



WASHINGTON • UNIVERSITY • IN • ST • LOUIS

School of Engineering & Applied Science

**SwarmView:
A Graphical Engine for the Interpretation
and Display of Visualizations**

Kenneth C. Cox

WUCS-91-09

January 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

SwarmView: A Graphical Engine for the Interpretation and Display of Visualizations

Kenneth C. Cox

Department of Computer Science
Washington University
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130

1 Introduction

We have implemented a C-based graphic engine, called *SwarmView*, which is used to display animation traces as produced by *SwarmExec*, the Prolog-based Swarm execution engine. *SwarmView* runs on a Silicon Graphics Personal Iris[®] running IRIX[®], a UNIX[®]-based operating system. This paper describes the major design elements of *SwarmView*. A basic familiarity with Swarm and its visualization extension is assumed; the interested reader is referred to the referenced papers. Familiarity with the paper “SwarmView Animation Vocabulary and Interpretation” is required; the appendices of this paper are reproduced from that one.

Section 2 of this paper describes *SwarmView*'s design. Section 3 briefly discusses the user interface with *SwarmView*.

2 SwarmView Design

The *SwarmView* system actually consists of two parts, which together act as the client in a client-server communications model using the Ethernet. The first part is a server daemon which runs continuously. The daemon responds to requests for service by initiating the *SwarmView* program proper. *SwarmView* then completes the connection with the client and begins to read and display animation spaces. In our current configuration, the client is a Macintosh running *SwarmExec*; however, the system can service requests from any source. Both the daemon and *SwarmView* are implemented in C running under Silicon Graphics' IRIX operating system. In addition, *SwarmView* uses the Silicon Graphics graphical library for the rapid definition and display of graphics. The daemon is a standard example of its type and will not be discussed further.

2.1 Design Overview

SwarmView has three major functions. First, it reads and stores collections of animation tuples from the client process. Secondly, it interprets these collections of tuples to produce images consisting of collections of graphical objects and renders these graphical objects in a window on the Silicon Graphics screen. Finally, *SwarmView* interacts with the viewer through both a keyboard and a mouse, providing functionality including pausing and restarting the animation, changing the viewpoint, and capturing the image in a file. This last function is controlled by the user interface, which is discussed in section 3.

SwarmView is a time-based system. At regular intervals, an update occurs; each update corresponds to one “tick” of the animation. During the update, the collection of tuples is examined; if necessary (i.e., the current animation has completed) a new collection is read from the client. The tuples are processed and values for all the parameters of each object are determined. The objects are then rendered, completing the update.

The next three subsections discuss the input, storage, interpretation, and rendering of tuples in greater detail.

2.2 Tuple Input and Storage

Tuples are transmitted from the client as strings. These strings are passed through a lexical analyzer written in LEX and a parser written in YACC. YACC produces an LALR(1) parser with associated actions; in this case, the actions are chosen such that each tuple parsed results in the creation of a dynamic data structure of type `SV_Object` and storage of this object in a list which represents the collection of tuples.

An `SV_Object` is actually a pointer to a C structure, declared as follows:

```
typedef struct s_object {
    SV_Objtype o_type;
    struct s_oparam {
        short p_deftype;
        SV_Value p_value;
        SV_Func p_func;
    } o_params[MAX_PARAMS];
    struct s_object *o_next;
} *SV_Object;
```

The first component of this structure, `o_type`, is a pointer to the master description of the object's type, which contains such information as the number of parameters, the permitted type of each parameter, the function used to render the object, and so on. All graphical objects of a particular type (i.e., all spheres) share the same type information.

The next component is an array `o_params` of three-component structures. This array stores the information associated with each parameter of the object. Returning to the sphere example, since spheres have four parameters (lifetime, center, radius, and color), the first four entries in the array will be used to store the information. When an object of a particular type is created, the default values for each object parameter are entered in the array; as with the type information, the default information is shared among all instances of each object.

Each array structure has three components. The first component, `p_deftype`, indicates if the parameter is the default value. The second component, `p_value`, is the value of the parameter. This component is used both when creating the object to store the value read (or the default value) and during interpretation and rendering to store the current value of a function. The third component, `p_func`, is used when the value of the parameter is a function. The type (i.e., number, list of n numbers, etc.) of each value or function is checked against the required type of the parameter when the tuple is read and any errors are reported.

The last component of the structure, `o_next`, is a pointer to another such structure. It is used for creating linked lists of objects. One such linked list represents the current collection of tuples, while another is used in memory management.

Functions are represented by objects of type `SV_Func`:

```
typedef struct s_function {
    SV_Functype f_type;
    int f_vtype;
    SV_Value f_args[MAX_ARGS];
    struct s_function *f_next;
} *SV_Func;
```

The first component, `f_type`, is a shared function type similar in use to `o_type` in the object class. In this case, the shared information includes the number of arguments to the function and the C function that is called to evaluate the function. The last component, `f_next`, is also similar to `o_next` of the object type.

`f_vtype` is the “type” of the function, that is, the type of value produced by the function; this is determined by examining the function arguments. The C functions which evaluate each function are written to accept any input value types; in particular, it is possible to provide the functions with lists as input, and the resulting output value will also be a list. The third component is an array of values which store the function arguments. At this time, function arguments are limited to numbers and lists of numbers, so a simple representation suffices. If future expansions require additional capabilities (use of functions as function arguments, for example), this field will be enlarged.

Values are represented by objects of type `SV_Value`:

```
typedef struct s_value {
    int v_type;
    union {
        int v_int;
        double v_dbl;
        struct s_value *v_val;
    } v_u;
    struct s_value *v_next;
} *SV_Value;
```

The first component of this structure, `v_type`, determines the value stored. Values are one of four types: integer, double, `t_max` (a symbolic representation of the maximum time in an animation transition), or list of values. The type determines which of the components of the union `v_u` is used to store the data. `v_next` is used in a manner similar to `o_next`; in particular, `v_next` is used to create lists of values.

2.3 Tuple Interpretation

Interpretation occurs once per update, after the global animation time (the current “tick”) has been updated. The linked list of `SV_Object` structures is processed in order. For each object, every parameter is examined. If a function is defined for the parameter (i.e., the `p_func` field is defined), the function is evaluated at the current tick using the appropriate C routine and the resulting value is stored in the parameter's `p_value` field. The result of the processing is a list of objects, each having a value for each parameter.

2.4 Rendering

Rendering refers to the transformation of the object descriptions to graphical form. The process of rendering in `SwarmView` is greatly simplified by the hardware capabilities of the Personal Iris and the software capabilities of the graphical library. The Personal Iris hardware provides the following useful functions:

- Three-dimensional coordinate calculations
- Transformation of 3-D model coordinates to 2-D screen coordinates, with clipping and perspective transformations
- Rapid rendering of vectors (lines) and filled polygons
- Viewpoint transformations, permitting a particular 3-D model to be quickly rotated, scaled, and translated
- Z-buffering, which automatically performs hidden-surface removal.
- Lighting calculations to permit a realistic, shaded 3-D appearance

The software provides clean interfaces to these (and other) hardware capabilities.

Each type of graphical object has an associated C routine which renders the object. Each object in the list is rendered by the appropriate routine, which extracts the calculated parameter values from the object and uses the graphics library functions to add the object to the image. Use of the Z-buffer means that the objects do not have to be sorted (into, for example, back-to-front order) for rendering, and object intersections are properly handled. As an example, spheres are rendered by the following code:

```
void draw_sphere(SV_Object obj,int t)
{
double x1,y1,z1,rad;
int red,green,blue;
float sp_params[4];

    if (!in_lifetime(obj,t)) return;
    val_get_list_dbl(obj->o_params[1].p_value,3,&x1,&y1,&z1);
    val_get_dbl(obj->o_params[2].p_value,&rad);
    val_get_list_int(obj->o_params[3].p_value,3,&red,&green,&blue);
    if (rad <= 0.0) return;
    set_colors(red,green,blue);
    sp_params[0] = x1; sp_params[1] = y1; sp_params[2] = z1;
    sp_params[3] = rad;

    sphdraw(sp_params);
}
```

The routine first checks if the sphere is currently “alive”, that is, whether it should be rendered at tick t . If so, the coordinates, radius, and color of the sphere are extracted from the object parameter array. The color is set, and the routine `sphdraw` (included in the graphics library) is used to produce the sphere. The routine actually produces the sphere using a mesh of polygons and performs the lighting calculations necessary to give the sphere a three-dimensional appearance.

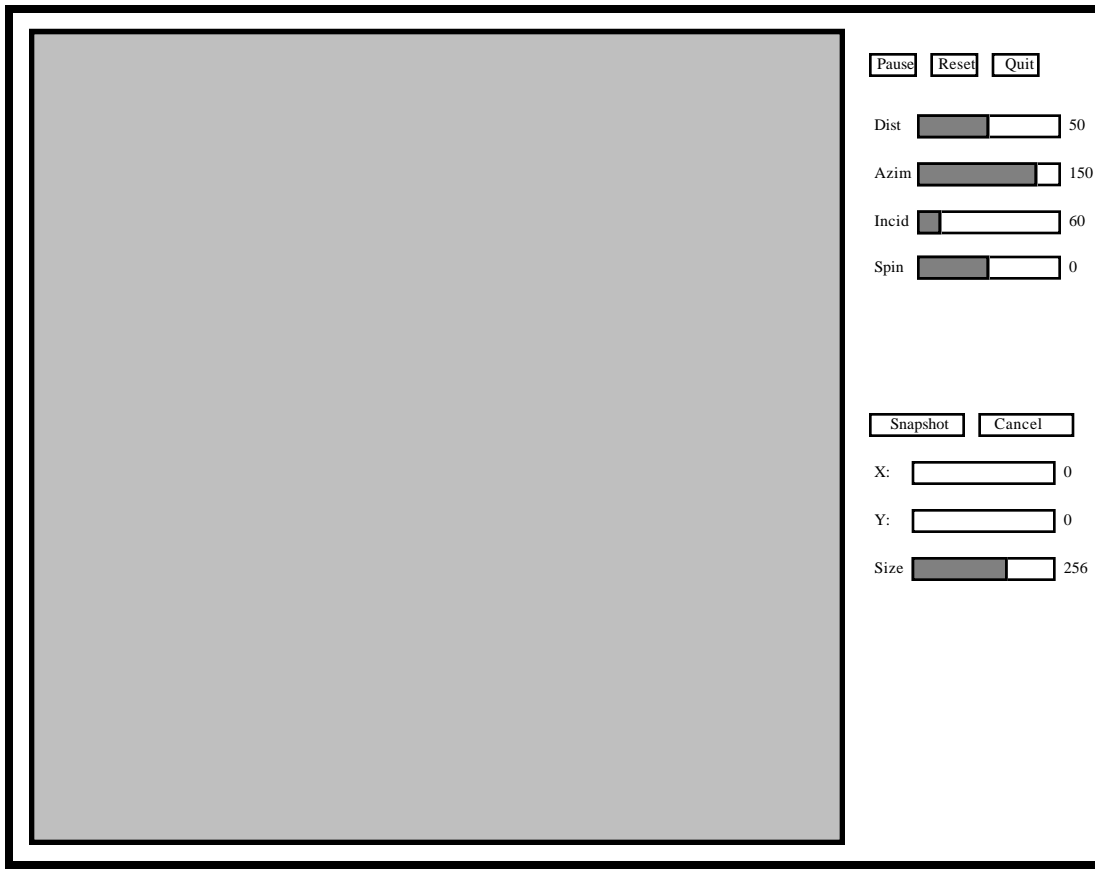


Figure 1. Diagram of SwarmView user interface.

3 User Interface

The user interface with SwarmView is depicted in Figure 1 in diagrammatic form. The large gray area in the left center is the main viewport, where the graphical objects are rendered. To the top right are three buttons, labeled Pause, Reset, and Quit, which are activated by clicking with the Silicon Graphics mouse.. Pause stops the animation while still allowing the user to manipulate the other controls. Reset restores the viewpoint controls to their initial values. Quit exits SwarmView.

Below the three buttons are four sliders; the values of these sliders are changed by dragging the slider with the mouse. These sliders control the user viewpoint in space in a polar coordinate system, as illustrated in Figure 2. The Dist slider changes the distance between the viewpoint and the model origin; Azim and Incid change the azimuthal and incidence angles; and Spin defines an increment to be added to the azimuthal angle at each update, thus causing the model to rotate about the Z-axis. In addition to these controls, the user can change the origin of the coordinate system by dragging in the viewport, thus “pulling” the model in some direction.

The final set of controls is used for capturing single images from the viewport into a file (our facilities also permit videotaping of entire SwarmView sessions). The Snapshot button pauses the animation; the viewpoint controls may still be used. A square is displayed in the image; its position and size are controlled by the X, Y, and Size sliders. Clicking the Snapshot control again saves the image; clicking Cancel ends the snapshot session without saving an image.

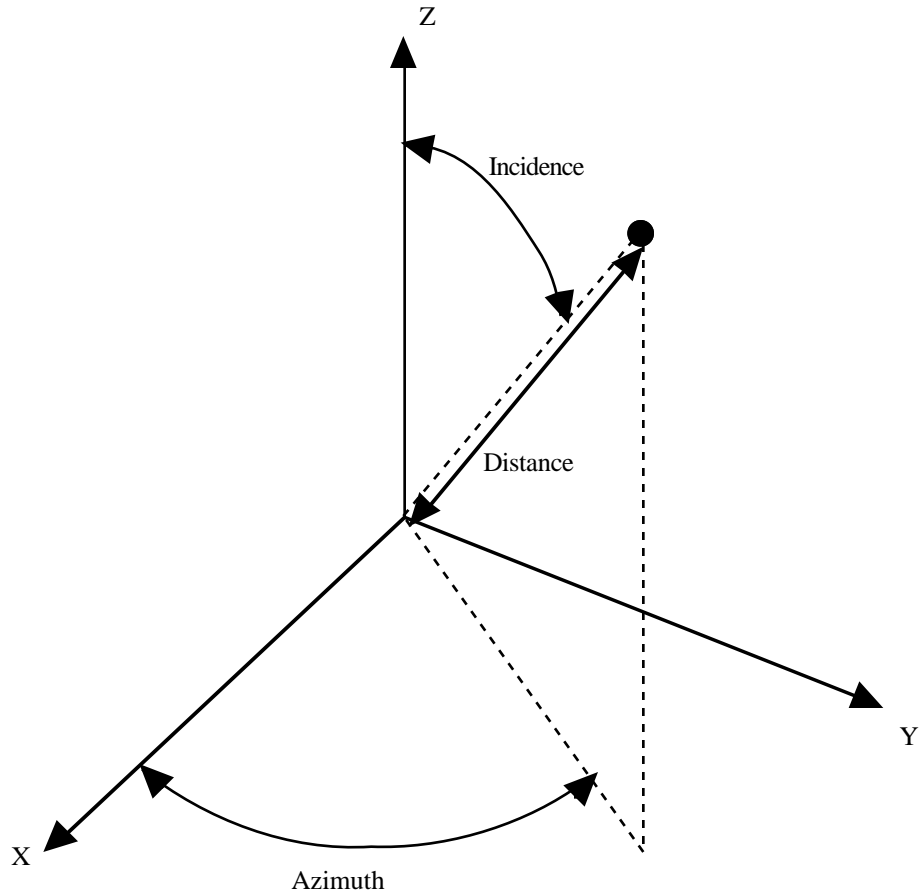


Figure 2. Polar coordinate system used by SwarmView interface.

5 Acknowledgments

The author would like to thank Dr. Jerome R. Cox of the Department of Computer Science at Washington University for his support. This research was supported in part by the National Fellowship Program in Parallel Processing, supported by DARPA/NASA and administered by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Bibliography

Research on visualization of concurrent computations at Washington University:

Cox, K. C., *Visualization of Concurrent Computations* (Doctor of Science Dissertation Proposal), Technical Report WUCS-89-32, Department of Computer Science, Washington University in St. Louis (June, 1989).

Cox, K. C. and Roman, G.-C., "Visualizing Concurrent Computations", Technical Report WUCS-90-31, Department of Computer Science, Washington University in St. Louis, September 1990. Submitted to the *13th International Conference on Software Engineering*.

Cox, K. C., "Visualization in Concurrent Contexts: A Model", Technical Report WUCS-91-7, Department of Computer Science, Washington University in St. Louis, November 1990.

Cox, K. C., Wilcox, C. D., and Plun, J. Y., "SwarmExec: A Prolog-Based Execution Engine for a Shared-Dataspace Language with Visualization Capabilities", Technical Report WUCS-91-8, Department of Computer Science, Washington University in St. Louis, December 1990.

Cox, K. C., "SwarmView: A Graphical Engine for the Interpretation and Display of Visualizations", Technical Report WUCS-91-9, Department of Computer Science, Washington University in St. Louis, January 1991.

Cox, K. C., "SwarmView Animation Vocabulary and Interpretation", Technical Report WUCS-91-10, Department of Computer Science, Washington University in St. Louis, November 1990.

Roman, G.-C. and Cox, K., "A Declarative Approach to Visualizing Concurrent Computations", *IEEE Computer*, Vol 22 No. 10, pp. 25-36 (October 1989).

Roman, G.-C. and Cox, K., "Declarative Visualization in the Shared Dataspace Paradigm", *Proceedings of the 11th International Conference on Software Engineering* (May 1989).

Swarm notation and proof system:

Cunningham, H. C., *The Shared Dataspace Approach to Concurrent Computation: The Swarm Programming Model, Notation, and Logic*, Doctor of Science Dissertation, Department of Computer Science, Washington University in St. Louis, August, 1989.

Cunningham, H. C. and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Distributed and Parallel Computing* Vol.1, No. 3, pp. 365-376 (July 1990).

Roman, G.-C. and Cunningham, H. C., "A Shared Dataspace Model of Concurrency - Language and Programming Implications," *Proceedings of the 9th International Conference on Distributed Computing Systems*, pp. 270-279 (June 1989).

Appendix 1. Grammar

The following is a BNF grammar of the language. Terminals are in **bold**. The symbol λ represents the null string. The “terminal” **identifier** represents any legal identifier (anything described by the LEX-style regular expression `[a-z][a-zA-Z_0-9]*`); **number** represent any legal numeral, either integer or real.

language	::=	transition_list
transition_list	::=	transition transition_list λ
transition	::=	pge ; transition end ;
pge	::=	type_name (attribute_list) type_name ()
type_name	::=	identifier
attribute_list	::=	attribute , attribute_list attribute
attribute	::=	attribute_name = expression
attribute_name	::=	identifier
expression	::=	function constant
function	::=	primitive_function [primitive_function_list]
primitive_function_list	::=	primitive_function , primitive_function_list primitive_function
primitive_function	::=	identifier (constant_list)
constant_list	::=	constant , constant_list constant
constant	::=	number t_max [constant_list]

Appendix 2. Graphical Objects

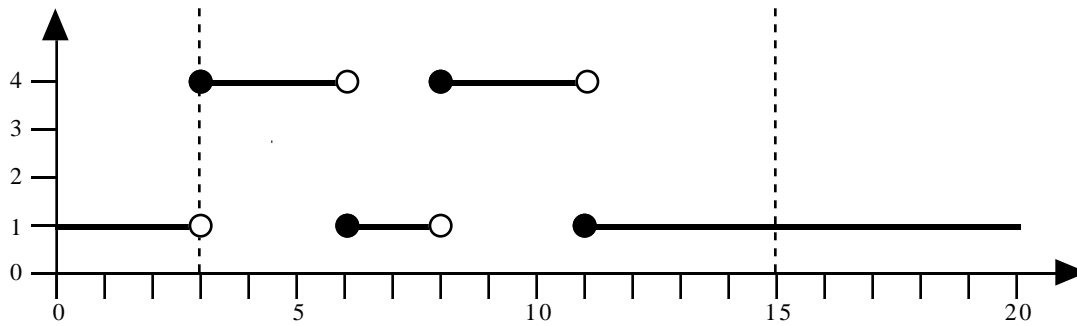
The following graphical objects are provided by the interpreter. All objects have a *lifetime* attribute, which is of type list of two numbers. The type *coordinate* is a shorthand for “list of three numbers” and specifies the X/Y/Z coordinates of the point. The type *color* is “list of three numbers” and specifies the red/green/blue color values, each in the range 0 to 255. The default color is *white*, or [255, 255, 255].

Object Type	Attribute	Type	Default	Notes
point	position	coordinate	[0, 0, 0]	location of point
	color	color	white	color of point
line	from	coordinate	[0, 0, 0]	one endpoint
	to	coordinate	[0, 0, 0]	the other endpoint
	color	color	white	
	width	number	1	width of line (screen pixels)
rectangle	center	coordinate	[0, 0, 0]	centroid
	xsize	number	1	dimensions
	ysize	number	1	
	color	color	white	
	fill	number	0	if non-zero, rectangle is filled
	xrot	number	0	rotation about X-axis, degrees
	yrot	number	0	rotation about Y-axis
zrot	number	0	rotation about Z-axis	
polygon	vertices	list of coordinates	[[0, 0, 0]]	in the order to be connected
	color	color	white	
	fill	number	0	
circle	center	coordinate	[0, 0, 0]	as in rectangle
	radius	number	1	
	color	color	white	
	fill	number	0	
	xrot	number	0	
	yrot	number	0	
zrot	number	0		
sphere	center	coordinate	[0, 0, 0]	
	radius	number	1	
	color	color	white	

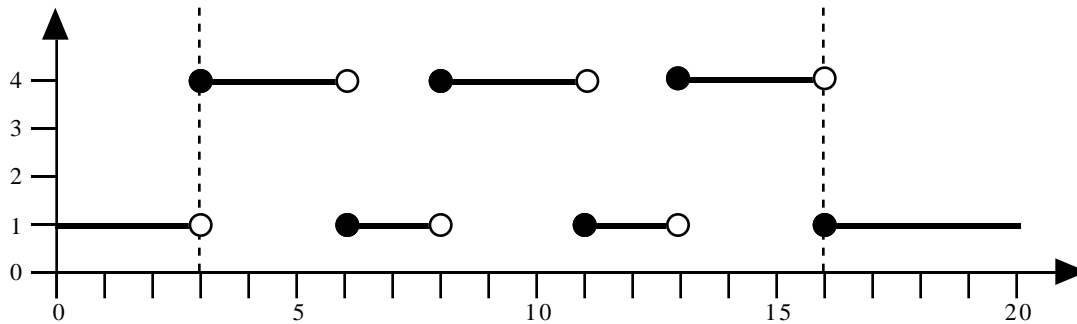
Appendix 3. Functions

Function	Start time	End time	Value before	Value during	Value after
step(t, v ₀ , v ₁)	t	t	v ₀	N/A	v ₁
ramp(t ₀ , v ₀ , t ₁ , v ₁)	t ₀	t ₁	v ₀	linear interpolation from v ₀ at t ₀ to v ₁ at t ₁	v ₁
constant(t ₀ , v, t ₁)	t ₀	t ₁	v	v	v
square(t ₀ , t ₁ , p _{on} , p _{off} , v _{on} , v _{off})	t ₀	t ₁	v _{off}	square wave: v _{on} for p _{on} ticks, v _{off} for p _{off} ticks	v _{off}

The *square* function takes value v_{on} at time t₀, then alternates between v_{on} and v_{off} for the rest of the *during* period. Assume v_{on} periods last a complete time p_{on}; if the interval remaining in the *during* period is insufficient for a complete v_{on}, the value will be held at v_{off} until the expiration of the *during* period. The following diagram gives some examples of this for clarification. Both graphs show a square wave with p_{on} = 3 and p_{off} = 2. In the upper graph the last v_{on} period ends at tick 11; if another period were started, it would begin at time 13 and end at time 16, after the expiration of the *during*. In the lower graph there is sufficient time for an additional v_{on} period to be included.



square(3,15, 3,2, 4,1)



square(3,16, 3,2, 4,1)