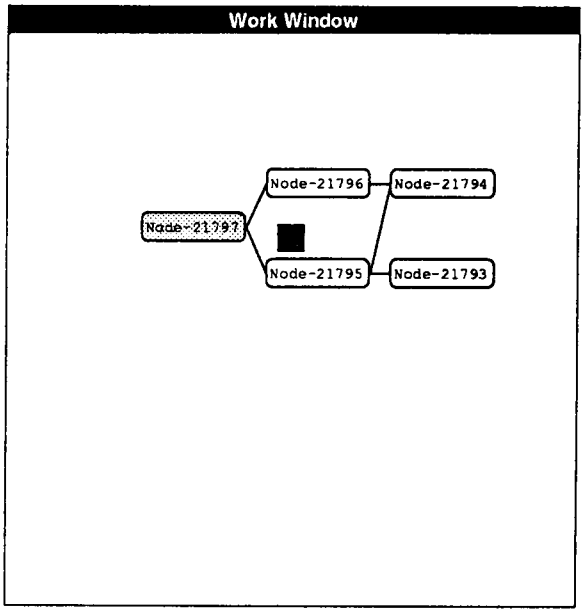


Figure 3.12: Object continues moving toward first adjacent matrix.

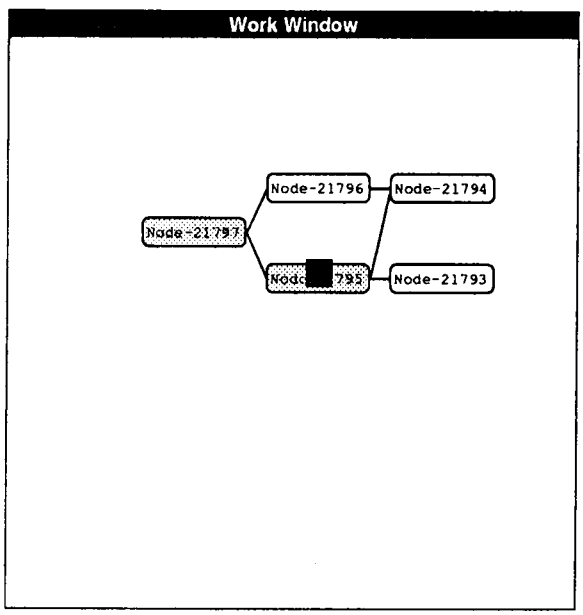


Figure 3.13: Traversal to the first adjacent vertex is complete. It is marked as visited.

3.3 2-3 Tree Insertion

A 2-3 tree is an ordered, balanced tree where every node has either 2 or 3 children or is a leaf node and has no children. In the insertion algorithm used, the nodes are visited recursively until the correct place for the new node is found. If that new position results in a node having four children then the parent node must be split so that each resulting child node has only two children. This may cause its parent to have four nodes also, in which case its parent may need to split into two nodes of two children. This splitting of nodes may continue recursively up the tree until the root is reached.

The actual animation for this algorithm is not as intuitive as the others. It is not as easy to imagine what the user actually needs to see to understand the algorithm. The animation we devised demonstrates the traversal of the 2-3 tree to determine the position of the new node. It also shows the creation of new nodes and links which are necessary to balance the tree.

For the traversal of the tree, the traversal animator used for the depth first search is used. This animator was initially created in order to demonstrate the movement between two elements of a data structure. In the depth first search, an object is moved between the elements. In this animation, the new node can be moved between the elements of the tree as it searches for its new position in the tree. For creating and adding new parent nodes, the same animator is used. The new node is created and is moved from its original position to its new position using the traversal animator. Figures 3.14 - 3.20 demonstrate insertion of a new node.

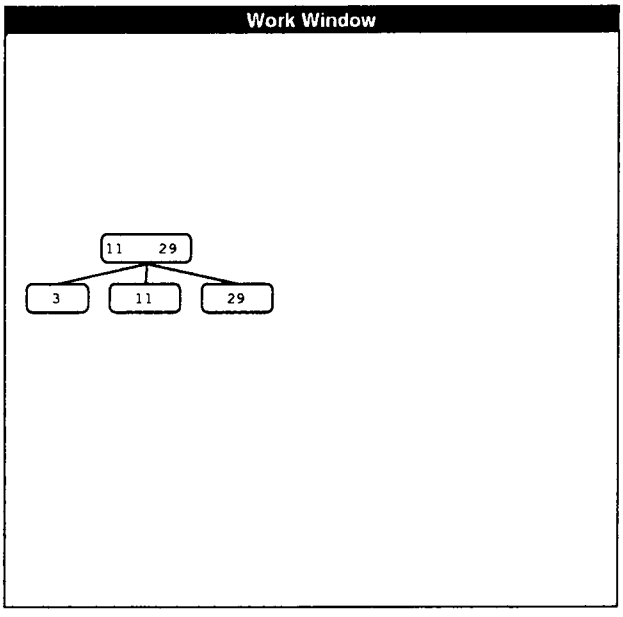


Figure 3.14: The 2-3 tree before node "17" is added.

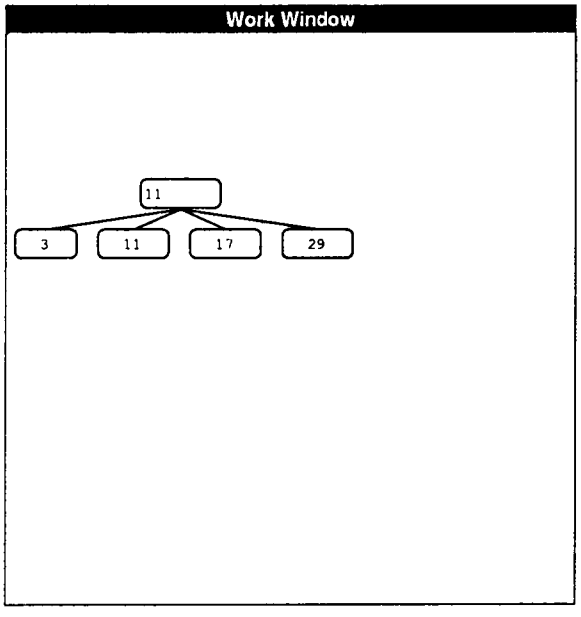


Figure 3.15: Inserting node "17" causes its parent to have too many children so it must be split.

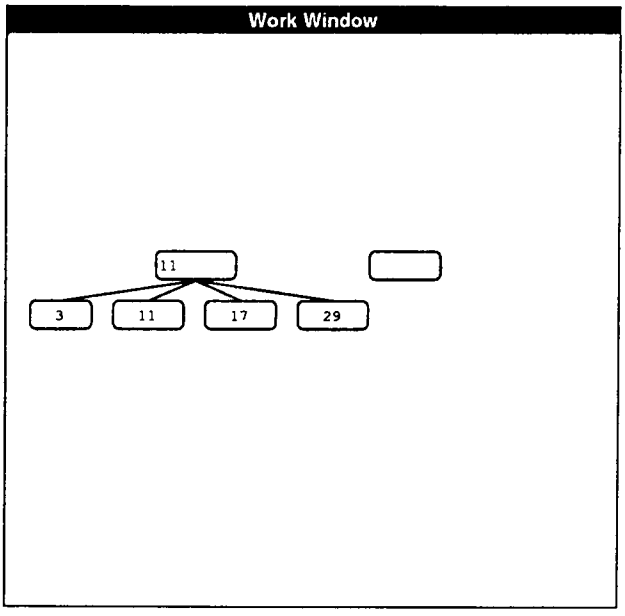


Figure 3.16: A new parent node is created.

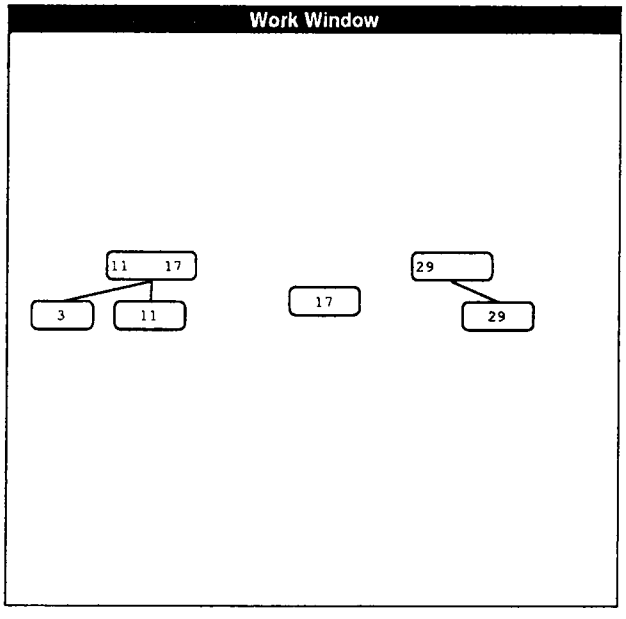


Figure 3.17: Moving node "17" to become a child of the new node.

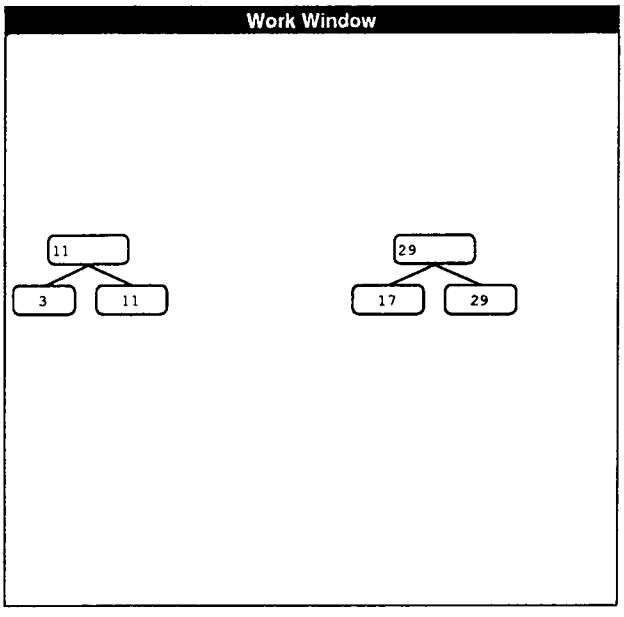


Figure 3.18: Both "17" and "29" have been moved to the new parent node.

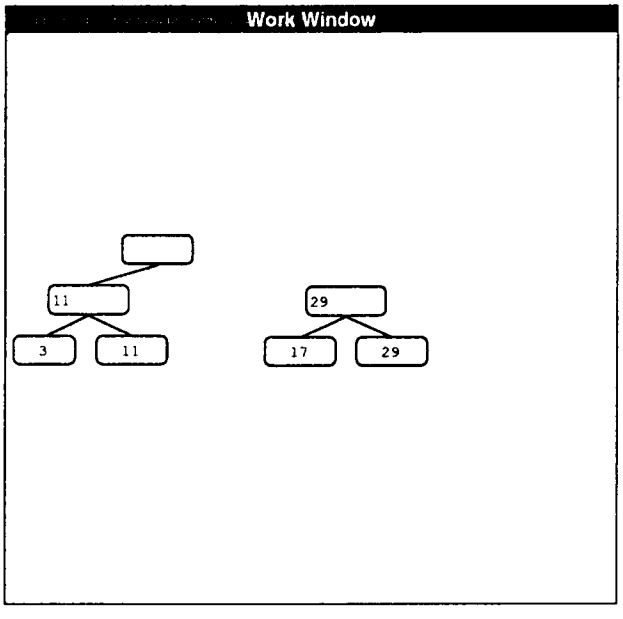


Figure 3.19: A new root is created and the first child is added.

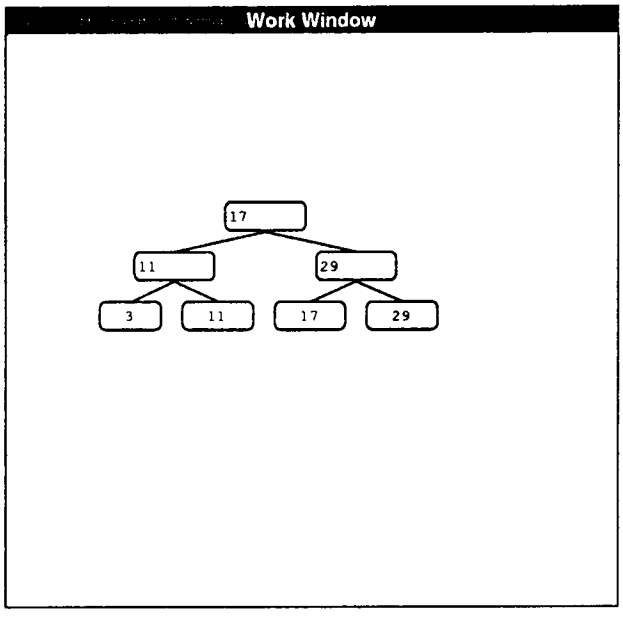


Figure 3.20: Second child of the new root is added and the insertion of node "17" is complete.

Chapter 4

Programming Layer

The programming layer supports the creation and use of paths, data structures and most importantly, the animator interactor. The animator interactor is comprised of many slots, which are attribute-value pairs [MGD+92]. The name of the slot, such as :path-info, indicates the attribute while the value indicates the value for that attribute [MGD+92]. Any slot may contain a formula as its value, which usually contains a dependency on another value. The formula can contain any LISP expression and reference any of the other slots (which would then constitute a dependency). By including dependencies on other slots of the animator, constraints are created, that is, the value of one object is constrained to the value of another (or others).

The animator interactor contains numerous 'interesting' slots which determine the way in which it works. Several of the slots contain formulas that depend on things such as the number of objects being moved or the type of paths the objects are to follow.

The main features of the animator interactor are the modifiable timing function and the provision of paths. The timing function is called at each time increment and

updates the positions of the objects being moved. The time increment is determined by the timing values entered by the designer. Many of the slots of the animator are used to determine the new positions of the objects at each increment.

Of the paths created, the arc path requires unique treatment. It has many slots that neither the polyline or line have which are vital to some of the slots of the animator interactor. Because arc paths must be distinguishable from other paths, additional information must be kept for each path in order to make the distinction.

4.1 Internal Creation of Paths

4.1.1 Path Slots

All paths are internally represented as polylines to allow for a variable number of point pairs. They, therefore, require a slot (arc-p) to distinguish between paths that are actual polylines and those that are arcs. The value in this slot determines how many selection boxes appear when a path is selected in the designer. For an arc, three selection boxes always appear but for a polyline, a selection box appears for every point pair in the point list. This slot is also used when the path-info slot is computed, discussed further in section 4.2.2, in order to determine what type of information is necessary (since different information is required for lines, polylines and arcs).

Arcs contain numerous other slots that the other two types of paths do not. These include the angle deviant from the origin ϕ, the height of the arc b, and half the distance between the end-points a. This information is used when the path-info for the animator is computed which in turn affects the path.

Every path that is created is also assigned timing information. Two slots contain

this information. They are: total-time and timer-repeat-wait. The information in these two slots is used by the animator in determining the number of times to update the positions of the objects being moved. Total-time is the number of seconds the animator-interactor should run and timer-repeat-wait is the interval in seconds between each move.

4.1.2 Polyline and Line Paths

Polyline and line paths are straightforward in their creation. They contain point pairs to identify the end-points (and segments points for polylines). When a path is created, its endpoints (or segment points for polylines) are contained in the point-list slot.

For the polyline path in Figure 4.1, the point-list contains the following list of point pairs:

$$((111, 290), (95, 243), (147, 232), (132, 201), (168, 180)). \tag{4.1}$$

4.1.3 Arc Paths

The unique slots of an arc mentioned previously are derived using the following formulas:

$$a = \frac{\sqrt{(\Delta x)^2 + (\Delta y)^2}}{2} \tag{4.2}$$

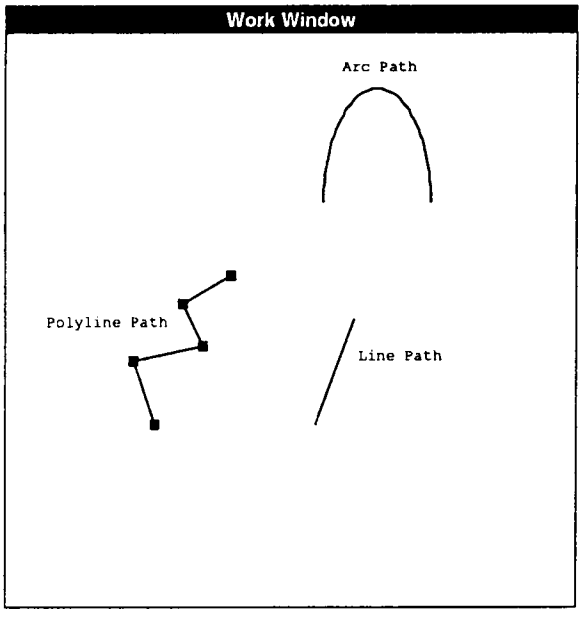


Figure 4.1: Example of each type of path. Polyline path is selected.

Δx and Δy are the distances between the respective coordinates of the end-points of the arc. a is computed by taking half the distance between the end-points of the arc.

$$b = a \tag{4.3}$$

When an arc is originally created, its a (half of the distance between the end-points - see Figure 4.2) and b (height of the arc - see Figure 4.2) parameters are given the same value. These values create an arc which is essentially a half-circle. a and b can be changed either directly or by editing the arc in the designer.

$$\phi = tan^{-1}(\frac{\Delta y}{\Delta x}) \tag{4.4}$$

Phi (ϕ) is the angle that the arc deviates from the origin (Figure 4.3). This parameter is required when an arc's end-points are not along the same axis. This value

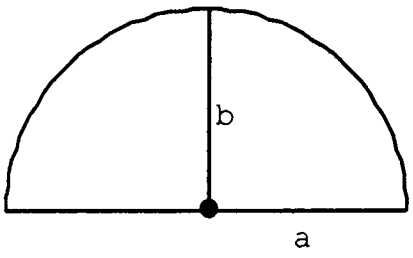


Figure 4.2: a is half the distance between the end-points and b is the height of the arc.

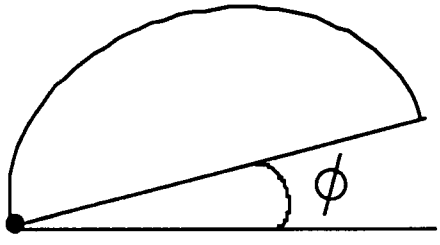


Figure 4.3: Phi is the angle that the arc deviates from the origin.

is saved as one of the slots for the arc.

The arc path requires additional formulas to generate its original point-list (which draws out the actual arc in the designer) as well as the path slot required for the animator. When an arc is originally created, it is assigned 51 point pairs. It is rotated according to its ϕ and translated to ($centerx$, $centery$) (since the arc is originally drawn around the first x,y coordinates) using the following transformations [FVF+90]:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} centerx \\ centery \end{pmatrix} + \begin{pmatrix} cos\phi & -sin\phi \\ sin\phi & cos\phi \end{pmatrix} \begin{pmatrix} acos\theta \\ bsin\theta \end{pmatrix} \tag{4.5}$$

$centerx$ and $centery$ are the point where b intersects the base of the arc and a, b, and ϕ are the parameters of the arc which in turn, are used when creating the animator.

θ begins as π for a left to right arc and 0 for a right to left arc, and is incremented by $\frac{\pi}{NumPoints}$ each time through the loop. Thus, an aesthetically pleasing arc is produced.

4.2 The Animator Interactor

The animation interactor provided by the programming layer builds on the basic, no frills Garnet animator-interactor. The main features of the Garnet interactor are: timing options and a modifiable timing function which is called at every time increment. Animator interactors can, therefore, be tailored to the application.

The animators created with the programming layer use numerous additional slots (besides the timing information and timing function). The path and the attach points of the path are two examples of slots that play an integral role in the creation of the animator. These slots are initially defined by the programming layer when the user creates the animation. The slots which contain formulas are re-evaluated as the values that they depend on change. By using constraints in the slots of the animator, the animator can be changed and used repeatedly by changing only a few key values. For example, in the depth first search example, the same animator is used for each traversal by changing the information in one slot - the slot that points to the objects that the path originates from and terminates at. Because other slots of the animator depend on the object that is being moved, the values of these slots will change as well.

Numerous objects and paths can be associated with one animator. Most path and object related slots are created as parallel lists which allows them to be distinguished from each other. For example, each object that is to be moved will have its own path information. So if there are three objects to be moved, there will also be three sets of path information. For each object to be moved, there exists information about it in most of the following slots.

4.2.1 Object Related Slots

Several slots of the animator are used to determine the paths the objects are to follow. The obj-to-change slot contains the actual object(s) that follow the path(s). This object can be an element of the data structure or an object independent of it. When the animator is created, the paths associated with it are examined to determine

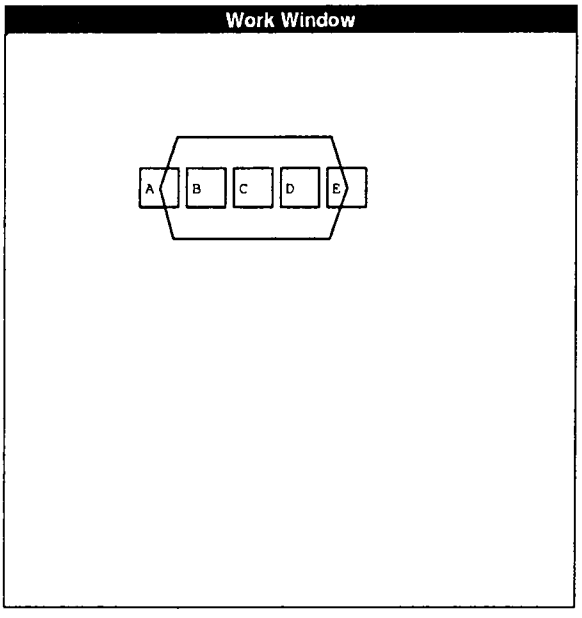


Figure 4.4: This animator's obj-to-change slot would contain two items: (ListItemA, ListItemE).

the contents of this slot. If an object intersects a path at any point other than the origination or termination points of the path (these points must lie within objects that belong to the data structure) then that is the object to be moved. If there is no such object, the element of the data structure where the path originates is the object to be moved.

For each path defined by an animator, there is an object in the obj-to-change slot. Figures 4.4 - 4.5 help demonstrate how this slot is set. Figure 4.4 does not have any objects, other than the data structure elements that intersect the paths, therefore, the actual elements of the data structure are moved. Figure 4.5 shows objects intersecting each of the paths of the animator, therefore, these are the objects to be moved. It is also possible that the animator might move one element of a data structure and one non-related element.

When an animation is run, the anim-obj-to-change slot is evaluated to see if it

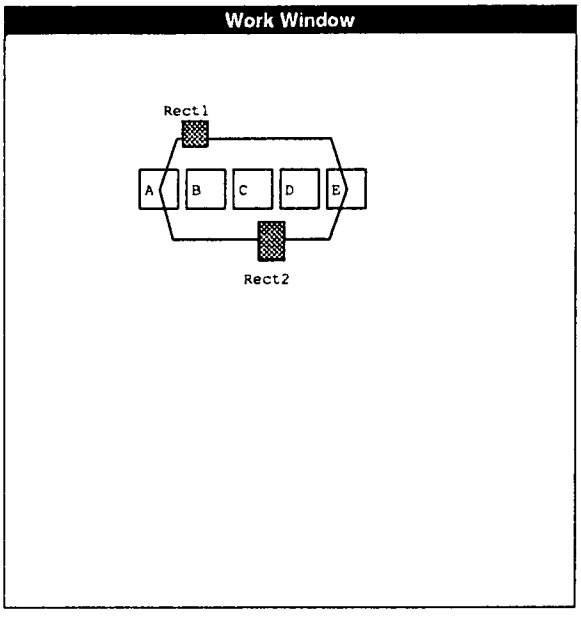


Figure 4.5: This animator's obj-to-change slot would also contain two items: (Rect1, Rect2).

contains any objects. If it does, then these objects identify the relative starting and ending points of the objects in the obj-to-change slot. If there are no objects in this slot, then the objects in the obj-to-change slot are used to compute the starting and ending points of the objects to be animated. For example, the sorting example animates elements of the data structure so its anim-obj-to-change slot is not used in evaluating the starting and ending points of the elements. Instead, the obj-to-change slot is used. The depth-first search, on the other hand, moves a box between nodes of the graph. Its anim-obj-to-change slot therefore identifies the nodes that the box moves between, and may be written as follows: (node1 node2). In general, the anim-obj-to-change slot will be empty when the animator moves elements of the data structure, and will contain elements of the data structure if an independent object, such as the box in the depth-first search example, is being moved.

4.2.2 Path Related Slots

Paths play an integral role in the creation of the animator, and therefore, there are several path related slots in the animator.

The path-info slot contains a list of the specifics of each path of an animator, such as the type of path (POLY, LINE, or ARC) and the important information about each path that is needed to create the point list. For a polyline or line, the only information needed to build the point list is the original point list and thus the original point list is the only other information stored in this slot for these two types of paths. The path-info that would correspond to the polyline path in Figure 4.1 would contain this list of points as well as the identifier :POLY which indicates that it is a polyline path:

$$(: POLY (111, 290), (95, 243), (147, 232), (132, 201), (168, 180)). \tag{4.6}$$

The path-info information for a line path would contain only two point pairs and the :LINE identifier. For an arc, the path-info slot includes: ϕ (the angle that the arc deviates from the origin), a (half the distance between the end-points), b (height of the arc), the point where b intersects the base of the arc, and the arc's origination point (Figures 4.2 and 4.3 and 4.3).

All of the necessary information for an animator's path-info slot is gathered by evaluating the slots of the paths that comprise the animator, (e.g., the a, b and ϕ of an arc path). This information is then used to create the path that the object is to follow when the animator is executed. If a programmer were to create an animator without the benefit of the designer, the programmer would have to manually compute the path information and explicitly assign it to the path-info slot.

Upon execution, the animator needs to compute the points on the path to which

the object will be moved during the animation. The path slot contains this set of points. The number of point pairs in this slot is determined by the timing information supplied by the designer. For example, if the total time for the animation is ten seconds and the time interval is one second, the number of point pairs in this slot will be eleven (the starting point plus the ten points the object is moved to).

A different formula is used to compute the path points for each type of path. Each is contained in a loop in order to produce the correct number of point pairs. For the line path the following equations are used:

$$x_{increment} = (x_2 - x_1)/(NumPathPoints - 1) \tag{4.7}$$

$$y_{increment} = (y_2 - y_1)/(NumPathPoints - 1) \tag{4.8}$$

$$x = x_1 + x_{increment} * counter \tag{4.9}$$

$$y = y_1 + y_{increment} * counter \tag{4.10}$$

The *counter* variable is the loop counter which controls the number of point pairs generated.

For polylines, it is not quite as simple. Numerous additional equations are required in order to preserve the distance between each of the points on the path.

The computation of the point list is begun by calculating the total distance of the polyline:

$$distance = \sum_{i=1}^{NumPoints-1} \sqrt{\Delta x_i^2 + \Delta y_i^2} \tag{4.11}$$

Δx_i and Δy_i represent the x and y distance between the endpoints of the ith line segment on the path and $NumPoints$ represents the number of points associated

with the polyline. The distance between each of the path points to be generated is then computed:

$$PointDistance = \frac{distance}{NumPathPoints - 1} \tag{4.12}$$

We subtract one from the number of points because with fifty points, for example, we actually have only forty-nine segments between the points that are connected by the path.

Two distances are used to determine whether or not we have reached the end of a path segment. One is the local distance, which is the distance from the origin of the segment to the last point we have computed, and the other is the segment distance which is the total distance for the segment.

$$SegmentDistance_i = \sqrt{(x_i - x_{i-1}) * (x_i - x_{i-1}) + (y_i - y_{i-1}) * (y_i - y_{i-1})} \tag{4.13}$$

$$LocalDistance = PointDistance \tag{4.14}$$

$$LocalDistance = LocalDistance + PointDistance \tag{4.15}$$

LocalDistance is initialized as *PointDistance* and is incremented by *PointDistance*. The following condition is checked to determine whether or not we've reached the end of a segment of the polyline (when this happens, we move onto the next segment):

$$LocalDistance >= SegmentDistance \tag{4.16}$$

When *LocalDistance* overflows *SegmentDistance* (i.e., we reach the end of a segment), we reset *LocalDistance* by subtracting *SegmentDistance* from it. By doing so, we obtain the appropriate *LocalDistance* to begin with on the next segment. The previous three formulas are contained within the same loop which computes the actual point list.

$$x = x_{i-1} + (LocalDistance/SegmentDistance) * (x_i - x_{i-1}) \tag{4.17}$$

$$y = y_{i-1} + (LocalDistance/SegmentDistance) * (y_i - y_{i-1}) \tag{4.18}$$

x_i and x_{i-1} represent the x coordinates of two adjacent segment points of the polyline, and the y variables similarly represent the y-coordinate points. By cyclying through the loop one time for each point pair required, the list of point pairs is calculated.

The path slot of the animator contains the actual point pairs, computed by using the appropriate previous formulas, that the object(s) will follow:

$$path = ((x_1, y_1), (x_2, y_2), ..., (x_{NumPathPoints}, y_{NumPathPoints})) \tag{4.19}$$

4.2.3 Path Transformations

In order to save mathematical computations, a prototype version of each path is created when the animation is saved (in design-mode, the original information is also retained). Therefore, when an animator is used, transformations can be performed on the path information in order to recreate the paths created by the designer, but in the context of the new objects and their positions.

To create the prototype version of a path, its path-info information is first trans-

lated to the origin, then rotated so that its ϕ is zero. The path-info slot of an animator at run-time, therefore contains the original path information translated to the origin with a ϕ of zero. The translation and rotation are accomplished using the following formula [FVF+90]:

$$\begin{pmatrix} x_{Prototype_i} \\ y_{Prototype_i} \end{pmatrix} = \begin{pmatrix} -x_0 \\ -y_0 \end{pmatrix} + \begin{pmatrix} cos(\phi) & sin(\phi) \\ -sin(\phi) & cos(\phi) \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \tag{4.20}$$

where ϕ is the angle between the two endpoints of the path:

$$\phi = tan^{-1}(\frac{(y_n - y_1)}{(x_n - x_1)}) \tag{4.21}$$

To rotate the path to the origin, we simply rotate it by $-\phi$.

At runtime, the prototype path (i.e., the path that has been translated to the origin) must be scaled, rotated and translated so that it is attached to the origination and termination points defined by the objects in the obj-to-change and anim-obj-to-change slots of the animation interactor [FVF+90].

$$\begin{pmatrix} x_{Path_i} \\ y_{Path_i} \end{pmatrix} = \begin{pmatrix} x_{PathBegin} \\ y_{PathBegin} \end{pmatrix} + \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} ScaleFactor & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{Prototype_i} \\ y_{Prototype_i} \end{pmatrix} \tag{4.22}$$

where $ScaleFactor$ is computed as follows:

$$ScaleFactor = abs(\frac{PathLength}{PrototypeLength}) \tag{4.23}$$

where $PrototypeLength$ represents the length of the prototype path and $PathLength$ represents the length of the actual path. Since the prototype path has been translated

to the origin and rotated so that is is flush with the x-axis, its length is equal to its length in the x direction which is $x_{Prototype_n}$:

$$PrototypeLength = x_{Prototype_n} \tag{4.24}$$

$$PathLength = \sqrt{(x_{PathEnd} - x_{PathBegin})^2 + (y_{PathEnd} - y_{PathBegin})^2} \tag{4.25}$$

The θ used in the transformation above is calculated as the angle that the path's endpoints makes with the origin.

4.2.4 Other Slots

The attach-points slot tells the animator interactor how to compute which point on the object, or data structure, from which the path originates and terminates. Legal values are combinations of the compass points N, S, E, W, NE, SE, SW, NW on the boundary of the object and the center of the object (for each path, there will be two values). The beginning and ending points of a path are determined by the values in this slot. For example, if an object is to move between two elements of a data structure, node1 and node2, and the attach-points slot contained the values ("S","W"), the originating point of the path would be computed by taking the south point on node1 and the terminating point would be computed by taking the west point of node2. If an object is to be moved between two elements of a data structure, then the obj-attach-point slot is used to determine which point on that object will move along the path. This slot will contain one value for each item in the obj-to-change slot. Legal values for it are the same as for the attach-points slot. This slot is computed from the obj-attach-point slot on any object attached to a path. The user

can set this slot, or use the default, which centers the object on the path. This slot can also be set at runtime to any of the previously mentioned values.

4.2.5 Programmer Setable Slots

The programmer can change any of the slots of the animator at will, although it is not recommended for all slots. The path should never be altered by the programmer since it is computed and maintained through the use of internal formulas. The attach-points, as well as the obj-attach-point slot are meant to be modified by the programmer.

At runtime, the programmer will generally change the obj-to-change and anim-obj-to-change slots. During the design of the animation, the designer maintains these slots. But, at runtime, these slots contain only place-holders and require objects to be placed in them. The timing slots are also easily changed by the programmer during design time using a menu, and during runtime by directly changing the information in the slots.

4.2.6 The Timing Function

The actual animation is accomplished through the timer-handler slot of the animator. This slot contains the function that is called at every time increment. The movement of the objects is done by changing the position of the objects in the obj-to-change slot. The timer-handler function takes each object in this slot and moves it to the points in the path slot. For each object in the obj-to-change slot, there is a list of points in the path slot associated with it. When the objects have been moved

to all of the point pairs in the path slot, the animator is stopped.

Because the path is computed by a formula that depends on both of these slots, if either change, the path is recomputed to reflect the change.

4.3 Using an Animator

To use an animator in an algorithm, the programmer must provide a small amount of information. This information consists of the object to be moved and, if appropriate, and the elements of the data structure that it moves between. The object to be moved is determined when the animator is originally created. For example, for the following example, the animator was designed such that the elements of the list are to be moved. When the programmer provides the objects for the obj-to-change slot, the animator already knows that these are the actual objects to be moved.

The following is an example of an insertion sort algorithm which does not include any animation code [KLT91].

```
program insertion_sort (list_type *head)

node      list_element;
list_type *tail=head, *ListTrav1, *ListTrav2, *ListTrav3;

begin
if (head) then
  while (tail.next) do
    ListTrav = tail.next;
    if (ListTrav1.info < head.info) then
      swap(ListTrav1,head);
      head=ListTrav1;
    else
      ListTrav2 = head;
      ListTrav3 = head.next;
      while (ListTrav1.info > ListTrav3.info) do
        ListTrav2 = ListTrav3;
```

```
        ListTrav3 = ListTrav3.next;
      end while;
      if (ListTrav1 = ListTrav3) then
        tail = ListTrav1;
      else
        swap(ListTrav1,ListTrav3);
      end if;
    end if;
  end while;
end if;
end;
```

The code including the necessary information in order to animate the algorithm is as follows: (The *'d code represents added animation code)

```
program insertion_sort (list_type *head)

node      list_element;
list_type *tail=head, *ListTrav1, *ListTrav2, *ListTrav3;
swap1     swap_animator;

begin
if (head) then
  while (tail.next) do
    ListTrav1 = tail.next;
    if (ListTrav1.info < head.info) then
      swap(ListTrav1,head);
*     swap1.obj-to-change = (ListTrav1, head);
*     animate(swap1);
      head=ListTrav1;
    else
      ListTrav2 = head;
      ListTrav3 = head.next;
      while (ListTrav1.info > ListTrav3.info) do
        ListTrav2 = ListTrav3;
        ListTrav3 = ListTrav3.next;
      end while;
      if (ListTrav1 = ListTrav3) then
        tail = ListTrav1;
      else
        swap(ListTrav1,ListTrav3);
*       swap1.obj-to-change = (ListTrav1, ListTrav3);
*       animate(swap1);
      end if;
    end if;
  end while;
end if;
```

```
end;
```

Notice the addition of only two lines per animation call - one to provide the objects to be animated and one to start the animation.

Chapter 5

Visual Design Environment

For any animation, the user must create the paths and the objects to be moved. The visual design environment provides the user with the means to create four types of data structures and three types of paths. Once created, all of these objects can be edited. The data structures available are graphs, trees, lists and arrays. They are treated as two groups: trees/graphs and lists/arrays. The editing available for each of the groups allows the user to add, delete and alter the look of the elements of the structures by changing the prototype. The paths available are arcs, lines and polylines. Editing for paths allows the user to change the look of the path after it has been drawn.

5.1 Paths

The user is provided with three types of paths to choose from: arcs, lines, and polylines (see Figure 1.1). Arcs and lines are created in the same general manner,

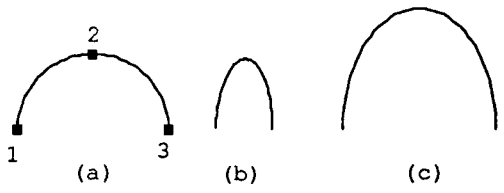


Figure 5.1: (a) An arc path which is selected and the three selection boxes.
(b) The result of stretching selection box #3 toward selection box #1.
(c) The result of stretching selection box #2 away from the base of the arc.

the user clicks where the path is to start and drag the mouse to where the path is to end. To create a polyline, the user simply clicks with the right mouse button where each of the points of the polyline are to be and ends with a left button click on the last point.

All paths are editable. When a path is selected, the points where it can be moved or stretched are highlighted by the selection boxes. For example, an arc has three selection boxes on it, one on each end point and one on the top of the arc (Figure 5.1 (a)). Any of these points can be dragged to move either of the end points or to stretch the arc so it has a greater height. Note that moving either of the end points of an arc path will stretch the arc without affecting its height (Figure 5.1 (b)). To change the height of an arc, the top selection box must be dragged (Figure 5.1 (c)).

It is important to note that a path cannot be created without an object to attach it to. Any path's originating point must be contained within an object. Because a path must originate from within an object, we can provide the user with defined attach points for the object. For any object, there are nine possible attach points: southeast, south, southwest, east, center, west, northeast, north and northeast. Upon path creation, the end-points of the path are "snapped" to the nearest attach point on the object. This provides for clear starting and ending points and insures that the paths start and end in precise locations.

One of the novelties associated with the creation of paths for animations is sequencing. All paths are assigned a sequence number when created. This number represents the order in which the objects attached to them move when the animator is executed. If another path is selected when a path is created, then it will be given the same sequence number which will then indicate that the objects attached to these paths will move at the same time. At any time, a path can be selected and all other paths with the same sequence will also be selected. All non-related paths will be designated by being changed to dashed lines. The effect of several sequences is that several animators, one for each sequence, will be created. When the animators are run, they will run in order according to their sequence number.

5.2 Timing Information

When a path is created, it is assigned a total time and a time increment which it gets from the "Speed" menu of the designer. The animator's timing information is assigned using the first path associated with the animator. To change the timing

information of a path, the user can select the path and change either (or both) of the Speed items. The timing information for all other paths associated with that path having the same animation sequence will also be changed.

5.3 Data Structures

The visual environment provides four data structures to be used in animations: lists, arrays, trees and graphs.

5.3.1 Trees/Graphs

Trees and graphs are created and manipulated using the Garnet aggregraph object. The aggregraph allows for the easy creation of trees and graphs with its many methods and customizable slots. These slots include: node prototype, link prototype, node-prototype-selector-function, link-prototype-selector-function, and various other graphical properties related slots. The links and nodes can therefore be customized by the user. Numerous methods are provided by this object and are used when creating and editing a tree or graph. These methods include: node addition and deletion, link addition and deletion and the layout-graph function which determines the layout of the tree or graph. To create the initial graph, only the layout-graph method is used. Two types of layouts are provided: horizontal and vertical. Each tree or graph is comprised of one or many nodes and the appropriate number of links (may be none).

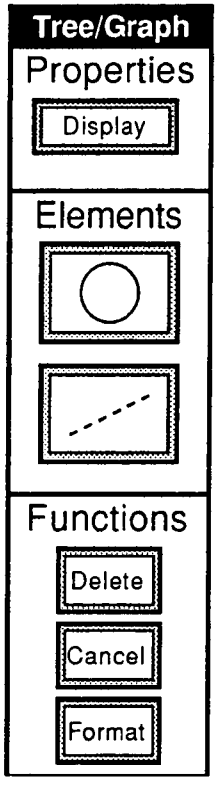


Figure 5.2: The options available for editing trees and graphs.

Interface

The animation editor provides the user with an easy method of building and maintaining trees or graphs. Figure 5.2 shows the main menu for creating and editing trees or graphs. The Elements portion of the menu provides the user with the basic building blocks for creating trees or graphs, nodes and links. The user creates the links and the nodes in the desired layout by drawing circles (nodes) and linking them together with lines (links). Note that a line cannot be created without first creating two circles that it links together. The "Delete" function can be used to delete any unwanted circles or lines once they have been drawn. Figure 5.3 shows an example of what a tree might look like when initially created by the user. Once the desired

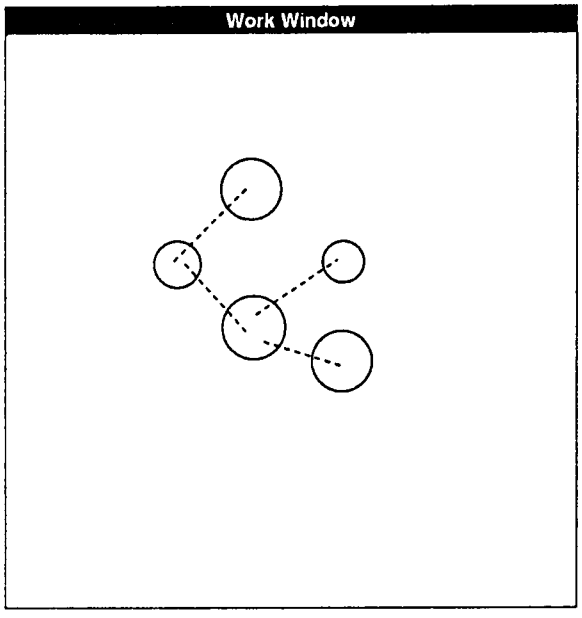


Figure 5.3: Creation of a tree involves creating the nodes and connecting then by creating the links.

layout of circles and lines is produced, the user must choose the "Format" function in order to create the actual tree or graph. This function calls the layout method for the aggregraph and all circles are replaced by the default node-prototype and all lines are replaced by the default link-prototype. Once a tree or graph is created, its graphical properties can be edited with the Graph Properties menu (Figure 5.4). The

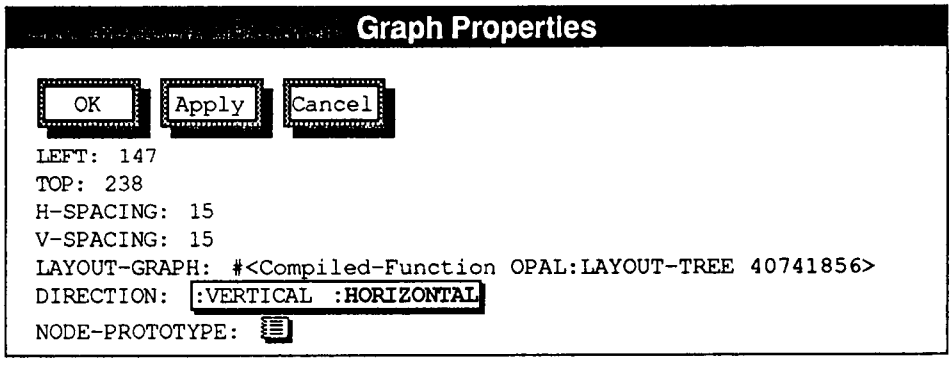


Figure 5.4: Editable properties of trees and graphs.

Graph Properties Menu (Figure 5.4) also allows for changing the actual function that determines the layout of the graph. Since a graph is always initially created as horizontal, the properties menu can be used to change the graph to vertical if desired. Changing the direction of the tree/graph causes the data structure to be re-drawn while maintaining the horizontal and vertical spacing. Also available through this menu is the ability to change the node-prototype. By clicking on the box next to the node-prototype option, the user is prompted for the file containing the new prototype as well as its name. By changing the node-prototype of the tree or graph, all nodes are immediately updated to represent the new prototype.

Implementation of Environment

When building the graph, the user creates circles (nodes) and lines (links) which are grouped in two lists, one for each. The layout of the tree is determined by the nodes and links in those lists and their relationship to each other. The tree or graph is always created as horizontal in order to effectively draw the tree or graph out; therefore, the leftmost node is always marked as the root. Once the layout of the graph is determined through the location of the nodes and links in these two lists, the graph is created by choosing the "Format" function.

The methods available for Garnet aggregraphs are made available through the two menus provided for editing trees and graphs, as well as the main menu. Adding nodes and links is accomplished by linking new nodes and links to an already formatted tree or graph (Figure 5.5). When the "Format" function is applied to such a tree or graph, the appropriate aggregraph method is called. The parameters required by these methods are supplied by the designer. For example, the add-node method

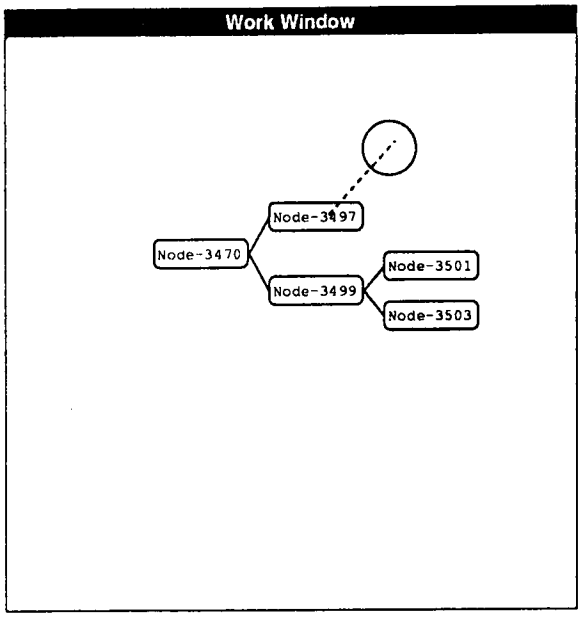


Figure 5.5: Adding a node to an already formatted tree.

requires as parameters: the graph, the new node and the list of parent-nodes. The designer determines the parent-nodes by locating the node of the graph that the new link intersects. The graph parameter is the graph being edited and the new node is a node connected to the graph by the new link. When the "Format" function is chosen (Figure 5.6, the circle and line are replaced by instances of the node-prototype and link-prototype respectively. The delete function from the main menu allows for deletion of nodes and links. It first determines if the object being deleted is an element of a tree or graph then calls the appropriate method if necessary. If a node is deleted, all links to it are also deleted.

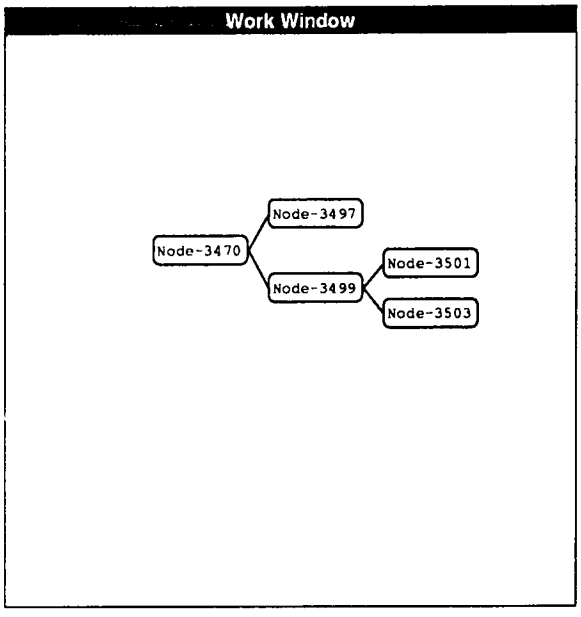


Figure 5.6: After the format option is performed on a tree.

5.3.2 Lists/Arrays

Lists and arrays are created using the Garnet aggrelist structure which allows the arrangement of a set of objects in a graphical list [MGD+92]. Aggrelists provide many customizable slots some of which are rank-margin (the number of items per row or column), item-prototype, and horizontal and vertical spacing. All of these slots, as well as other graphical property slots, are editable through a list/array editor. Many methods are also available for aggrelists and include, among others, add-component and replace-item-prototype.

Interface

To create a list or array, the user designates the width and height of the array by clicking with the right mouse button, dragging and letting go when the appropriate

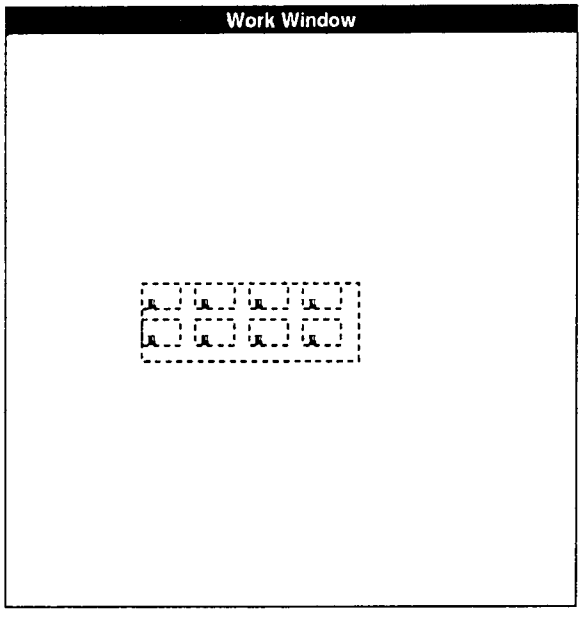


Figure 5.7: Creation of a list involves dragging the mouse from the top right corner to the lower left corner. The bounding box of the list appears while it is being created.

size bounding box has been drawn (Figure 5.7).

The result (Figure 5.8) is a list/array that fits within the bounding box drawn. The height and width of the default item prototype, initially used to create the list or array, are used to calculate the number of items that will fit into the bounding box.

Once the list or array is created, the user is provided with numerous editing options (figure 5.9). These include: insertion before, insertion after, deletion, prototype replacement and list item editing. The editing options provided exploit the methods available for Garnet aggrelist. By changing any of the properties, the appearance of the list can be slightly or dramatically altered.

The items slot of an aggrelist may contain either a number or a list of items. If the items slot contains a number then adding an item merely requires changing the number in the items slot by adding a one to it (Figure 5.10). This number can also be changed to a list of numbers, characters, objects or strings, thereby associating each

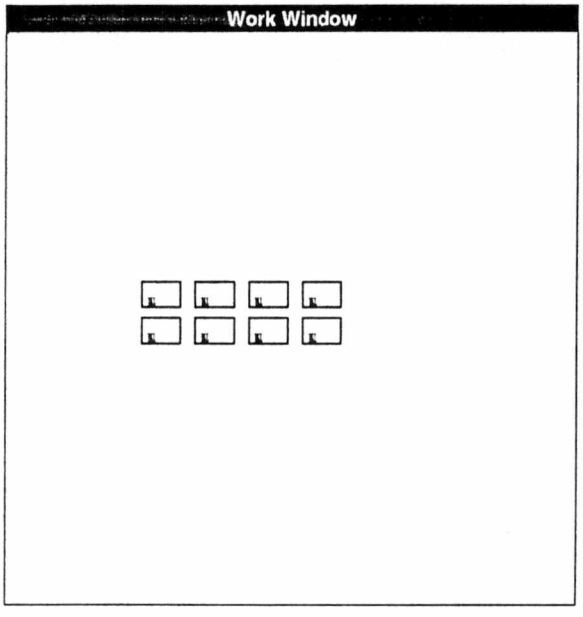


Figure 5.8: After the list is created.

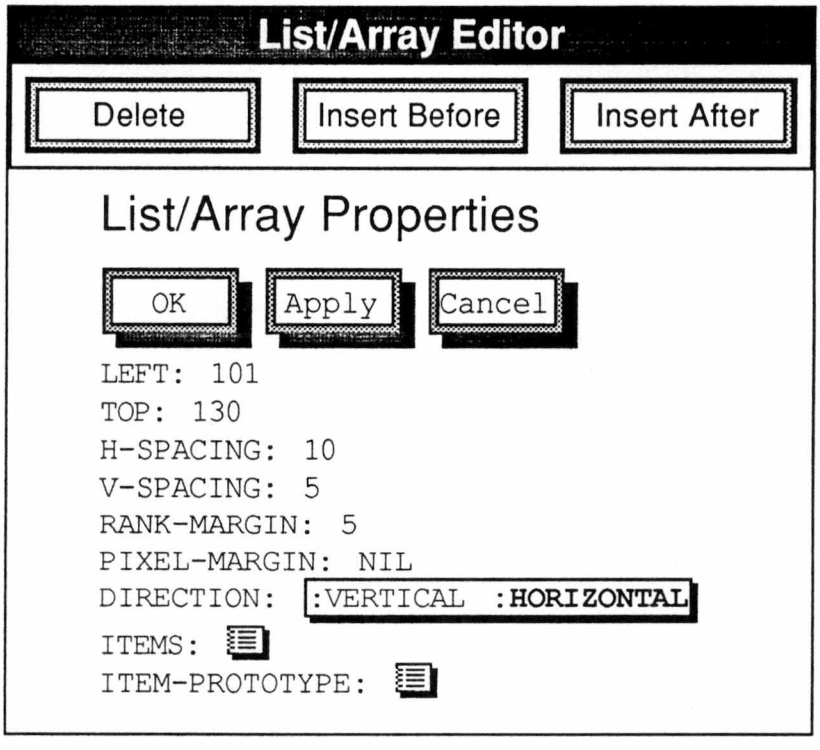


Figure 5.9: The list/Array editor allows new nodes to be added, the item prototype to be changed and all other applicable properties of a list or array to be changed.

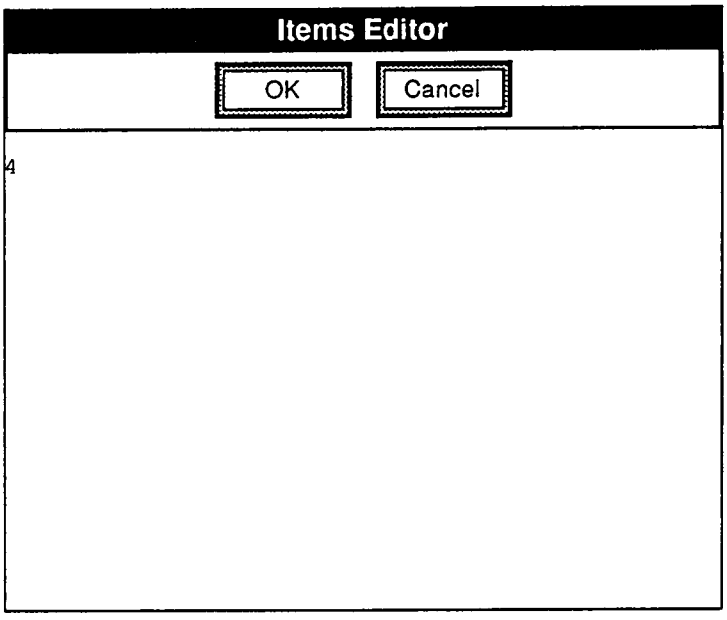


Figure 5.10: Items editor as it appears when the information in the items slot is a number. Change this number to a 5 to add another item to the list being edited.

node in the aggrelist with an element in the list in the items slot. When the items slot contains something other than a number, adding a new item requires using either the insert before or insert after function. When either of these functions is chosen, a dialog box pops up which prompts the user for the name of the new item. Once the new item is named, it is inserted into the list either before or after the selected item. An item-prototype may choose to use the values in the items slot as the labels for the items by constraining the label of the item prototype to the items in the items slot, as the default item-prototype does. When the items slot is a list, each object in the list or array is labeled with one of the values in the list. When a new object is added to an array or a list with labeled items, that object appears with the label the user enters. If the items slot contains a number, the items in the list do not have labels. Note however, that adding an item to a list does not affect the rank-margin. For example, if the rank-margin of a list is six and a new item is added, the new list

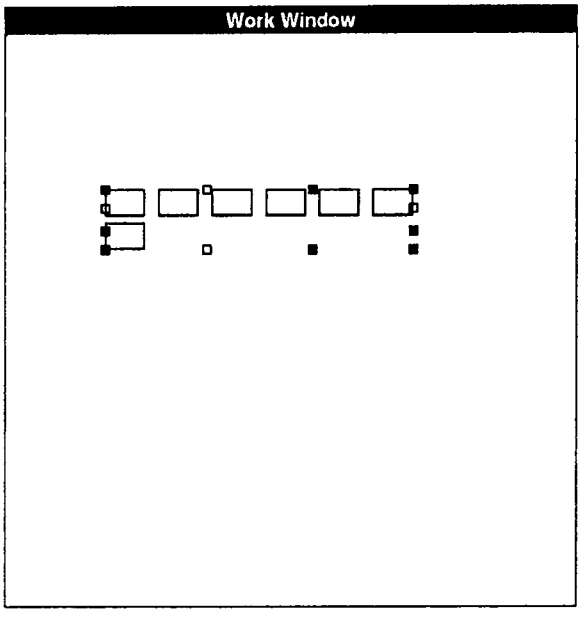


Figure 5.11: Adding an item to a list does not affect the rank margin.

will have six items in the top row and one in the bottom row (Figure 5.11).

It is also possible to add and delete items by using only the items editor. For example, if a list has six items (a b c d e f), by deleting "a" "b" and "c", the resulting list will have only three items. The rank-margin of the list, however, will not be affected. If the rank-margin is six, then even after deleting three items, the rank margin remains six and if three more items are added, they will all appear on the same row or column.

As with trees and graphs, the user has the option to change the item-prototype. This change will not affect the information in the items slot. The only thing affected by this change is the appearance of the list. The user can use any Garnet object as a prototype. If however, the user enters an item-prototype that requires values in the items slot but neglects to provide any items, then an error message will pop up: "Error In Item Prototype" and the list will not be affected.

Implementation Environment

As previously mentioned, numerous editing options are available for lists and arrays. Many of these options make use of the methods available for aggrelists. For example, the replace-item-prototype method is used when the user changes the item-prototype of a list. This method requires the aggrelist and the new item prototype as parameters and changes the item-prototype of the list such that all of the items in the list represent the new prototype. All item addition and deletion are accomplished through methods as well. The add-item method requires only the aggrelist but also allows a position and locator to be specified. For example, if the elements of a list are: ("Rick" "Neal" "Vivian" "Mike"), a new element "Fred" can be added before "Neal".

The interface handles all parameters for these methods, for example, by using the aggrelist that is selected. When it is necessary for the user to provide a parameter, the name of the new item or the name of a new item prototype for instance, he will be prompted for it.

The items of a list are related to each other using the next and prev slots. The previous example of inserting "Fred" would alter the to slot of the item "Rick" to point to "Fred" and the item "Fred"'s to slot would contain the item "Neal". These slots can also be set directly by the user. If the user wants to swap two items of a list for example, the to and from slots can be set to reflect such a change.

5.4 Creating An Animation

Animators are created when the user chooses "Save" or "Run". At this time all of the important information for each animation is gathered and all of the previously mentioned slots are created for each animator. This requires associating paths with the objects that they are to move, getting the timing requirements and attach-points from these paths and determining the relative end-points of the objects that are to be moved.

5.5 Save Options

The user has the option of saving only an animator or the entire animation session. By saving the entire session, the user can continue editing any aspect of the session, including the paths or data structure, by loading the file which contains the session. Saving only the animator provides the user with a portable file which can be used in any other session or algorithm.

5.5.1 Saving an Animator

Saving an animator requires that the animator be made "generic" so that it can be used in any program. This means that any objects that the animator uses cannot be included in any of the slots of the animator. If they were included, this would require the programmer to insure that these objects were present whenever the animator is used. By not including any such objects, the programmer is free to substitute other

objects, not limited by the object originally created. When an animator with such an object is saved, a placeholder is saved in that object's place. For example, if an object "R1" is the object that is moved between two elements of a list, then the object saved in the obj-to-change slot is not "R1", it is nil. This allows the programmers to substitute whatever object they would like in the obj-to-change slot when the animator is used.

When an animator is saved, it is assigned the same name as the file it is saved in, without the .lisp extension. This allows the user to use the animator immediately in some animation without having to edit the file.

The mode slot of the animator is related to the saving and use of an animator in a program. It determines whether an animator is being used in the run-time environment or in the design environment. There is very little difference in how the animators are treated in design mode compared to run-time mode. This slot is checked when the path operations, such as translation and rotation, are performed. If an animator is to be run, the mode is first checked to see what types of operations need to be performed on the path. In design mode, the path can be used as it is. In run-time mode, the path must be scaled, rotated then translated. This will not be an exact duplicate of the path created by the designer. The path created by the designer is used as the prototype. By performing the required operations on it, the path can be used as a prototype to produce the new paths between any two objects.

When an animator is originally created by the designer, it is assigned the design mode. This means that all objects and paths can be used as is. When that animator gets saved to a file, then its mode is saved as run-time. This way, whenever the animator is used, it will be used in run-time mode.

5.5.2 Saving a Session

When an animation session is saved, the information written to the file must be independent of the design environment. When data structures are created, they are assigned names which are generated in that session. By saving the important information about a data structure and not the name of it or any of its elements (which depend on the animation session), the actual data structure need not be saved.

Each group of data structures has different information that is needed in order to be recreated. For both groups, the graphical properties of the structure, such as left, top, node or item prototype, and vertical and horizontal spacing, are saved, as well as an indicator for which type of structure the information represents. Trees and graphs require that the traversal of the tree also be recorded. This traversal is saved as a list and indicates the structure of the tree. Figure 5.12 shows how the nodes are labeled when the traversal of the tree is calculated.

The actual list for the traversal saved would be: (111 110 11 10 1). The traversal as well as other important information about a data structure is saved in the anim-obj slot. The structure for this slot for each structure type is as follows:

Tree/Graph: ("GRAPH" left top node-prot direction vert-spacing hor-spacing traversal)

List/Array: ("LIST" left top item-prot items direction vert-spacing hor-spacing rank-margin)

When a session is saved, and not just the animator, the paths do not need to be altered in any way. The paths are saved exactly as they appear in the designer,

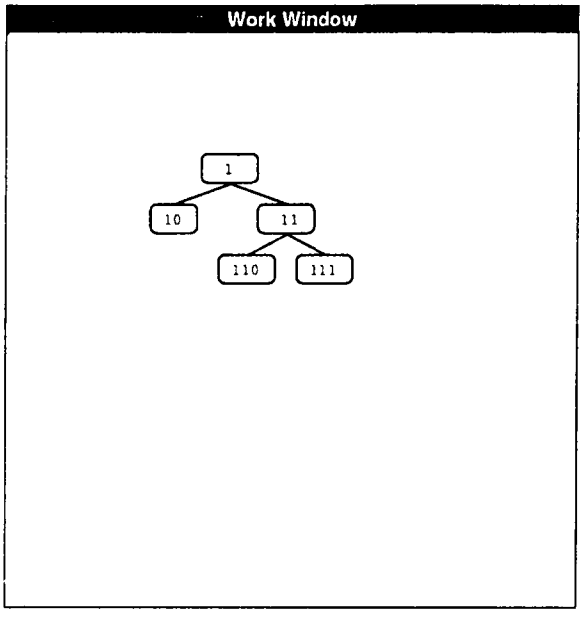


Figure 5.12: Tree labeled with its traversal which is assigned when an animation is saved to be edited.

as well as any other objects related to the animation. This allows for the session to be loaded with all of the paths and objects in their original positions.

5.6 Loading a previous session

By saving the entire animation session and not just the animator, the user can continue editing an animator created in another session. Loading this session causes the designer to recreate all information associated with the original session which may include an animator, a data structure and other associated objects. The anim-obj slot of the animator provides the designer with all of the information needed to recreate the data structures just as they were originally created. The traversal allows for the structure of the loaded tree or graph to be exactly the same as the original. Any

other objects associated with the animation session are also created. For example, if a rectangle "R1" was created in the original design session, then it will also be recreated when this session is loaded and will be called "R1". And for paths, as mentioned previously, they are recreated through the use of the design-info slot.

Chapter 6

Conclusions

With a well-designed graphical environment, animations can be produced by visual manipulations of the positions of objects. We have provided a toolkit that takes advantage of a graphical environment and allows the user to define generic actions which can be used repeatedly. By incorporating these actions and behaviors into one object, the user is not encumbered with the details of the animation, such as updating the position of the objects.

The toolkit consists of a visual programming environment, which allows the user to create an animator by demonstration, and a programming layer which, among other things, encapsulates the behaviors demonstrated into one object, the animator. Our main goal is to provide the user the capability of creating animators without requiring extensive knowledge of the system. We achieve this through the use of the animator interactor. By demonstrating an animation sequence, the user defines the desired behavior, but makes no other assumptions about the objects or data structures to be operated upon. At this point, the user has an animator which can be used immediately or saved to be used in an algorithm animation. The user supplies

the data structures or objects to be moved and little else. This provides an easy to use environment for creating animation sequences.

The animation of an algorithm is then possible by manipulating the animator in a programming environment. This is done by changing key properties of the animator to allow it to operate on different objects each time it is used. Therefore, an algorithm which performs the same action many times can re-use the same animator each time that action is desired.

Animation of an algorithm is therefore accomplished by demonstrating a desired action visually, then using the object created as a result of this demonstration in a programming environment. No application restrictions are currently apparent. The complexity of an algorithm or data structure is not an issue since the primary concern is animating objects, regardless of their composition. The animator is in no way dependent on the algorithm, therefore, as long as an object is to be moved, and the path can be specified, the animator can be created. As a result, the programmer or user can animate any algorithm which involves the moving of objects between two points.

Bibliography

[AHU83] Aho, Hopcroft and Ullman. Data Structures And Algorithms. Addison-Wesley Publishing Company, 1983.

[Bro88] Marc Brown. "Exploring Algorithms Using Balsa-II." *Computer* Volume 21, Number 5, May 1988, pp.14-36.

[Bro91] Marc B. Brown. "Zeus: A System for Algorithm Animation and Multi-View Editing" *1991 IEEE Workshop On Visual Languages* October, 1991, Kobe, Japan.

[Dui86] Robert Adamy Duisberg. Constraint-Based Animation: Temporal Constraints in the Animus System. PhD Dissertation, University of Washington, September 1986, Seattle, WA.

[Dui87] Robert Duisberg. "Visual Programming of Program Visualizations." *1987 Workshop On Visual Languages* August 1987, Lin Koping, Sweden.

[FVF+90] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes. Computer Graphics: Principles and Practice. Addison-Wesley Publishing Company, 1990.

[KLT91] Robert Kruse, Bruce Leung and Clovis Tondo. Data Structures and Program Design in C. Prentice-Hall, Inc. 1991, Englewood Cliffs, NJ 07632.

[MGD+90] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, Philippe Marchal. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment", *IEEE Computer* 23(11):71-85, November, 1990.

[MGD+92] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, James A. Landay, Richard McDaniel and Vivek Gupta. "The Garnet Reference Manuals Revised for Version 2.0. User Interfaces in Lisp." Carnegie Mellon University Computer Science Department, May 1992.

[MZD89] Brad A. Myers, Brad Vander Zanden, Roger B. Dannenberg. "Creating Graphical Interactive Application Objects by Demonstration". ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov. 1989, pp. 95-104.

[Sta89] John Thomas Stasko. TANGO: A Framework and System for Algorithm Animations. PhD Dissertation, Brown University, May 1989.

[Sta91] John Stasko. "Using Direct Manipulation To Build Algorithm Animations By Demonstration." *Human Factors in Computing Systems*, CHI'91 Conference Proceedings, New Orleans, Louisiana, April 1991, pp. 307-314.

VITA

Debra Lynn Jones was born on March 3, 1967 in Owosso, Michigan. She spent most of her childhood there until moving with her family to Fort Lauderdale in 1981.

She graduated from Stranahan High School in 1985 then began her college career at Tulane Univerity. She was on the women's track team at Tulane, where she threw discus and shotput. After changing majors twice, she was awarded a Bachelor of Science Degree in Computer Science in 1990. She still enjoys trips back to New Orleans, and has made it back for Mardi Gras at least once since graduating.

She entered the graduate program at the University of Tennessee, Knoxville in the following fall. She enjoyed three years on the rowing team as a club rower competing in many regional events, and winning several honors. She still enjoys rowing when she can and hopes to do more of it now that her thesis is complete. A year before finishing her thesis, she accepted a job with Oracle Corporation. During that year with Oracle, Debra received the Rookie of the Year award, as well as an award for technical excellence. In spite of a challenging and demanding work environment, she managed to find the time to finish this thesis. She was awarded a Master of Science in Computer Science in August of 1994.