EOM: A GRAPHICALLY-SCRIPTED, SIMULATION-BASED ANIMATION SYSTEM

by Paul Pangaro, Seth Steinberg, Jim Davis, and Ben McCann

ABSTRACT

EOM is an implementation of an interactive animation environment based on the following premises:

- Simulation-based animation is the most appropriate kind for computers;
- Scripting animation sequences with two-dimensional, spatially-arranged information sheets is conceptually more advantageous than linear language typed at a console; and
- Interactive and interpretive systems are essential for reasonable feedback for the animator.

This paper describes the appearance of the system to the animator and its basic internal organization; conclusions based on experience with the system point towards its successor system, Loom.

ARCHITECTURE MACHINE GROUP
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUGUST 1977

Note: This paper was written when the authors, who designed and constructed EOM, were members of the research staff or students at the MIT Architecture Machine Group (the predecessor to the MIT Media Lab, under the direction of Nicholas Negroponte). The original text and graphics of the paper is reproduced, modified from its original form only to correct a few typographical errors. The system was fully functional and ran reliably; for example, it was used to generate some of the computer animation sequences for the WGBH "NOVA" program about Linus Pauling (perhaps the first appearance of color raster scan animation in commercial production).

For this first electronic version of the original paper, the text from the original typescript was scanned and converted by OCR. The figures that constitute the "visual programming" scripts are reproduced here from those scans. For the original paper the figures were re-typed directly from the real scripts as they appeared on the graphical interface, without modification; re-typing was necessary because the hardcopy facility available at that time did not provide an acceptable photocopy. — Paul Pangaro, July 2000, pan@pangaro.com

INTRODUCTION

Our work recognizes useful, conceptual perspectives explored by other systems, including powerful modelling spaces [1,2]; systems which move toward the blurring of animation script and animation sequence [3,4]; and the fundamental point of view that animation activity is basically simulation activity[5].

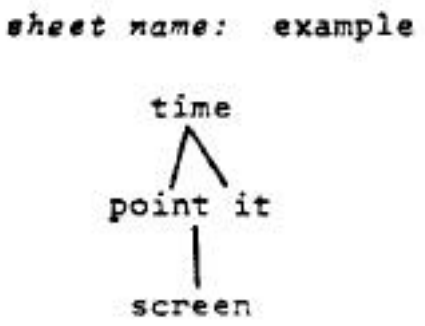
Innovations of the system are: (1) the fundamentally 2-dimensional way in which scripts are specified, thus eliminating the need for countless number specifications and typing of initial conditions, since these are =bedded in the scripts themselves; and (2) the ease of arbitrary subroutining as a conceptual aid to the animator. These two aspects catalyze one another to suggest a mode of computer animation giving the animator an environment in which to construct any conceptual approach to a dynamic image.

EOM (pronounced ee-oh-em) was conceived as a limited experiment, and this implementation incorporates necessary but confining compromises, including vector-based scripting and the distinction within the system of program and data. This paper is a presentation of the manner in which the system is used to produce animation and the internal organization. LOOM, currently under design (and a direct outgrowth of EOM) will contain full power and be completely couched in a touch-sensitive, raster-scan display environment.

EOM has been the outgrowth of rich interaction over a period of a year's time by a number of individuals. They are: Jim Davis, Larry DeMar, Dan Franzblau, Ben McCann, Paul Pangaro, Craig Reynolds, and Seth Steinberg.

SCRIPTING

The script editor is the basic environment for the animator. From menu lists presented on a dynamic vector-graphics display, primitives of the system called BIFS (Built-In Functions) are grabbed via lightpen and pulled unto the basic scripting structure called a sheet. A simple script sheet would be:



The lines joining the BIFs are links which are specified by a minimum number of indications via lightpen and/or button pushes on the keyboard with the free hand. Links are the simplest way of specifying syntactical relations on the sheet. Links are paths by which any variety of data type (such as a number, point, or shape) can be passed around the sheet from one node to another.

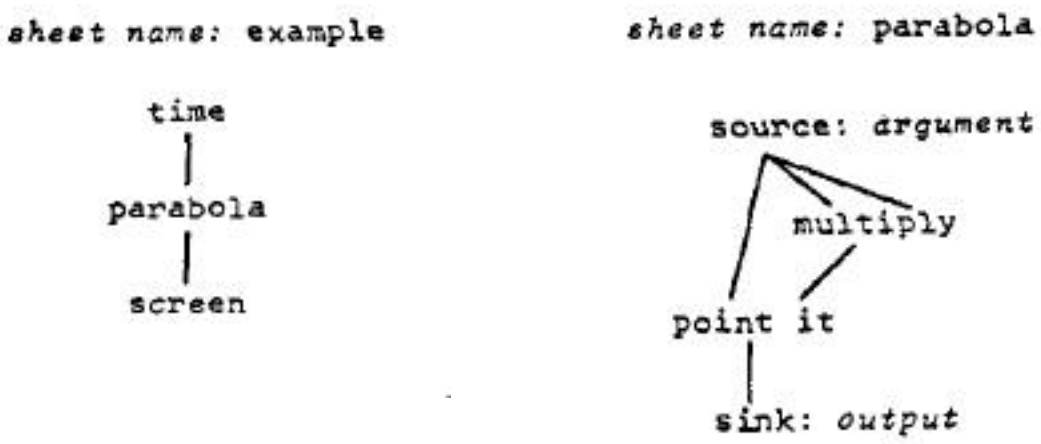
In the above ‘example’, the 'time' node has two links from its output, down which travel the global parameter of the current value of time, which increases each frame. The 'point it' node takes this data into its x input and its y input, and sends a point of that location down its output link. Multiple inputs and outputs are possible for a given node, but are not named on the sheet to prevent clutter. Sub-menus appear during linking for lightpen identification of specific inputs or outputs. The editor allows simple interrogation via lightpen of all the information relevant to links, nodes and sheets.

The output of 'point it' node goes explicitly to a 'screen' node, which causes it to be displayed when the sequence is "evaluated", as execution is called. Allternatively, the data could be sent to a 'console' node which would print out its value for debugging purposes, or into any other node which could take a point as meaningful input. The physical connecting and un-connecting, rearranging, deleting, and integrating of nodes inside the editor is extremely fast and easy to learn, and is one of the first advantages experienced by the animator.

This example produces a point moving linearly in time along the diagonal of the screen. To vary the path of the point, one could draw in an arbitrary shape and use the shape to define a path in time and so on. The list of primitive transformations available in the BIF library is listed in a later section.

SUBROUTINING

in the context of simulations, what if the path desired were a failing parabola? One format might be:



This introduces the subroutining syntax, using a class of nodes called UDNs (User Defined Nodes). 'Parabola' is defined inside the editor, using the nodes 'source' and 'sink' to pass arguments into and out of the sheet. These argument passing nodes contain names, specified by the animator during linking, and these names are used during linking on the calling sheet to match the proper input or output. To invoke an instance on a calling sheet, the menu button 'user-defined' is hit via lightpen, and the name "parabola" is found in a sub-menu list.

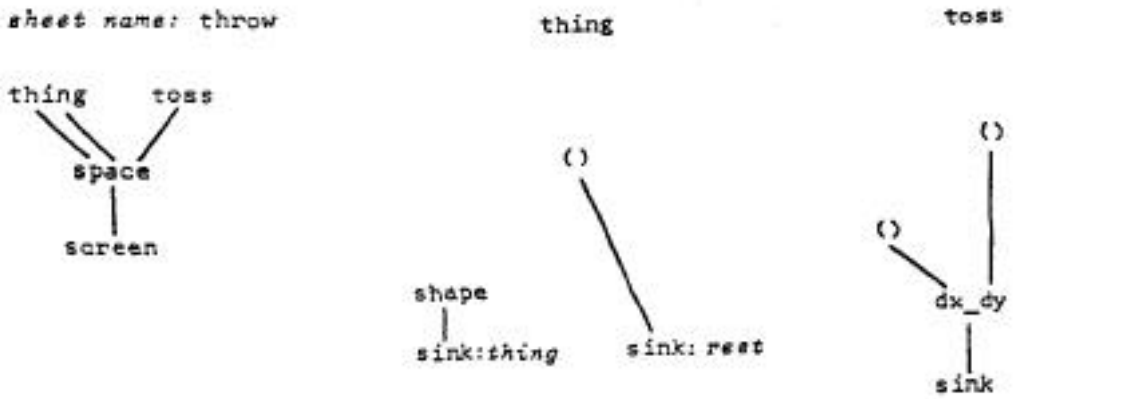
When this name is hit via lightpen, an instance of that node appears, and can be pulled unto the sheet and linked as if it were a BIF. No other action is required, with the system resolving all internal links. Only one sheet is viewed at a time, and to facilitate moving around the various levels of the network for complex sequences, there exist menu buttons called DIVE and POP. Hitting the DIVE button and choosing a UDN causes the editor to display that particular UDN sheet, and make it available for editing. POP returns back to the calling level. Thus, the network subroutines can be traversed, diving and popping to any depth of the tree. This provides an interface consonant with the sheet concept, and in practice, becomes helpful in assimilating the structure of the script.

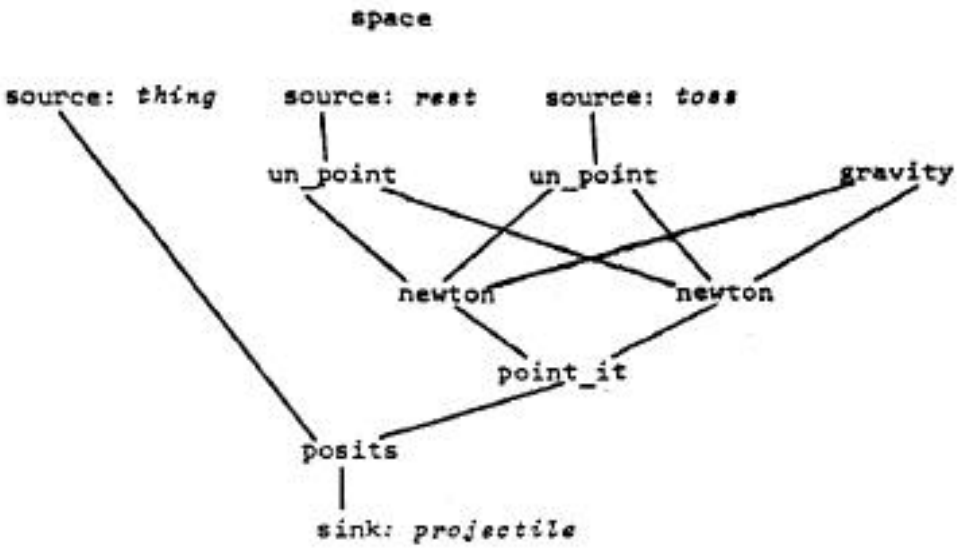
Subroutining is thus implemented in a clean manner, allowing for conceptual clarity while moving through the scripts, and requiring no management. Importantly, these aspects encourage a clarity of scripting since any conceptual element can be placed on any sheet as is appropriate for that sequence. These features are demonstrated in the next section.

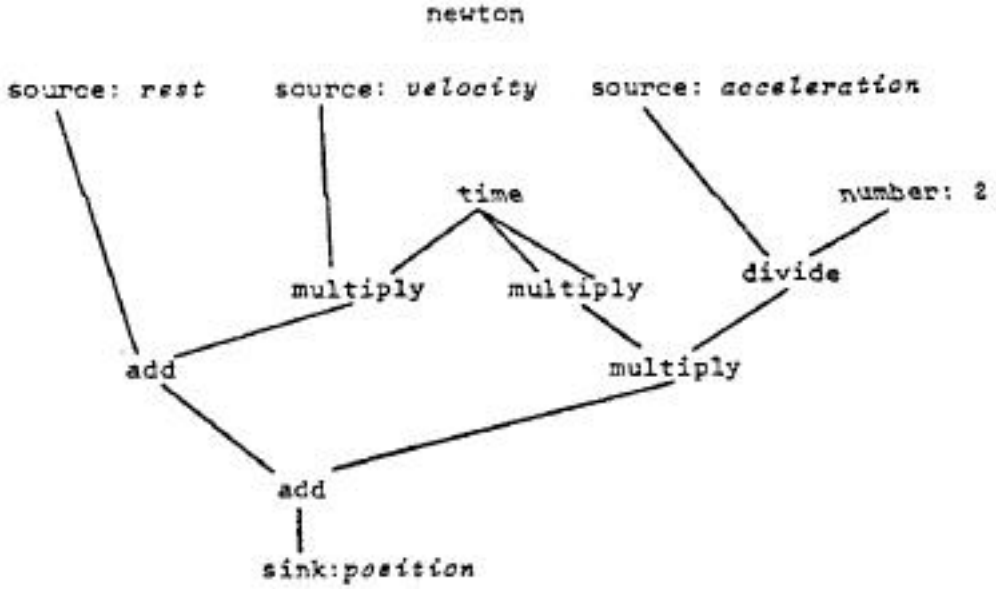
A SIMPLE SIMULATION

Consider the following sequence, which simulates a 'thing' (an object) being 'tossed' into space, and viewed on the screen. On the top level, called

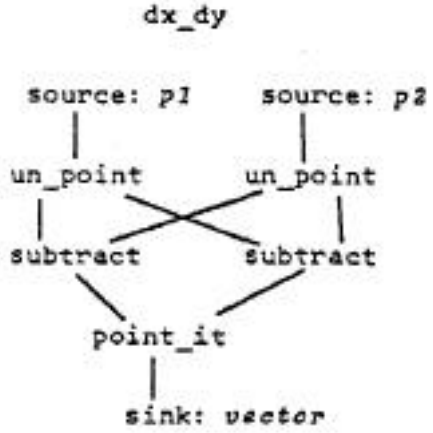
'throw', a 'toss' and a 'thing' are put into 'space', whose output is displayed on the 'screen'.




space
source: thing
source: rest
source: toss
un_point
un_point
gravity
newton
newton
point_it
posits
sink: projectile


newton
source: rest
source: velocity
source: acceleration
time
number: 2
multiply
multiply
divide
add
multiply
add
sink:position


gravity
()
()
dx_dy
sink


dx_dy
source: p1
source: p2
un_point
un_point
subtract
subtract
point_it
sink: vector

'Thing' is simply a defined shape (a shape is a series of points which are to be joined by lines), and an initial position called 'rest'. The rest position is specified via the '()' node, which gives its absolute position on the sheet, as an x, y point. The coordinates of the sheet are identical to the coordinates of the screen, so placement of the rest position on the sheet 'thing' is determined by the position of the node '()' on the sheet itself. No concern for numbers is required, and the specification is completely graphical.

This extraction of graphical data from sheets is also used to specify vectors (magnitude and direction) in the 'toss' and 'gravity' nodes. The position of two '()' nodes on the sheet is passed into 'dx_dy', which computationally subtracts their x components from each other, and the same for y components. 'Dx-dY' returns this information in the format of a point data item.

'Space' takes its inputs, brings in the effect of 'gravity' (which is also defined graphically as a vector on the sheet), and invokes the UDN 'newton', which computes for x and y the position of the object as a function of time using the standard mechanics equation

$$Sx = Sx_0 + vxt + 1/2\ axt^2$$

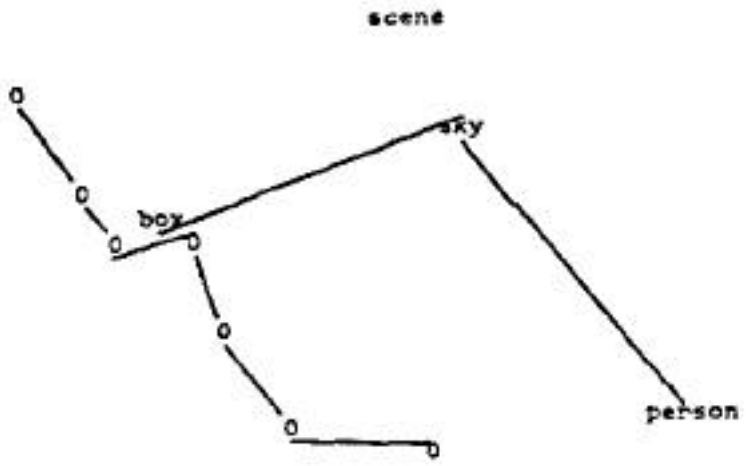
The lambda notation is clumsy for known equations to read in EOM, and would advantageously be replaced by a simple algebraic interpreter.

The turnaround inside the system for changing initial conditions, such as the vector of the toss or even the force of gravity, is just a few seconds. These parameter changes are achieved inside the editor simply by pointing at the node via lightpen and pulling the node to a new position.

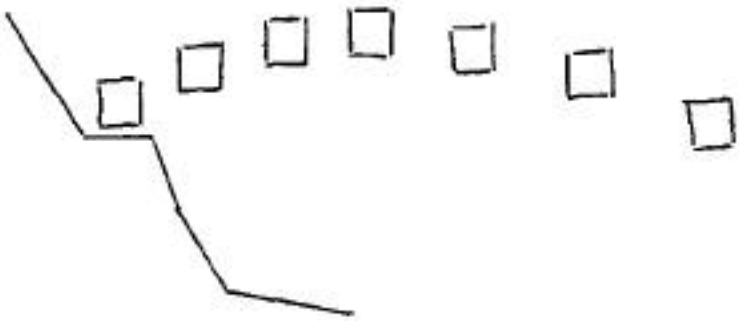
This uniformity between animation action (process) editing and initial values for parameters (configuration) editing is very pleasing. The value returned by the '()' node is, in the current implementation, a constant, but in later versions would be a variable, changing as the simulation moves the object's position. Thus, halting the simulation in the middle of the run would find the position of the node in the editor changed to the current position as specified by the simulation. In a uniform system based wholly in raster scan, the effect would be as follows: In the editor, an initial position for the shape is chosen. The menu command 'simulate' is invoked, and the network of nodes disappears and the actual shape moves from the rest position node up and across the screen. Halting execution at a particular frame results in automatic return to the editor, wherein the network reappears with the position node moved to the shape's position on the trajectory.

In less trivial examples, where the information extracted from the sheet is more than simply position (say, mass, or connectivity of neural elements) the feedback involved in the process (script) and product (animation) relationship is exceptionally powerful.

The previous script network could be loosely termed a 'word-oriented" definition of a sequence. One facility of EOM is its flexible script formatting, an outgrowth of both its graphical, two-dimensional aspect and its subroutine syntax. A network of equivalent visual output, but a more spatial definition, is suggested by the following top level:



The sequence produced by this script is:



The 'O' subroutines draw lines from their inputs to themselves, and output their own position. A string of them reproduces their appearance on the script.

'Box' outputs its box-like shape, positioned where that node rests in the sheet, in this case on the ledge formed by the 'O' nodes.

'Sky' performs the usual kinematics computation over time, as 'space, and 'newton' in the previous example. Here the vector computed to be the initial velocity causes the 'box' to pass through the position of 'sky' on the sheet, making the relation between 'box' and 'sky' determine the initial 'toss' of the previous example.

'Person' takes the output of 'sky' and displays it. A whimsical explanation for the naming of the UDN lies in the ancient question of whether the sound of a tree falling in a forest is present without someone to hear it. The actual point is to demonstrate the possible metaphorical couchings which can be evoked by the script.

it is important to note that, for instance, the ‘O’ node takes unfair advantage of the equivalent visual appearance of a link in the script and a line drawn in an animated sequence. The current system has as the subtle but debilitating limit that, at any instant in the process/product experience of creating an animation by moving from script to animation and back, it is forced to one extreme or the other: viewing only the static script or only the fully dynamic animation.

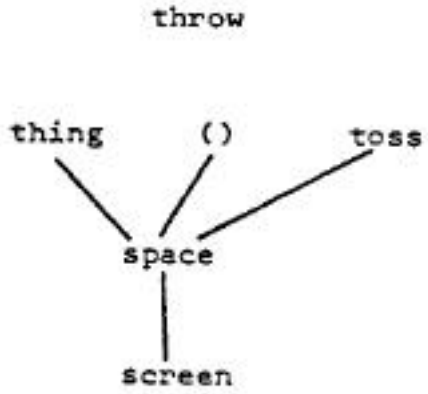
A continuum must exist in which the animator can specify whether each UDN on a current sheet is to be viewed as script (description) or animation (action). In this example, once-the definition of 'box' was satisfactory, it would no longed be necessary to view the disturbing word 'box' on the sheet; the box would simply appear superposed on the sheet in its current visual (active) appearance. Thus as a sequence were built up in complexity, returning to absolute zero at each edit would not be a necessary handicap. Continued reflection on this point leads to understanding its importance, particularly in the quest for further blurring any distinctions between script and animation, programming and animating.

HOOPERS AND SCOOPERS

To facilitate the manipulation of nodes on a variety of sheets to produce a completely scripted animation sequence, certain editor functions are required.

A hooper is a function that allows the specification of a set of nodes to be manipulated together. Manipulations include moving a set of nodes as a cluster across the sheet to make the arrangement clear or meaningful; or to scoop.

A scooper is the function that allows the moving of a node or set of nodes (a hoop) to a different sheet level, that is, to a different conceptual organization. Frequently while constructing a sequence on-line, nodes are initially placed inside a particular UDN inappropriately, as in this case:



The rest position of 'thing' is on the top level sheet, and probably belongs conceptually (as a matter of taste alone, frequently) inside of 'thing'. By indicating the node '()' as a hooped set, and the UDN 'thing' as the place to embed the hoop, the action scoop is invoked by lightpen. This places the node '()' inside of 'space', automatically resolving links, modifying internal argument lists, and adding the necessary 'source' node inside of 'space'.

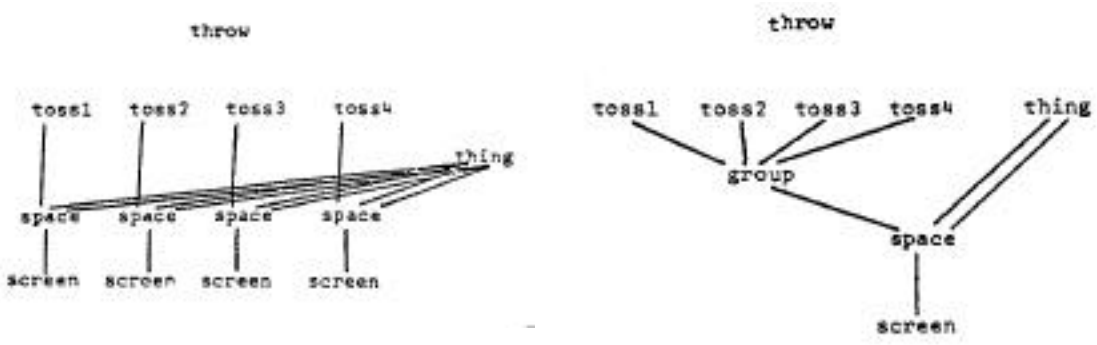
The system will also resolve editing actions such as the modification of subroutine input/output lists even while higher level invocations of that routine are rendered inconsistent by the editing. For example, when an input is deleted from a UDN, sheets which call that UDN will automatically resolve those links which it can, and signal the user of places where further editing is needed.

These editing functions are extremely important because they encourage experimentation with conceptual organization of scripts. It has been found that the sheet syntax, combined with the use of the conceptually simple parceling up of functions and activities of a simulation that produce clear but numerous UDNS of evocative names, leads to clarity of expression in the animation, and is a major attribute and innovation of EOM.

SERIAL/PARALLEL

Quite often, multiple instances of objects are to be displayed in a sequence. To minimize the number of instances of nodes necessary to define a sequence, and to keep the conceptual organization of a script as clear as possible, EOM contains the facility for parceling data items. At any point in the scripting, multiple instances of any type of node can be avoided (if desired).

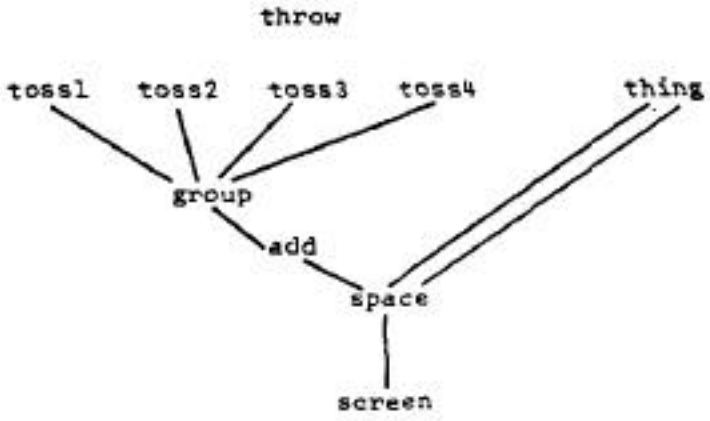
The parallel function is implemented in the form of a 'group' node. This node takes a number of inputs, and internally strings the data into a set called a group, which it passes down its output link. Thus, for example, the following two sheets are equivalent in their painting of the screen:



The group in the adjacent figure passes each vector from each toss, in turn, to the 'space' node which maps its other non-group input arguments per invocation. After evaluation, these are grouped again and passed into 'screen'. Inside the 'space' node, each downward link passes the group of four data items, which at each node are individually mapped as before, and so on down the tree.

If 'thing' were replaced by a group of four different things, the result would be four objects each thrown a different way. Thus, if more than one input to a node is a group, the elements of each group are matched, and mapped unto the node one by one. Any data item of the system may be grouped at any level in the network that comprises the complete script, and all subsequent operations below that point in the network operate on the individual elements of the group. All these effects are controlled by the interpreter, which results in simple coding for the BIF library. This is discussed further under the section on internal organization.

The conceptual inverse of the 'group' node is a serialize function. For ease of implementation in the current EOM, the decision was made to use serial input ports to the nodes, rather than have an explicit 'serialize' node. Thus, in the ‘throw’ example, to pass the sum of the vectors into the space, the sheet would be configured:



The output of the 'group' node is a group, which connects to the serial input of the 'add' node. This is visually implied by the side link to the node, and is achieved in the editor by a menu button. The output from the 'add' is a single element, the sum of the serial inputs. The use of the parceling relieves the need for multiple instances of 'add', and in practice, can reduce the number of nodes greatly, while also contributing to conceptually clarity. There is a later example under the section on entity simulation of this.

MEMORY

The addition of memory to the system is necessary for a wide class of modelling to take place. Because of the multiple sheet leveling of the scripting, a simple syntax had to be devised to distinguish between global parameters and local instances. All variables in the system are matched by name, consist of an initial value set in the editor, and have either inputs, outputs, or both.

The output of a variable evaluates to its present value, straightforwardly.

The input link to a variable contains a variety of information about the particular usage. The absence of an input link indicates that the value of the variable is defined somewhere above (that is, back up the stack of invoked UDNs) and not in the current sheet; the stack is therefore searched back until the name match is found, and the value read. This is an implicit global value search.

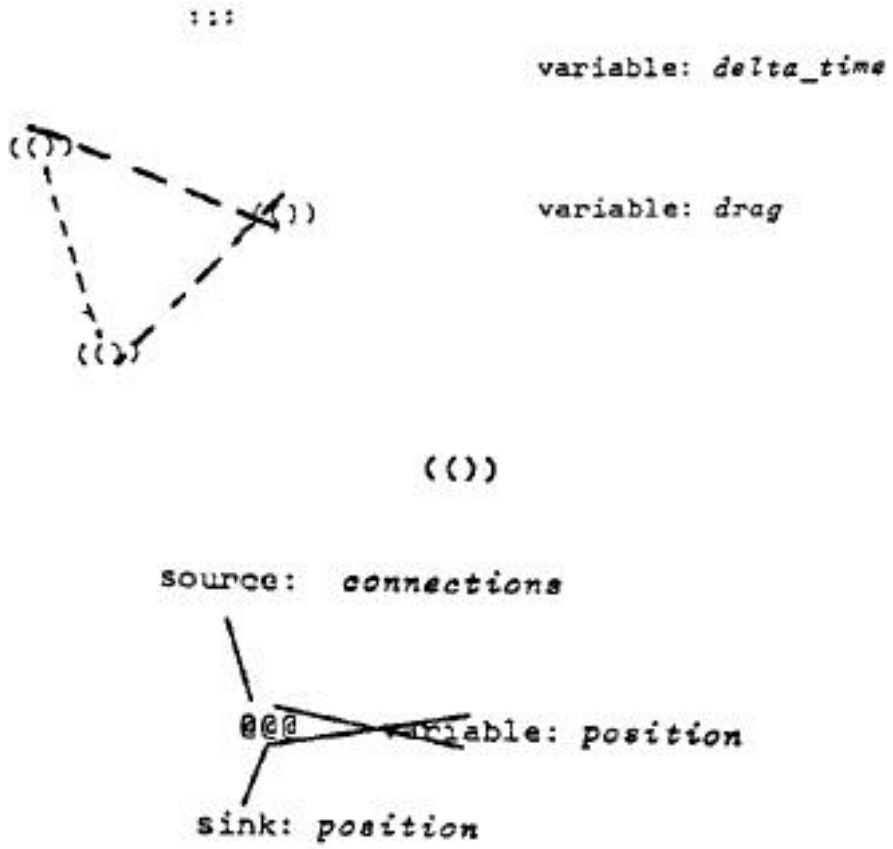
If an input is present, the variable is tagged as a local instance to that sheet, though UDNs invoked from (below) that sheet may access its value by name, as just described. If the value of the input link is null, then the value of the variable is not changed during this frame. If a value is present, then the value is updated at the end of the frame, so that during the next frame this value is available at the output of the variable.

The values of variables internal to a UDN are available on the invoking sheet. Linking is achieved in a manner consistent with sources and sinks, by menu hits on the output list of a UDN.

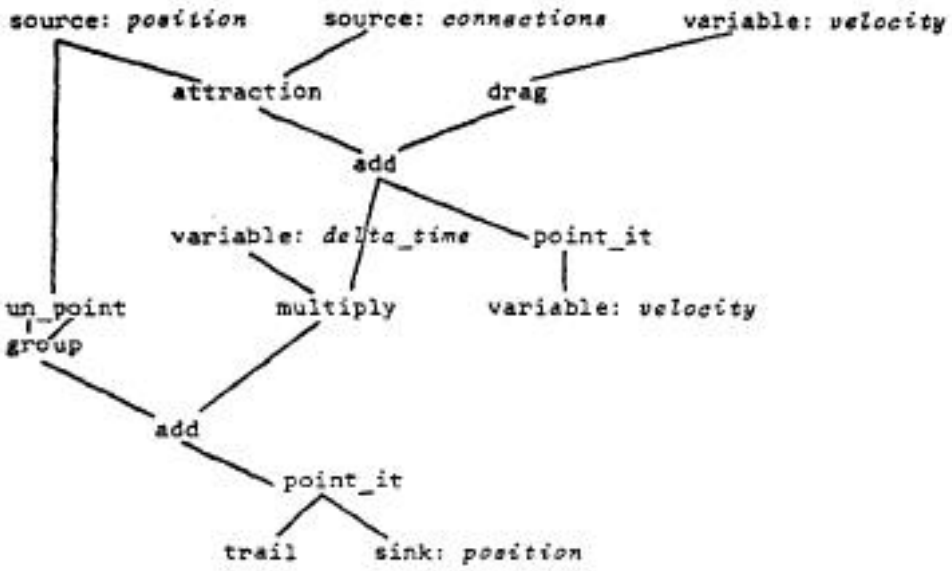
In practice, this scheme is quite sensible and intuitive, and allows a flexibility of memory organization which is clear, Ring-buffers, flip-flops, sequencers, and echo-images are easily implemented. The next section shows examples.

ENTITY SIMULATION

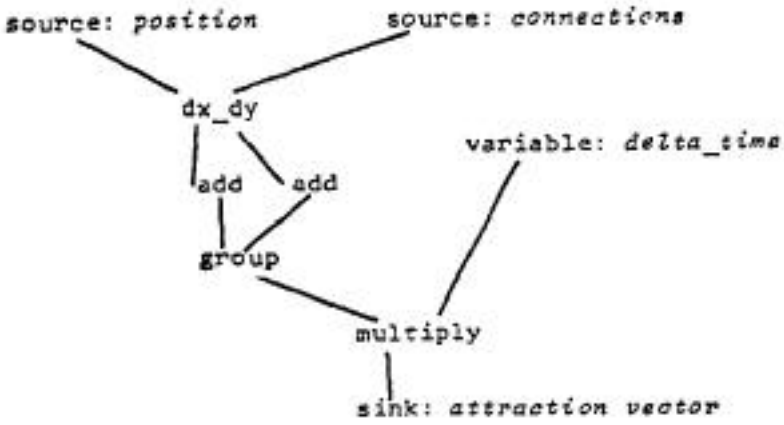
Two-dimensional scripting suggests an easy way to explore a wide variety of simulation classes, including automata evolution. The following favorite class of entities behave as if they were attached to one another by rubber bands.



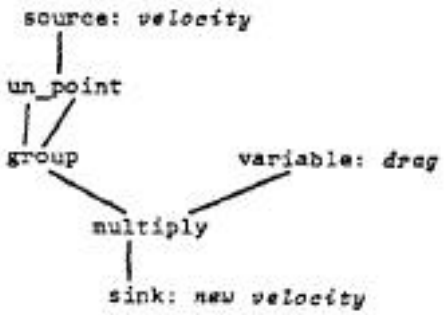
@@@

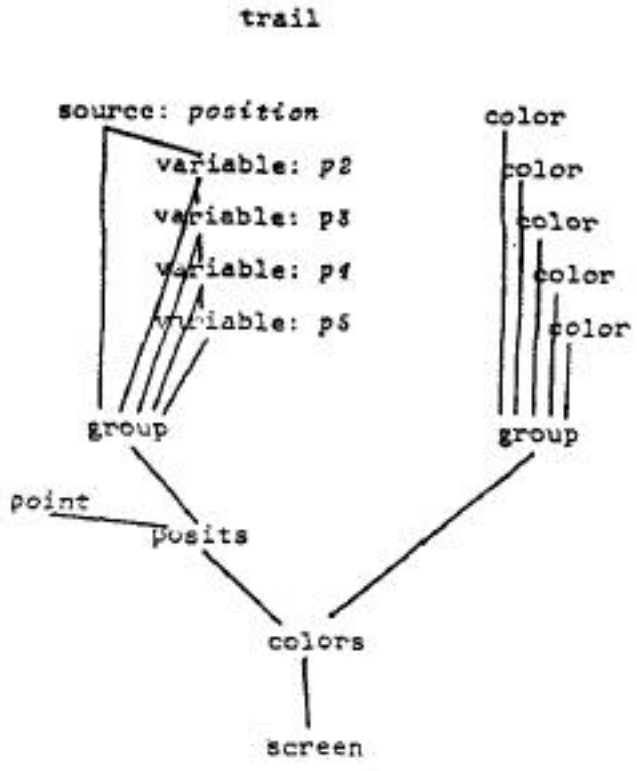


attraction



drag





The entities ‘(())' each output their current position. These links are dotted to indicate that they connect to the output of a variable, and hence can be evaluated at any time during the frame. This distinction between solid links (causal; wait-for) and dotted (get state) is necessary to allow for cycles. Also on top level (called: ‘...’) are stored the global variables ‘delta-time’ and ‘drag’, which are accessed deeper inside the net.

On the ‘(())' sheet is stored the variable 'position’, which for each instance of ‘(())’ on the sheet ‘...’ stores its current position. This position and the connections are passed into ‘@@@’ which does all the motion computing. On '@@@', the current position of this instance of ‘(())’ and the group of positions of the other entities to which they connect pass into 'attraction'. On that sheet, the position of this instance (position source) is compared to all other instances (connections source), and a group of delta distances is passed out of 'dx_dy'.

The output of ‘dx_dy' is a group of x differences and y differences, which are each summed by the serial ‘add’. The sum of x and the sum of y is made parallel by the group node, and each is multiplied in turn by the delta_time variable, which determines the time slice. The resulting group of x,y attraction is passed out the sink node.

'Drag' on the sheet '@@@' takes the current velocity as argument. The velocity, in point format, is made into parallel format by the 'un-point' and 'group' nodes. The x and y velocity is then multiplied by the drag factor, which means that drag is a function of speed. This is put out as a group of x,y.

On the '@@@' level, the results of the 'attraction' and 'drag' effects of velocity are summed in the add node. Since this data is in parallel form, it is passed into the serial input of 'point_it', which outputs a point (interpreted always as an x and y vector of velocity), stored in the velocity variable. Note that there are two instances of 'variable: *velocity*' the sheet. They indicate the same memory cell, and their inputs indicate that the new value is set at this point, as a result of the computation just described. The instance at the top of the sheet, which has only an output connection, provides the current value (this is discussed more explicitly in the section on variables). In the meantime, the position point passed into the source at the top of the sheet '@@@' is transformed from a point into a parallel x,y format by the 'un_point' and 'group' nodes. (This necessity will be avoided in later implementations by more sensible internal formats for the data). The result of this transform combines with the new velocity in the 'add' node, resulting in the new position. Before the velocity is combined with the position, it is multiplied by 'delta_time' to compute 'velocity' times 'delta_time' equals delta position. Still in parallel format, the x,y data is connected to the serial 'point_it' node. This is returned to '(())' where it updates the position variable and is sent up yet another level to '...' where it is available for the next frame to provide the updated positions to each connected entity.

The computed position for this instance is also passed to 'trail' from the '@@@' sheet. This UDN displays the object, which consists of a series of points at its present and previous n positions where n is the number of cells in the shift register implemented by the chain of variables, as shown.

Each time step, the new position shifts down into the register, variable by variable. The outputs at each time are grouped and passed into the positioner node 'posits'. As explained under the section on serial/parallel, this results in a group of five points returned from the 'posits' node. These five are matched, one by one, to the group of five colors when the 'colors' node is invoked. The result to the screen is five points, at the current and previous four positions, each a different color.

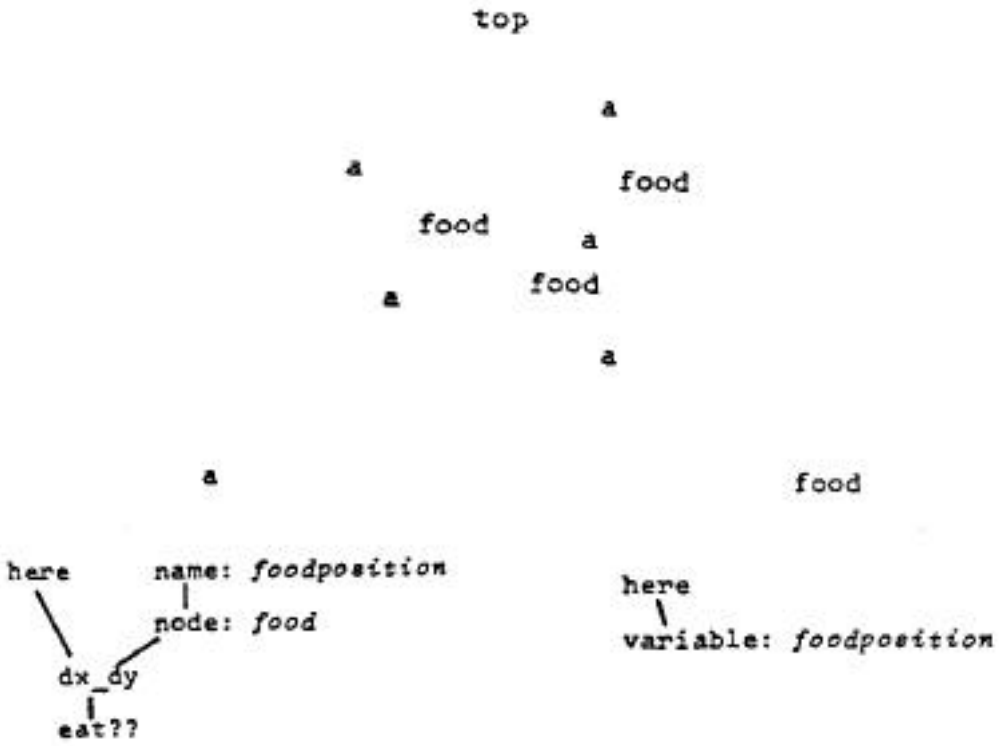
Though there is only a single instance of 'trail' in the entire script network, its multiple invocations from top level by the '(())' nodes cause the proper instancing of the shift register to occur. Similarly for the velocity and position variables. Note in '...' that the two variables drag and delta time simply sit, their initial values being set in the editor and never changed. They are accessed inside of every invocation of '@@@' and 'attraction' and 'drag' with no user special actions.

This example shows in detail the various workings of the system's features. The variable instancing scheme is extremely easy to adopt, and makes implementing complex interacting processes simple. Note that the explanation of any set of existing UDNs is more difficult than the creation of them inside the system. The variable scheme, the UDN sheeting, and the various editor functions provide an environment that is both a pleasure to use and is conceptually transparent and helpful.

ATTRIBUTES

The limitation of variable usage just described is that information can only be accessed below the instance of a particular variable. Frequently, it is useful to pass the information across the network, particularly in entity simulations. This is particularly true when the modeled environment is uniform, that is, all entities interact with all instances of other entities on the sheet. Eliminating the need for links from all entities to all others is conceptually better and makes for practical scripting.

Consider this hypothetical script:



The top level, called ‘top’, contains a few entities, say, cellular automata, which interact as a function of distance on the sheet. Each entity needs position information of all its neighbors to 'see' if closeness relations allow for eating of food or the forming of coalitions. This is achieved as shown, by using the node name' to indicate the name of the variable in the sheet 'food' to be extracted. When encountered by the interpreter, these nodes cause the search of the database for the variable ‘foodposition' in all instances of the nodes on its own sheet level, in this case UDN top. These are, one by one, passed into the node 'distance' on sheet 'a' to determine if that food morsel can be eaten, or whatever.

This is expressly the all condition, since the case of 'a' being related only to specific instance of ‘food’ is included in the case above with ‘(())’, wherein specific connections are made to indicate the instancing. Also, the extraction is made across on the current level in the network as a first restriction of implementation, since it is not clear what the implications of multi-level extraction of attributes in the network are.

INTERNAL ORGANIZATION

The keystone of the EOM system (implemented in PL/1) is its interpretive executor of the network, called the evaluator. The evaluator operates on the database that is constructed by the editor under the command of the animator. Invoking instances of nodes from menus, linking, setting constants, and creating sheets all modify the database which defines the particular script.

The current implementation of EOM maintains a distinction between program and data, and hence the two basic formats are items and nodes.

Items

Items are database elements such as numbers, booleans, colors, points or shapes, which are passed down links from node to node as arguments, and ultimately, in the case of points and shapes, to the 'screen' for display. Numbers consist of float values; points have x and y float values; colors are specified by intensity, hue, and saturation values; shapes are points to be connected by lines by the 'screen' node during evaluation; etc. These items are operated on by the nodes themselves.

Nodes

Nodes are either BIFs (Built-in Functions, taken from the editor menu lists) or UDNs (User-Defined Subroutine Nodes, which can be called from disk or defined on the fly). Nodes either transform data items from one format to another (such as 'point_it' which takes two numbers and outputs a point) or transforms the data itself (such as ‘rotate’, which takes a shape and an angle, and rotates the points of the shape).

The internal database of a BIF consists of a data block for each instance of the node and a description block which is shared by all instances of the node. The node instance contains pointers to the nodes which are inputs, and outputs, flags, and a pointer to the description block. The description block contains information about the number of input and output arguments; argument type information for error checking in the evaluator; and the code entry point in the BIF library.

User-defined nodes have a similar description block, but instead of an entry point to code, the UDN description block has a pointer to a linked list of node instances which make up that UDN.

EVALUATOR

The evaluator walks this list of nodes looking for a node which has all of its arguments (if any) evaluated. Once found, it checks the argument types and then executes a subroutine call to the appropriate BIF code. If the node is an instance of a UDN, the executor calls itself recursively. This process is repeated until every node in the network has been evaluated once. This is done every frame. Once evaluated, any node has a value which can be any item type.

GROUPS

Items can also be linked into sets called groups. Groups are created by inputing a number of instances of items (that is, outputs of nodes) into a 'group' node. The output of the 'group' node is the group whose elements are its inputs. The evaluator breaks up argument sets (n-ary trees) by walking the fringe of the tree and passing the individual 'leaves' to the BIF. If only one of the input arguments to a node is a group, then the other arguments are used repeatedly, once for each leaf, and a new group is constructed from the results of the group of BIF calls. If more than one argument is a group, the fringes of all group arguments are matched, one for one, and the same process is repeated until the smallest group is exhausted.

MULTIPLE INPUT/OUTPUT ARGUMENTS

Nodes (both BIFs and UDNS) may have multiple inputs/outputs. These are handled by building a list of the individual arguments with each referencing node knowing which of n arguments it wants.

SERIAL INPUT PORTS

Several BiFs have one serial argument which can be used instead of the normal input argument set. The argument to the serial input port must be a group. The evaluator breaks this group up in one of three ways:

1. It can accumulate the results of the BIF function as it walks the group leaves. For example, 'add' will sum the elements of the group;
2. The evaluator will parse the group into subsets which are mapped onto the arguments of the BIF, which is then called. This is repeated until the group is exhausted. The final result is a reduced group of values. For example, passing the group (l 2 3 4) to the serial input of the BIF 'point_it' would result in the group of two elements which are points: (1,2 3,4); or
3. The group is passed intact for special casing inside the BIF.

VARIABLES

Variables are bound in each UDN in which a link passes into the named variable. If no input is indicated, the variable is a reference to the nearest binding of that variable name on the execution stack of UDNs. Each Instance (call) of a UDN which binds a variable has a memory cell for that variable In that instance of the UDN. Variables are pre-initialized in the editor and their new values are propagated at the end of each frame. The value of a variable is always defined. Because of this and the delayed propagation, first-in-first-out and circular buffers can be made simply from chains of variables.

Circular networks can be made provided there is at least one variable in the circle.

CONDITIONALS

The system has conditional constructs which either 'switch' one of the two inputs according to a Boolean truth value, or 'gate' one input according to same. If a 'gate' is closed, all succeeding nodes will be ignored with the exception of variables. Variables, instead of changing their value that frame, will retain their old value until the 'gate' opens.

LIST OF PRIMITIVES

The following list consists of those BIFs currently implemented. The set was determined by arbitrary choice, and limited to core size. The coding of BIFs is simple, since the evaluator performs all overheading, such as subroutine calls, group operations, and variable binding.

NAME	INPUTS	OUTPUTS	ACTION
Constants, values are specified in the editor			
number	---	float value	
point	---	x and y float values	
color	---	intensity, hue, saturation values	
shape	---	flood point, color, line definitions	
()	---	null data & position (see ‘here’ node, below)	
here	---	point	absolute position of the instance of the called UDN on the invoking sheet
Controls:			
console	data	---	prints out input data
screen	data	---	draws display data on screen
time	---	global time	range from 0 to infinity, by 1
sink	data	---	passes data to invoking UDN

source	---	data	extracts data from invoking UDN
variable	(new value)	(current value)	memory (see section on variables)
slate	color	---	sets background color
Arithmetic:			
* + / -	a, b ...	result	performs operation on inputs
**	base, power	result	exponentiation
random	---	number	output a random number
Item Makers, these re-format data items:			
point_it	x and y	point (x,y)	
color_it	i, h, s	color (i, h, s)	
shape_it	flood point, color, points	shape (...) with lines drawn between the points	
Number Crunchers:			
sin	angle	sine (angle)	
cos	angle	cosine (angle	
modulo	a,b	modulo (a, b)	
atan	tangent	arctangent (tangent)	
Shape Manipulation			
interp	shape, percent	point	interpolates along the points of the shape and returns a position on the shape at percent distance along the curve
posits	shape, (scale), position	shape	moves a shape to a new origin at position; scale performs an x,y scaling of the position
colors	shape, color	shape	changes the shape color to new value
rotate	shape, angle, center	shape	rotate the shape by angle about the center
scale	shape, factor, center	shape	changes the size of the shape by factor about the center
modify	shape, data, which	shape	replaces the data in the shape by which to the value of the new data: ➢ if which < 0 then change flood point ➢ if which = 0 then change color ➢ if which >0 then replace nth point
Groups:			
group	data, data, ...	group	makes a froup of the input elements and outputs it
series	from, to, incr	group	performs a do-loop on the inputs and outputs all values in the series as a group

comb	group of items, group of booleans	group	reduces group of items according to the group of booleans; ➢ where the boolean is true, the item element which matches is passed; ➢ if no trues, then outputs null
Conditions:			
gate	data, boolean	data	➢ if boolean is true, then outputs data; ➢ else, outputs null
switch	boolean, data_1, data_2	data	➢ if boolean if true, the outputs data_1; ➢ else, outputs data_2
greater lesser equal	arg_a, arg_b	boolean	➢ outputs a boolean based on results of comparison operation
if null	item	boolean	➢ if input is null data, then true ➢ else false
and or not	boolean	boolean	performs logical function
Extractors, these are the reverse operation of Item Makers:			
un_point	point	x, y	
un_color	color	i, h, s	
un_shape	shape, which	data	➢ for which =0, outputs the nth point; ➢ for which > 0, outputs the color; ➢ for which <0 outputs flood point
node_pos	any node	position	➢ outputs the absolute position of the argument node on the sheet

SUMMARY

EOM has proven to be a fertile testbed for a variety of assumptions about the nature of graphical animation on computers. The descriptions in this paper are barely evocative of the ease with which wide classes of animation sequences can be implemented quickly.

Further, the sheet approach has created an environment for exploring conceptual programming in which the relation between mental concept and physical script has some meaning for the animator. The leveling of subroutines, graphical information extraction from 2-D scripts, and dynamic data configuration within the networks are all mutually contributory to a powerful conceptual system.

This work precurses systems that completely blur the distinction between programming and animating. Environments in which programming is performed by creating animated sequences, rather than vice versa, are possible. Loom, the next generation of EOM, will contain a full implementation of all the features implemented and suggested here, fully based in raster-scan hardware. With the programming space (the editor) and the animation space (the moving sequence) merged unto the same display area and viewed under a transparent, touch-sensitive tablet, a major stride toward purely visual programming will be made.

FOOTNOTES

1. An Overview of KRL, Bobrow and Winograd, XEROX PARC CSL76-4.
2. SMALLTALK-72 Instruction Manual, Goldberg and Kay, editors, SSL76-6.
3. PYGMALION: A Creative Programming Environment, David Canfield Smith, Stanford AI Memo AIM-260.
4. Interactive Computer-Mediated Animation, PhD Thesis, MIT MAC, TR-61, 1969.
5. Alan Kay, discussions at Architecture Machine Group, 1975.