EOM: A Graphically-scripted, Simulation-based Animation System

Pangaro, Steinberg, Davis, McCann

Scott: Can you look over before Monday? Need critiques on omissions, clarifications and real-world contexts — Lee

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 on sheets is conceptually more advantageous than linear languages typed at a console;

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 toward its successor system, Loom.

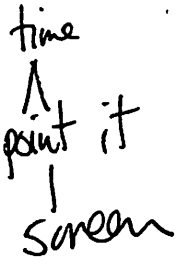
INTRODUCTION

COmputer animation systems have been implemented in a wide variety of forms. Our work is derivative in the sense that it recognizes useful conceptual perspectives explored by other systems. These include powerful modelling spaces (Smalltalk and KRL); systems which move toward the blurring of animation script and animation sequence (Baecker -- Picture driven animation); and the fundamental point of view that animation activity is basically simulation activity (Kaye).

EOM (pronounced ee-oh-em) combines and expands these concepts, adding its own conceptual perspectives as described in this paper. Conceived as a limited experiment, this implementation incorporates necessary but confining compromises, including the vector-based scripting and the distinction within the system of program and data. What follows is a presentation of the manner in which the system is used to produce animation, the internal organization, and future directions. Loom, currently under design and a direct outgrowth of EOM, will contain the full power and be completely couched in a touch-sensitive, raster-scan display environment.

SCRIPTING

The script editor is the basic environment for the animator. From menu lists presented on a dynamic graphics vector-graphics display, primitives of the system called Bifs (for built-in functions) are grabbed via light pen and pulled unto the basic scripting structure called a sheet. A simple script sheet would be:



(caption figure one: A simple sequence)

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

In this 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 by some delta, specified at run time. The 'point it' node takes this data into its x input and its y input, and sends a point of that value down its output link. Multiple inputs and outputs are possible for a given node, but are not named on the sheet

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

Sub menus appear during linking for lightpen identification of specific inputs or outputs.

to prevent clutter. The editor allows simple interrogation via lightpen of all the information relavant to links, nodes and sheets.

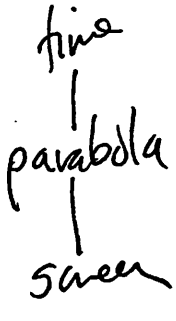
Note that the output of 'point it' goes explicitly to a 'screen' node, which causes it to be displayed when the sequence is ~~xxxxxxxx~~ "evaluated," as execution is called. Alternatively, the data could be sent to a 'console' node which would print out its value for debugging purposes, or into any other node which could take a point as meaningful input. The physical connecting and unconnecting, re-arranging ~~and~~ deleting and interrogating 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, ~~as is clear by now,~~ 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, for this point, or indeed, for any shape. The list of primitive transformations available in the BIF library is listed in Appendix one.

SUBROUTINING

In the context of simulations, what if the path desired were a falling parabola. One format might be:

parabola

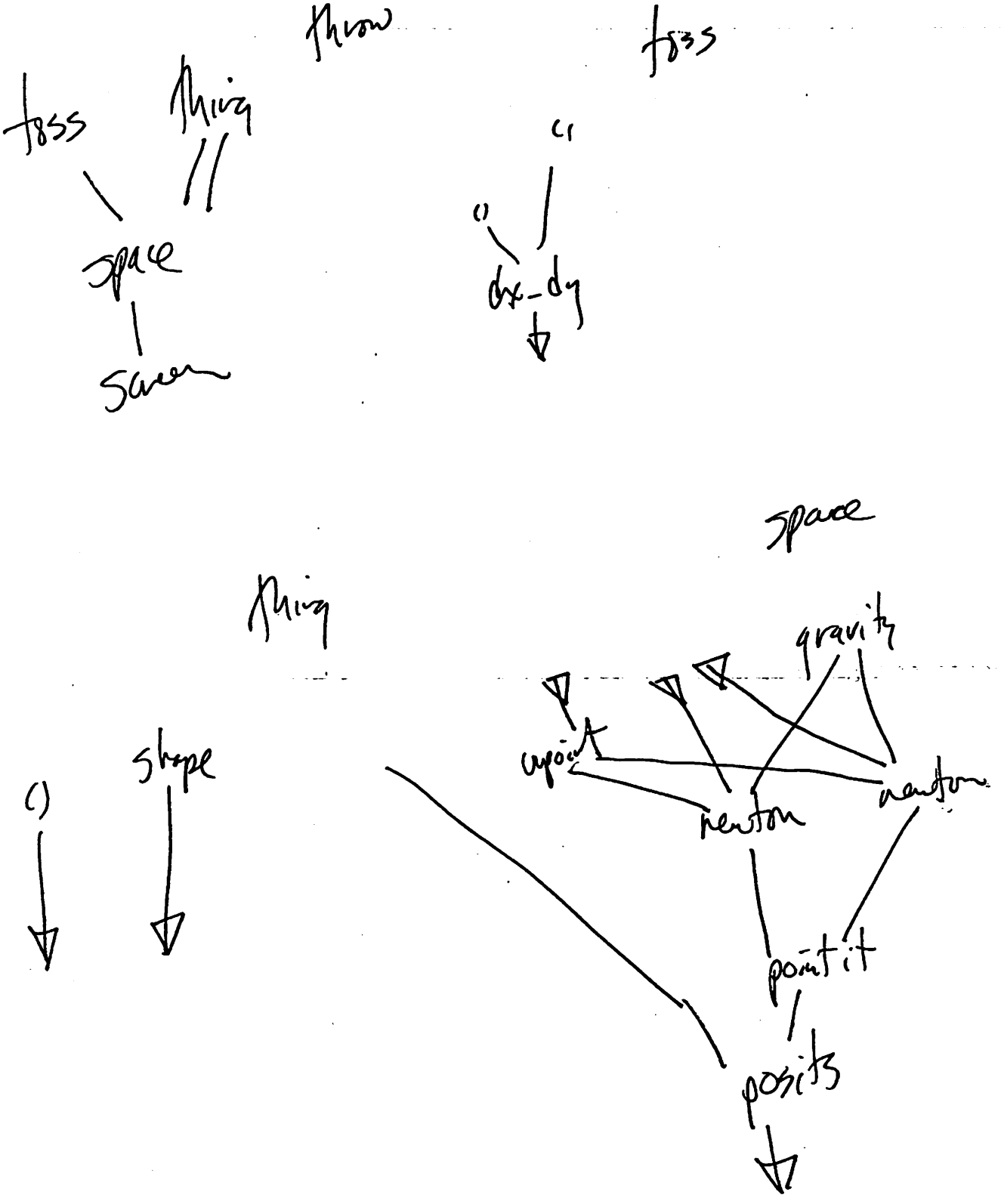


This introduces the subroutineing syntax, using a class of nodes called UDNs (for Used-Defined Nodes). 'parabola' is defined inside the editor, ~~in straightforward fashion~~, using the nodes 'source' and 'sink' to pass arguments into and out of the sheet. These argument passing nodes ~~are named~~ contain names, specified by the animator during linking, and these names are used to link to instances of UDN node on the calling sheet. To invoke an instance on a calling sheet, the menu button 'used defined' is hit via lightpen, and the name is typed at the console. ~~This~~ An instance of the node appears, and ~~instance~~ is ~~then~~ pulled unto the sheet and linked ~~to the~~ as if it were a BIF. No other action is required, with the system resolving all internal links, ~~and subroutining overhead.~~

Only one sheet is viewed at a time, and to facilitate moving around the various levels of UDNs for complex sequences there exist menu buttons called DIVE and POP. Hitting the DIVE button and chosing a UDN causes the editor to display that particular UDN sheet, and make it available for editing. POP returns back to the calling level. This is not limited to single level motions, and traversing the entire tree of script by this method is conceptually clarifying. Subroutineing is thus implemented in a comcpletely clean manner, allowing for conceptual clarity while moving through the scripts, and requiring no management by the user to resolve links or keep track of status. 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

CAnsider the following sequence:



On the top level, called 'throw', a 'toss' and a 'thing' are inputed into 'space', whose output is displayed on the screen 'screen'. Inside 'toss' is simply a vector specified graphically. The position of the two nodes '()' on the sheet is passed into 'dx_dy', which computationally subtracts their x components from each other, and the same for y. 'dx_dy' passes this information as a point back to 'toss' which passes it to 'space' on the top level. 'thing' is simply a defined shape (a shape is a series of points which are to be joined by lines), and a rest position, which is specified by the absolute position of the '()' node on the sheet.

'space' takes these 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 for y the position of the object as a function of time, using the standard mechanics equation, $Sx = Sx0 + vxt + \frac{1}{2}ax\ t^2$. The lambda notation for equations is clumsy to read in EOM, and should would advantageously be replaced by a simple algebraic interpreter.

This sequence simulates an object being thrown, as if from a cliff unto a plain. The turnaround time 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 light pen and pulling the node to a new position. This uniformity between animation action (process) and editing and

initial xxxxx values for parameters (configuration) editing is very pleasing. The value returned by the '()' node is in the current implemenation a constant, but in later versions would be a variable, changing as the simulation moves the objects 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 wholey 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 ~~proper~~ actual shape moves from the rest position node up and accross 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 shapes position on the trajectory.

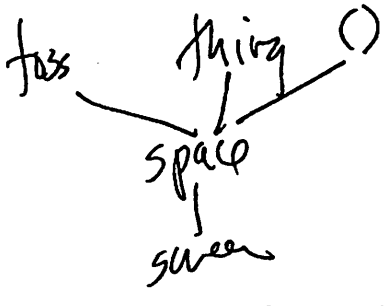
In less trivial examples, where the information extracted from the sheet is more than simply position (say, mass, or connectivity of neural elements), the feedback involved in the process (script) and product (animation) relationship xxxxxxxxx is exceptionally powerful, and not available in any other animation system.

HOOPERS and SCOOPERS

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

A hooper is a functions which allows the specifictaion of a set of nodes to be manipulated together. Manipulations include moving a set of nodes as a cluster accross the sheet to make the arrangement clear or meaningful; or to scoop.

A scooper is the function which allows the moving of a node of 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 'space' as the place to imbed the hoop, the action scoop is invoked by light pen. This places the node '()' inside of 'space', automatically resolving links, modifying internal argument lists and adding the necessary 'source' node inside of 'space'.

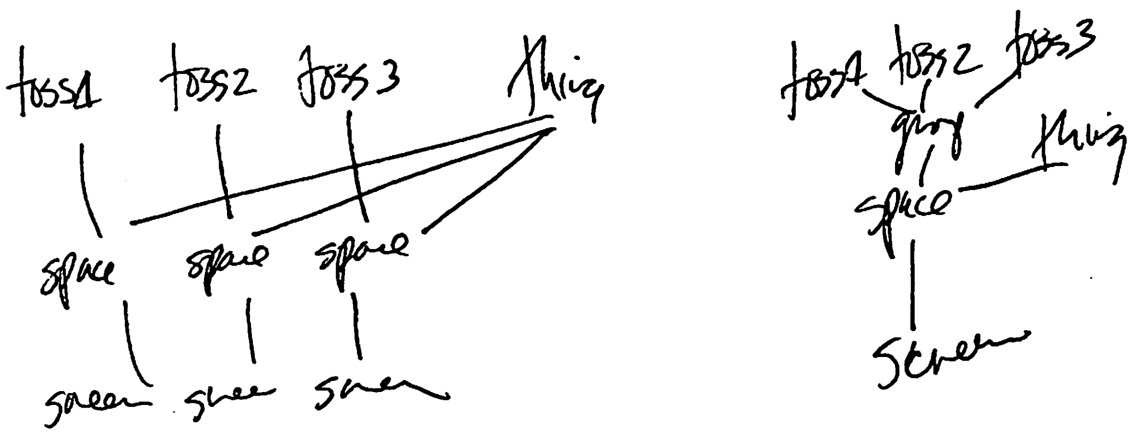
The system will also resolve editing actions such as the modification of subroutine input/output lists even while higher level invocations of that routine are rendered inconsistent by the editing. For example, when an input is deleted from a UDN, sheets which call that UDN contain links to it which are no longer valid. The system will automaticall 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 is found that the use of conceptually simple parcelling up of functions and activities of a simulation, by producing 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/PARRALLEL

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 parcelling data items. At any point in the scripting, multiple instances of any type of node can be avioded (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:

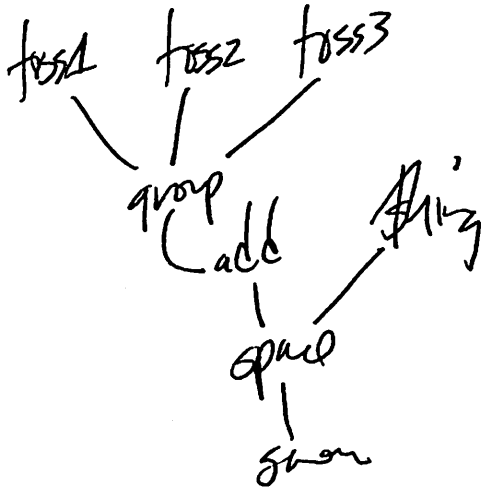


The group in the right figure passes each vector from each toss, in turn, to the 'space' node which maps its other non-group input arguments per invocation. Inside the 'space' node, each downward link passes the group of three data items, which at each node are individually mapped as before, and so on down the tree.

If 'thing' were replaced by a group of three different things,

the result would be three objects thrown a different way. Thus, if more than one input to a node is a group, the elements of each group is matched, and mapped unto the node one by one. Groups may be of arbitrary structure, that is, groups of groups are handled consistently. This is discussed further under the section on internal orgainization. Any data item of the system may be grouped at any level in the network which 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 interpretor, which results in simple coding for the BIF library

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 thrown 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 parcelling relieves the need for multiple instances of 'add', and in practice can reduce the number of nodes greatly, while also contributing to conceptual clarity. The later example under the section on entity simulation has an example of this.

in 19

MEMORY

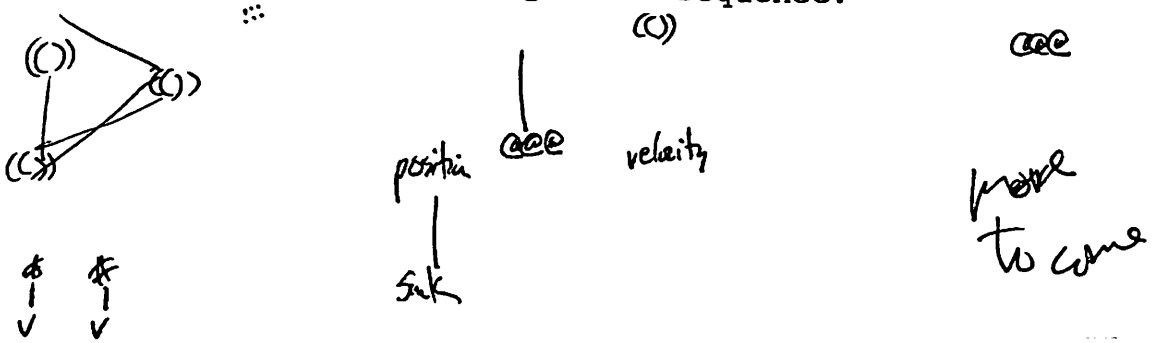
The addition of memory to the system is necessary for a ~~greatly wider~~ class of modelling to take place. Because of the multiple sheet leveling of the scripting, a simple syntax had to be devised to distinguish between ~~xxxxx~~ global ~~values~~ parameters and local instances. All variables in the 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 ~~xxxxxxxx~~ absence of an input link indicates that the value of the variable is defined somewhere above (that is, back ~~on~~ up the stack of ~~xxxx~~ 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 to that sheet, though UDNs invoked from (below) that sheet may ~~via the scheme~~ access its value by name, as just described. If the value of the input link is null data, then the value of the variable is not changed during this time frame. If a value is present, then the value is updated at the end of the time frame, so that during the next frame this value is available at the output of the variable. In practice, this scheme is quite sensible and intuitive, and allows a flexibility of memory organization which is ~~quixx~~ clear. IN → ~~Ring-buffers, flip-flops, sequencers, and echo-images are easily implemented, XXXXXXXXXX (see the next example for echo-images).~~ The next section shows examples.

ENTITY SIMULATION

Two-dimensional scripting suggests an easy way to explore a wide variety of simulation classes, notably automata evolution experiments.(ref--Pask or whatever). A favorite class of entities are described in the following sequence:



The entity '(())' is attracted to other entities' of the same type as if by rubber bands, that is, linearly proportional to distance. The top level sheet called ':::' contains the instances of '(())', and their relationships. The output of each entity is its position on the sheet, which is grouped and saved in the variable associated with that group. Internally, on the '(())' sheet, are posistion and velocity variables, and a call to '@@@', which computes the attraction ~~component~~, drag components of the velocity change, updates the posistion, and outputs these back to '(())', which returns the posistion value to the top sheet ':::' for the next frame time.

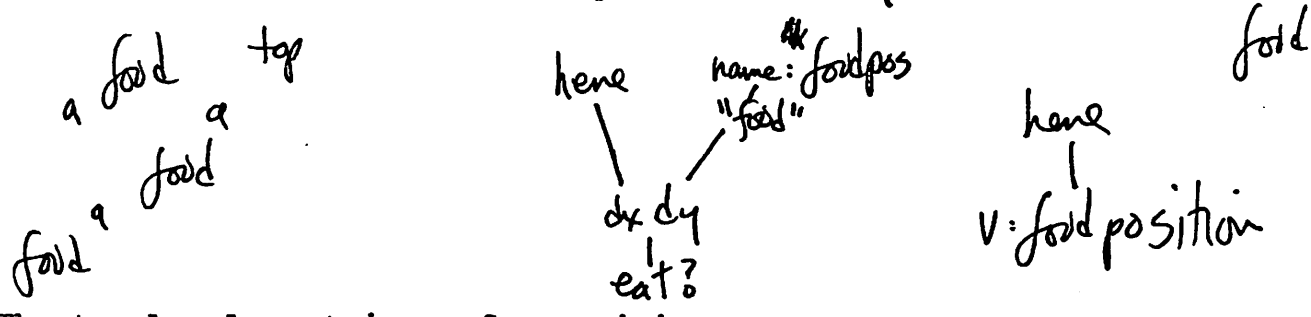
'trail' simply takes in the current position of the entity, and saves via the shift register (implemented by the string of variables) the last five positions. These posistions are grouped and linked to the shape positioner node 'posits', which places the shape (of a single point) at each position in turn. The 'colors' node changes the color of a shape. The various colors are grouped and determine the color of each point in the trail. The group of five colors coming from the 'color it' node are matched to the group of five positions grouped from the variables in the shift register, and the result is five points of varying color (see the cover of this issue of machinations for a sample frame).

It can be seen that the instancing of variables makes sense, and 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 on the '(())' sheet. Note in ':::' that the two variables drag and delta_time are simply set as constants; they are accessed inside of every invocation of '@' in the way that makes sense. The input link has been found to be appropriate and intuitive.

ATTRIBUTES

The limitation of variable usage just described is that information can only be accessed below the instance of a particular variable. Fequently it is useful to pass the information accross the network, particularly in entity simulations. This is particularly true when the modelled environment is uniform, that is, all entities interact with all instances of entities on the sheet. eliminating the need for links from all entities to all others is conceptually better and makes for practical scripting. The following resolution of this weakness, though not implemented, is consistent with the current system.

Consider this hypothetical script:



The top level 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 the node 'name' to indiatce the name of the variable in the sheet 'food' to be extracted. When encountered by the interpretor, these nodes cause the search of the data base for ~~all instances of~~ the variable named 'foodposition' in all instances of the node 'food'. These are one by one

passed into the node dx_dy on sheet 'a' to determine if that food morsel can be eaten, or whatever.

This is expressely the all condition, since the case of 'a' being related only to specific instances of 'food' is included in the case above with '(())', wherein specific connections are made to ~~specify~~ indicate the instancing. Also, the extraction is made accross on the current level in the network as a first restriction on implementation, since it is not clear what the implictaions of multi-level extraction of attributes in the network are.

INTERNAL ORGANIZATION

The keystone of the EOM system is its interpretive executor of the network, called the evaluator. ~~as designed by Steinberg and expanded by McCann.~~ The evaluator operates on the data base which 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 data base which defines the particaular script.

~~DATA BASE~~

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

Items are data base elements such as numbers, booleans, colors, shapes, points or shapes, which are passed down links from node to node as arguments, and ultimetely, 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 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 (such as rotate, which takes a shape and an angle, and rotates the points of the shape).

The internal data base 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 ~~particular kind of~~ node. The node instance contains pointers to the nodes which are ~~xx~~ inputs and outputs, flags, and a pointer to the description block. The description block contains information about the number ~~and type information for error~~ ~~xxxxxxxx~~ of input and output arguments, argument type information for error checking in the executor, 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. An auxiliary pre-pass can be made to order the nodes in the ~~UDN~~ network to ~~xxxxxx~~ minimize the number of passes over the list in search of nodes whose inputs have been evaluated.

Once evaluated, any node has a value which can be any item type,

GROUPS

Items can also be linked into sets called groups. The executor 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 and only the group argument is broken down. The Bif is called 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

Many of the Bifs have multiple inputs/outputs. Nodes (both Bifs and UDNs) may have multiple inputs/outputs. These are handled by building a vector of pointers to the individual outputs with each referencing node knowing which of n outputs 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:

a) It can accumulate the result 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. Thsi is repeated until the group is exhausted. The final result is a reduced group of values. For example, passing the group (1 2 3 4) to the Bif 'point it' would result in the two-element group [(1 2) (3 4)].

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. Becauuse of this and the delayed propagation, first-in-first-out and circular buffers can be made simply from chains of variables. Circular newtorks can be made provided these is at least one variable in the circle.

CONDISTIONALS

The system has conditional constructs which either 'switch' one of the two inputs according to a boolean truth value, of 'gate' one input according to same. If a 'gate'

is closed, all succedeeding noes 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

kjfogcitc o rt p 4 t

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 levelling 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 which completely blur the distinction between programming and animating. Environements 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) further merged unto the same display area and viewed under a transparent, touch-sensitive tablet, a major stride toward purely visual programming will be made.