An Interpreted Algorithm Animation System

James Abello and Craig Smith
Computer Science Department
Texas A&M University

Abstract

We present the main elements of a novel algorithm animation system. Algorithms are expressed in a language that resembles textbook set-theoretical descriptions. An animation editor allows a user to express animations via a graphical interface. A language mechanism for binding algorithmic operations to animation actions is provided. By using our own interpreted programming language, more flexible ways to map conceptual-level algorithmic operations to animation actions are possible than with more conventional animation approaches.

1 Introduction

Recently in computer science, visualization has become a popular technique for helping people better understand and analyze data. While the field of scientific visualization has garnered much attention and research, the concept of using visualization techniques to animate algorithms in computer science is relatively new, and methods of designing algorithm visualization systems are still being explored.

Previous algorithm animation systems, such as **AGE** [1], **BALSA II** [3] and **Tango** [8], have contributed much towards the research community and have served well as teaching tools for fostering a deeper understanding of fundamental algorithm concepts. However, these systems suffer from several limitations, the foremost being that they can require a good deal of effort to implement a reasonable animation of an algorithm.

We describe here a model for a visualization environment designed to make the work of the algorithm animator easier. Our goal is to allow a user to describe an algorithm to our environment at a reasonably high level of abstraction, without needing to pay too much attention to details which are not important to the fundamental concepts behind the algorithm. Furthermore, the animation facilities provided by the system are powerful enough to allow the animation of common algorithmic operations in an appealing manner, without being too cumbersome for the algorithm creator to specify to the system.

To foster our effort to create an easy-to-use animation system, we examined several previous algorithm animation systems in order to try to assimilate their most useful features into a new system while trying to avoid their shortcomings.

John Stasko's **Tango** system [8] provides a sound model for the creation of algorithm animations. He categorizes algorithm animations into three components: the algorithm component, the animation component, and the mapping component. By providing an editor for each of these components, he allows one to create animations in a reasonable manner [7]. We feel this paradigm to be very useful in aiding the design of a modular system which is simple yet powerful. Tango is influenced by a similar system called **Balsa II** [3], which was one of the earliest systems created for dynamically illustrating algorithms.

Direction in developing programming constructs and data types supported by our system has been obtained by examining the pseudo-code and data structures used by several algorithms textbooks [2, 4, 5, 6]. Inspection of algorithm clients written for the **AGE** animated graph environment created by Abello, Sudarsky, Veatch and Waller has also proved instrumental in assessing the usefulness and feasibility of our ideas.

Significant guidance in developing animation tools sufficient to tackle the task of animating algorithms has been provided by the research of Sudarsky, who has developed a library of algorithmic animation primitives on top of the AGE platform [10]. This has proven to be quite useful in helping program algorithm animations more quickly and easily [9].

2 Overview

Our view of algorithm animation follows the paradigm adhered to by Abello [1], Brown [3] and Stasko [8]. The essential premise is that algorithm abstraction should be separated from animation. We have achieved this separation by providing two distinct sets of primitives: programming language primitives, and animation primitives. In order to strengthen the degree of separation between the two, our system provides an *interpreted* C-like object-oriented programming language in which algorithm animations are written.

The *programming language primitives* consist of a collection of built-in high-level classes such as sets, bags, weighted directed graphs, iterators, and references, and a collection of built-in algorithms for performing commonly needed routine operations such as searching, sorting, and transitive closure. Providing our own interpreted language allows us to support algorithm descriptions at a higher level than usual, using syntax which is very similar to the pseudo-code used in algorithm textbooks.

Sets and bags are expressed in our language using traditional mathematical set notation. Graphs can be manipulated using this notation as well (and in fact are internally implemented using the underlying set representation). This contributes to the development of algorithm code which resembles textbook set-theoretical algorithm descriptions.

The *animation primitives* in our system consist of a collection of visual objects called *actors*, and a collection of animation actions which are applied to these actors. Actors, such as geometric shapes, bitmapped images, lines, grids, and bars, each have a variety of parameters which control how they are displayed on the screen (color, scale, dimension, etc.) Animation actions, such as alternation, interpolation, follow-path, and iconification, cause the parameters associated with a given visual object to vary over time. Any animation action can be applied to any parameter of an actor, providing a great deal of expressibility and flexibility in a manner which is easy to use.

A powerful mechanism for *binding* algorithmic operations to animation actions is provided to help maintain a clean degree of separation between the two. By using our own interpreted programming language, more flexible ways to map conceptual-level algorithmic operations to animation operations are possible than with the conventional library approach.

Our system allows for the binding of statements in the algorithm code to animation actions, such that these actions are automatically triggered when the corresponding statements are executed. Furthermore, the system allows the binding of *variables* used by the algorithm to animation actions, so that the action is triggered any time the variable is assigned to or otherwise incurs a change of state. This kind of binding gives the algorithm animator a great deal of expressibility and can save a lot of effort, as it is no longer necessary to create separate bindings for every individual statement that modifies a data structure which is to be animated.

The user interface of our system is split into two modes of operation: Composition Mode and Interaction Mode. Under Composition Mode, one can connect the input and output of existing algorithms together to create a pipeline, and select any portion of this pipeline for animation viewing. Under Interaction Mode, a user interacts more intimately with a single algorithm. The user may edit an algorithm, view a demonstration of the algorithm on pre-made input data, or provide his own input from a variety of sources. When viewing an algorithm, the user can rewind or single-step through the algorithm using controls similar to that of a video cassette recorder. Figure 1 shows a prototype of our system in Interaction Mode, while executing Kruskal's Minimum Spanning Tree Algorithm.

3 Division of conceptual and visual operations

The strict division between the conceptual operations and the visual operations is fundamental to the design philosophy of our system. We believe this paradigm helps provide a clean, object oriented structure to the system, thus making the system simpler to use. One major problem with some previous systems is that the animation has to deal with both the conceptual algorithm and its visualization at the same time. For example, while AGE [10] does provide separation between conceptual level graph manipulation primitives and animation primitives, this separation is more philosophical than physical. Calls to routines dealing with the animation are haphazardly mixed with calls to routines important to the algorithm itself, and the client has to store the bookkeeping information required for the animation along with the data structures used for the algorithm. Moreover, the code for the algorithm has to be written in an unnatural event-driven loop manner in order to appease the animation system, which is written using a client-server methodology. The result is a gallimaufry of C code and embedded animation library calls which tends to be cluttered and hard to read, and as such becomes unpleasant for the algorithm implementor to deal with. In summary, the animation source code itself can not be easily used to further one's understanding of the algorithm.

Providing a cleaner separation of the conceptual and visual components of the algorithm, together with primitives powerful enough to allow expression of the conceptual algorithm in a reasonably high-level form, not only simplifies matters for the implementor, but also allows the code itself to be more readable and more useful when viewed side by side with the animation as it progresses.

Another problem with previous systems is that both algorithm primitives and animation primitives have been made available to the user only as relatively low-level C library calls. For example, while John Stasko's **Tango** [8] system does provide for a more physical separation between conceptual and animation components, the user must implement each of these in C code with embedded library calls. There are several pitfalls associated with programming at this level which increase development time of an algorithm animation.

Our system seeks to solve this problem by allowing algorithms to be specified in the built-in *interpreted* object-oriented language, which is similar to typical textbook pseudo-code. Providing an interpreter allows the algorithm developer to produce animations more quickly and easily. Interpreted environments are well-known to decrease development time by providing for easier debugging and eliminating the tedious "edit, compile, run" cycle. Moreover, debugging is made easier because the user can stop the interpreter at any time and examine variables or animation parameters used by the program.

Of course, there is some loss in performance incurred by taking this approach rather than using a compiled language. The primitives used by the interpreter are implemented as a C++ class hierarchy, and are made available in library form for animations which must be implemented directly in C++ for efficiency reasons. (Previous animation systems have made very similar tradeoffs; AGE, for example, uses the client-server paradigm, and thus all animation calls have to be communicated from the client to the server, thereby incurring a loss in animation speed.) For most purposes the speed of our interpreter is comparable to previous systems, while the readability, ease of use, and debugging capabilities afforded by using an interpreter provide significant advantages.

4 Programming language primitives

In order to allow the user to describe algorithms at a high level of abstraction, our system provides a minimal yet powerful set of programming language primitives which the user can employ in the design of an animated algorithm. These primitives are available as constructs built-in to the language provided by the system. These primitives may take several forms.

4.1 Built-in types

One group of primitives is a collection of built-in data types and operators or methods which act on these data types. The implementation of these primitives is internal and not visible to the user. These primitives include sets, graphs, iterators, subcomponent references (i.e. subgraphs and subsets), and disjoint-set operations.

4.1.1 Sets

A set type for maintaining a collection of objects is built-in to our system. Although we use the term "set", our set type is more limited than the mathematical definition of a set. Infinite sets, for example, are not allowed in our system. A set is defined as a concrete collection of objects. Sets may contain other sets, but the semantics of the language disallow any self-reference or recursive definition. This is achieved by implementing all set operations such that they make copies of the set rather than working in place. For example, if the user tried to perform an operation such as A.insert(A) (or A += A), which would insert the set A as an element of itself, the system would make a copy of the set A, and insert this copy into the original set A. For example, if set A contains the integers $\{1,2,3\}$ initially, it would contain $\{1,2,3,\{1,2,3\}\}$ after the operation.

The system supports operators to insert and delete from sets, test set membership, and perform the union and intersection of sets.

There are many algorithms whose pseudo-code is expressed in part using set notation. The goal of providing a built-in set type is to allow the user to express such algorithms to our system with as little translation from set notation as possible. For convenience, another container object called a "bag" [5] is provided, which operates as a set but removes the restriction against duplicate objects.

Our current implementation uses Red-Black trees for the underlying representation of sets and bags. This provides O(log n) insert, delete, and membership test operations, and is, of course, transparent to the user.

4.1.2 Graphs

Since graph algorithms comprise a large portion of the types of algorithms we wish to be able to visualize, it seemed imperative that a powerful, extensible graph type be provided for representing undirected and directed weighted graphs so that each user does not have to start from scratch. Operations to add and remove edges and vertices from the graph and conveniently traverse the graph are available. Tags (which may be strings or any other simple data type) may be associated with the individual nodes and vertices in the graph, or with a whole subgraph), and more complicated graph manipulations such as joining two subgraphs based on common tags are also provided.

Furthermore, the algorithm implementor can add new methods or operations to the graph type using inheritance.

To ease specification, the built-in graph type can be treated semantically as a set of edges and vertices; all the set operators act on the graph in the expected manner.

4.1.3 Iterators

Iterators [5] provide a general mechanism for the traversal of arbitrary data structures. An iterator is a special method in an object which describes how to step through the components of the object in a natural order. Built-in iterators are provided for graphs and sets, and the user can provide their own for user-defined classes.

An iterator is considered a subtype of the class to which it belongs (a vertex of a graph, or an element of a set of integers, for example). It is bound to a particular iteration method on declaration, and traversal methods (in addition to its inherent methods) become available after such a declaration. Iterator methods are provided to initialize the iterator to the first element in the order, and to advance the iterator to the next element. These methods can be called implicitly by using the iterator in a for loop (e.g. "for v in G", where G is a graph and v is a vertex declared as a depth-first-search iterator.) Iterators provide power by hiding the details implicit in traversing complex data structures, and allowing these traversals to be expressed as simple while- or for-loops.

4.1.4 Subcomponent reference

Our system provides the user a method for creating "references" to subcomponents of compound objects such as graphs, sets, and bags, such that the rest of the object is hidden from the user. Changes to elements of a reference are reflected in the original object of which the reference is a part. As an example, let G be a graph (a set of vertices and edges), and let F be a set of references to some of the elements of G. Then F can be viewed as a subgraph of G. Any changes to parameters of any vertex or edge in F affects the same component in G.

Sets of references can be quite useful in many graph algorithms. For example, the fringe list used by the Dijkstra-Prim Minimum Spanning Tree algorithm [2] could be represented as a set of references to vertices in the original graph. Shortest path algorithms, Max-flow, and other algorithms which isolate a portion of a graph or partition a graph benefit from this capability.

References are analogous to pointers in C. However, references are intended to provide the advantages of pointers in a more convenient manner, so that the user does not have to worry about the pitfalls normally associated with pointers. References are automatically maintained by the system, and automatically deleted when the object to which the reference is made is no longer valid.

4.1.5 Disjoint set operations

Our system supports the UNION, FIND, and MAKE-SET operations on disjoint set forests as built-in primitives [4]. Finding which set an object belongs to, taking the union of two sets of objects, and creating a new set containing an object are fundamental operations in many algorithms. Our system contains a built-in *disjoint set forest* datatype, and supports these operations on sets of objects, which can be either built-in datatypes or objects from user-defined classes. Sets of any basic datatype are permissible. This feature is different than the built-in set datatype, which is a general container object.

4.2 Built-in algorithms

Another group of primitives are merely built-in algorithms which perform basic functions commonly needed by higher-level algorithms, but whose specific implementation may not be of concern to the user. These include selection based on search criteria, sorting, transitive closure, and shortest path.

Searching for an element in a collection based on some sort of selection criteria is a routine operation used in many algorithms. We provide a "select" primitive which locates a particular element or group of elements in a set based on a given selection criteria, to keep the user from having to code such mundane searches by hand. Selection criteria can be described in a manner similar to that used by the SQL database language, using an expression composed of a set of selection keywords, such average, sum, and count, whose functions are self-evident.

For example, a select statement in the Dijkstra-Prim MST [2] algorithm could use a select primitive to select edges to add to the fringe list. Many greedy algorithms benefit from this capability.

Other procedures which are commonly used as elementary building blocks in more sophisticated algorithms include *sorting*, *transitive closure*, and *shortest path*. Built-in functions are available to sort lists of elements of any data type for which an ordering is defined. Transitive closure, and single-source and all-pairs shortest path functions operate on a given graph and a weight function.

4.3 Animation primitives

Separate but parallel to the programming language primitives provided by the system is a set of animation primitives. These primitives may be combined in scripts to create visualizations of fundamental operations performed by an algorithm. These scripts are then "bound" to the corresponding operations in the conceptual implementation of the algorithm to create a complete animation.

Primitives for animation in our system are divided into two categories – visual objects, or "actors", and animation actions. Visual objects are static iconic representations of objects represented by the system. Actions are dynamic primitive operations on these objects which are used to define motion or show emphasis or change of state.

4.4 Actors

The following are basic types of actors which are supported.

- Shape Actors — These actors are used to display simple geometric objects such as rectangles, circles, and simple polygons. These can be bound to algorithm data variables to represent vertices in a graph or tree, nodes of a linked list, and the like, or be used to enclose any other visual object to separate it from other objects in the display window.

- Bitmaps — Bitmapped images (read from external files in xbm format) allow the user to depict arbitrary shapes in the system.

- Tags — Numeric and string tags allow descriptions of the objects or states represented to be shown clearly. The user can create an association between a tag and the value of a particular variable or object maintained by the algorithm, so that when the value is changed by the algorithm, the display is updated accordingly.

- Splines — Both straight lines and splines, with arrows optionally displayed at either end, are provided. The coordinate position of either endpoint can be specifiable either as a coordinate position or as an attachment point to another actor, such as a polygon or bitmap, so that the object is redrawn as the objects it is attached to are moved.

- Grids and Bars — To allow the representation of matrices, arrays, and similar entities, the system provides a visual *grid* object [9]. To emphasize the difference in value between objects, a visual *bar* object (as in bar graphs) is provided. Parameters control grid dimensions, width and height of each bar or grid cell, and the rotation of the grid or bar from the horizontal axis. Both the intersections of the lines which form a grid and the cells delineated by these intersections are addressable by indices, so that tags and other objects can be attached at any point in the grid. Grids and bars can be bound to variables in the algorithm such as arrays/lists, sets, and bags, to provide the user an easy way to display their contents.

4.5 Animation actions

Each actor has an associated set of parameters controlling its placement and appearance. These parameters include dimensions, rotation, scale, position, color. All actors also have a "display" parameter which can be used to turn on or off the animation of the object regardless of the setting of other parameters.

Each of these parameters is "animatable" by varying its value or association over time. This is done by providing a set of animation *actions*. The most simple animation actions is a single change in a parameter of an actor. This change could either be a change in value, or a change in binding. For example, the scaled value of an object could be changed from 1 to 2, doubling its size on the display, or the binding of a tag could be changed from one algorithm variable to another. Any desired visual effect should be producible by executing a sequence of these parameter changes. Since it would be very tedious to express complex animation in this way, a set of more complex primitive animation actions is provided which control how various parameters are to be changed over time.

For maximum generality, the actions can be applied to any parameter of an actor for which they make sense. (The only restriction is based on type – an action which varies a numeric parameter cannot be applied to a string tag, for example).

The animation actions themselves take as arguments the actor and parameter to be varied, and duration or speed at which to display, plus any parameters to control the particular action being performed. The following animation actions are supported:

- Alternate — To create effects such as blinking on and off and flashing colors, we provide an *alternation* primitive which alternates a parameter between two given values.

- Follow Path — We provide the ability to move any actor along a path by specifying parametrically a line or spline for the object to follow, and a duration. More generally, we can apply this same action to other parameters to vary color, scale, and the like gradually over time.

- Interpolate — For some actors, such as shape actors, lines, splines, and bars, it is reasonable to provide a means for automatically interpolating between two sets of parameter values. The interpolate action takes an initial and final set of parameters corresponding to a visual object, and linearly interpolates the values to create a smooth transition from one state to another over a given duration.

- Compound Actions — Any collection of animation actions can be combined into a script, which is a sequence of actions whose time parameters may overlap, allowing many objects on the display to be animated simultaneously.

The ability to apply any of these actions to any set of parameters of a visual object provides a tremendous amount of power and flexibility for algorithm visualization.

To further simplify the creation of animations, a visual animation editor is also provided. This editor allows the user to place actors on the screen, draw spline-based paths for actors to follow, bind actors to each other, and specify time parameters via a graphical user interface (Figure 2). This helps the user avoid having to tediously specify numeric position coordinates and other parameters of actors by allowing them to be specified visually.

4.6 Binding

Given code written using our programming language primitives to execute an algorithm, and scripts of animation actions designed to animate the steps performed by the algorithm, the user creates algorithm animations by providing a mapping which binds the two together so that the visualization occurs "automatically" as the algorithm is executed.

There are two levels of binding between conceptual algorithmic operations and animation actions. Firstly, both individual variables in an algorithm and entire data types can be bound to particular types of actors. Binding a class with a type of actor will cause all objects of that type to be displayed using that actor, unless another overriding binding is given for a particular object. This type of binding controls the appearance of an object, but not its behavior or motion. Secondly, events in the execution of the algorithm can be bound to animation code to be executed when the event occurs.

There are two kinds of event bindings, variable event bindings and statement event bindings.

A variable event binding causes the specified animation code to be executed whenever the value of a specified variable changes. For example, the vertices in a graph could be mapped to code which changes the color of their respective actors based on their state (e.g. fringe vertex vs. unseen vertex in a spanning tree algorithm, or the like). Thus, by using this one mapping, a great deal of animation is provided "for free", because *any* statement in the algorithm which changes the state of the vertex (by assigning to a field the vertex contains) automatically triggers the bound action. This provides a large amount of power and flexibility. Frequently algorithms can be animated using predominately this kind of event binding, thus avoiding the many statement event bindings which might be necessary to achieve the same effect.

As a simple example of what can be achieved with variable event bindings, consider an animation of a sorting algorithm. To animate the operation of the sort, we can merely add a one line variable binding to the array to be sorted. The following code fragment shows a sample declaration of an array along with a variable binding which could be used to animate it:

```
int arr[100] : "bar"
    { $x = #offset; $y = 0; $length = arr[#offset]; }
```

This declares an array **arr** of integers of size 100, as in standard C. The “bar” association causes each element of **arr** to be represented with a bar actor. The compound statement in braces is the variable event action which will be triggered anytime an element of **arr** is modified. Visual parameters of the associated actor are represented by variables beginning with the “$” character. Here **$x** and **$y** are the display coordinates of the bar actor, and **$length** is the length of the bar in pixels. The special variable **#offset** refers to the particular offset in the array during the current triggering which has been modified. The overall effect of this binding, then, is to create a row of side-by-side vertical bars, whose lengths each represent the value of the corresponding array element. Such a binding could be used to animate insertion sort, mergesort, quicksort, heapsort, or any other array sorting algorithm, even one which is not comparison-based. This is simple example of the power and ease of animation provided by variable event bindings.

Statement event bindings associate a particular statement (or subroutine call) with animation code. During execution of the algorithm, the associated action will be automatically triggered when the statement bound to it is executed. This is akin to the traditional in-line animation calls used in other animation systems, but provides a higher degree of separation between algorithm and animation components, in that the algorithm code can be displayed by the system (without the associated bindings) as the algorithm executes, and help to elucidate the workings of the algorithm to the observer.

Statement event bindings are useful when a change in a single variable may not provide enough information to allow animation using variable event bindings. For example, the creator of an AVL-tree animation might provide animation action code to animate the rotate-left and rotate-right operations used to balance the tree. These actions would animate the position of several actors (representing nodes) at the same time to create a "rotating" effect. The actions could then be bound to the corresponding functions in the algorithm itself, so that the animation is triggered when these functions are called by the algorithm.

4.7 System implementation

The prototype for our system is implemented in the C++ language, on a Unix development platform. The graphical user interface is developed using the X Windows System[1], the Motif[2] Widget set. UNIX and X/Motif are both logical choices because of their wide appeal and availability. Since our system is designed to be highly object-oriented, C++ is the natural language of selection. The class structures and inheritance mechanisms provided by C++ aided significantly in the object-oriented design of the system, and in the development of the built-in data types and animation primitives provided by the system.

[1]The X Window System is a registered trademark of the Massachussetts Institute of Technology.
[2]Motif is a registered trademark of the Open Software Foundation.

References

[1] J. Abello, S. Sudarsky, T. Veatch, J. Waller, *"AGE: An Animated Graph Environment,"* to appear in DIMACS Special Volume on Computational Support for Discrete Mathematics, Vol. 15, AMS, Boston, MA, 1994.

[2] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, MA, 1988.

[3] M. H. Brown, *Algorithm Animation*, MIT Press, Cambridge, MA, 1988.

[4] T. H. Cormen, C. E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[5] J. Uhl, H. A. Schmid, *"A Systematic Catalogue of Abstract Data Types,"* Lecture Notes in Computer Science, Vol. 460, Springer-Verlag, May 1990.

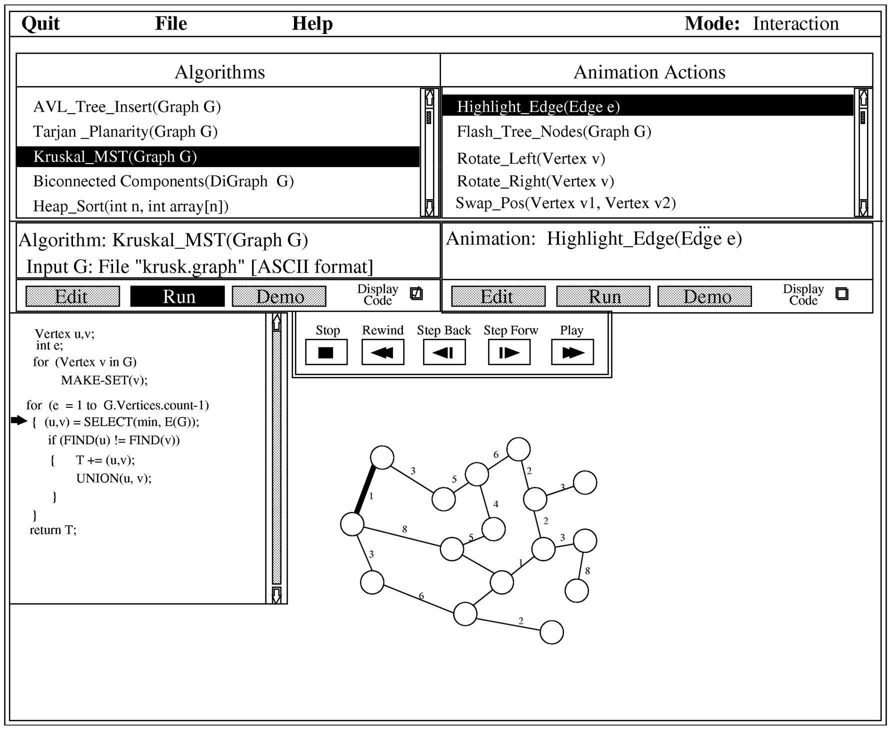
[6] R. L. Kruse, *Data Structures and Program Design*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[7] C. J. Roda, *"A Graphical Interface for the Integration of Algorithm Animations,"* M.S. Thesis, Texas A&M University, December 1992, supervised by J. Abello.

[8] J. T. Stasko, *"Tango: A Framework and System for Algorithm Animation,"* IEEE Computer, pp. 71-85, September 1990.

[9] S. Sudarsky, *"Primitives for Algorithm Animation,"* M.S. Thesis, Texas A&M University, December 1991.

[10] T. R. Veatch, *"AGE: A Distributed Environment for Creating Interactive Animations of Graphs,"* M.S. Thesis, Texas A&M University, December 1990, supervised by J. Abello.



AVL_Tree_Insert(Graph G)
Tarjan _Planarity(Graph G)
Kruskal_MST(Graph G)
Biconnected Components(DiGraph G)
Heap_Sort(int n, int array[n])

Highlight_Edge(Edge e)
Flash_Tree_Nodes(Graph G)
Rotate_Left(Vertex v)
Rotate_Right(Vertex v)
Swap_Pos(Vertex v1, Vertex v2)

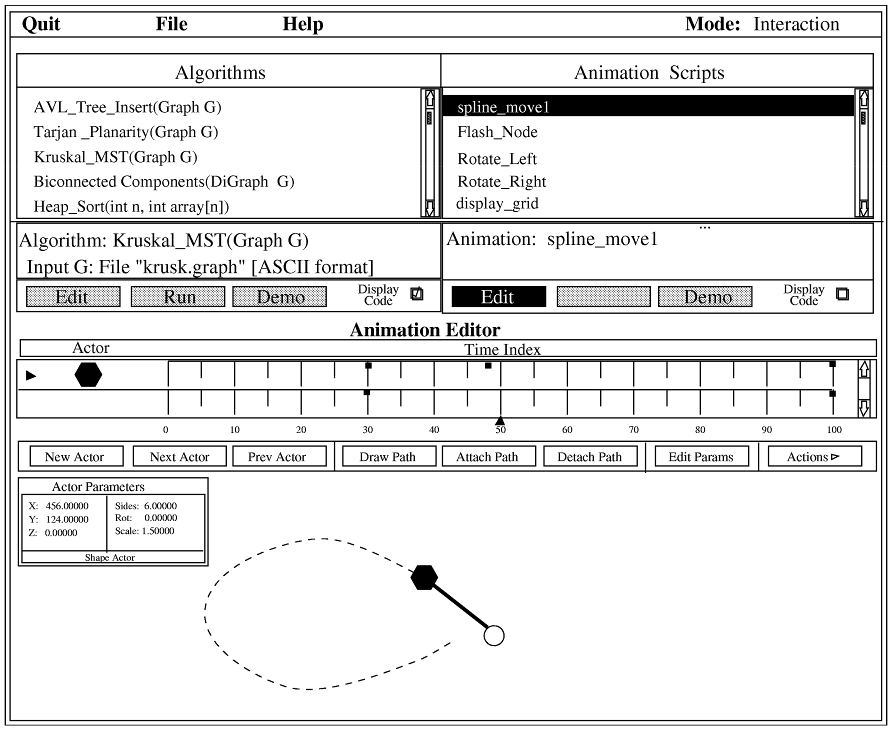
Algorithm: Kruskal_MST(Graph G)
Input G: File "krusk.graph" [ASCII format]

Animation: Highlight_Edge(Edge e)

```
  Vertex u,v;
   int e;
  for  (Vertex v in G)
         MAKE-SET(v);

 for  (e  = 1 to  G.Vertices.count-1)
 {  (u,v) = SELECT(min, E(G));
      if (FIND(u) != FIND(v))
      {      T += (u,v);
             UNION(u, v);
       }
  }
 return T;
```

Figure 1: Animated Execution of an Algorithm



AVL_Tree_Insert(Graph G)
Tarjan _Planarity(Graph G)
Kruskal_MST(Graph G)
Biconnected Components(DiGraph G)
Heap_Sort(int n, int array[n])

spline_move1
Flash_Node
Rotate_Left
Rotate_Right
display_grid
...

Algorithm: Kruskal_MST(Graph G)
Input G: File "krusk.graph" [ASCII format]

Animation: spline_move1

X: 456.00000
Y: 124.00000
Z: 0.00000
Sides: 6.00000
Rot: 0.00000
Scale: 1.50000

Figure 2: Animation Editor