Programming And Animating
On The Same Screen
At The Same Time

originally printed in Creative Computing, November 1980

by Paul Pangaro

Note: This article was first published in Creative Computing's issue dedicated to Actor Languages and edited by Ted Nelson. This text is unchanged and the figures are scanned from the original magazine. The text was converted by an OCR process. —Paul Pangaro, pan@pangaro.com

I think there is much more work to be done in the area of matching the programmer's approach to the software environment, and I spread this philosophy whenever I can. My focus in this article is the importance of matching the concept of a system to a sensible programming structure, for ease of translation from idea to executable software.

This has an important bearing on the problem of conceptual clarity -especially clarity in multi-process simulations. Multi-process systems are of increasing interest to computerists, for they are closer in spirit to the complex environments which we encounter in real life, and very different from the simple checkbook programs or Adventure games. So, in a Space War I have independent ships, missiles, heavenly bodies — each affects the others, all must be processed every time step. This is where Smalltalk (and other actor-based languages) make implementation so straightforward. The language itself contains constructs for multiple processes and, most importantly, communication structures between them. It is substantially easier in a language with multi-process programming facilities already built for it. Concepts like Class bootstrap the programmer into the bliss of specifying behaviors directly and without repetition. (In Basic, arrays of objects would be required, with explicit code to scan the arrays, pass parameters, test for conflict; all processing having to be under the programmer's elaborate and cumbersome control.) Actor languages also allow you to program faster and help you find those structures which preserve, and are sympathetic to, your own personal approach.

I came upon Smalltalk several years ago and was struck by the intelligence of its general philosophy - with one significant reservation. If one is interested in multi-object simulations for graphical display (a premise of Smailtalk's originators), then why am I forced to continue specifying my instructions in a linear, character-by-character, typed language?

Moreover, I want to make pictures with the computer, so why couldn't a system be made where the programs themselves were pictures (or at least two-dimensional things)? Might we not blur the distinction between programming and animating?

I was fortunate to be at a research laboratory at MIT that had both the facilities and the expertise to explore this idea. The result of this collaboration was a system called EOM, whose name has lost its historical meaning. EOM was essentially an actor language, but one in which interconnection of actors by messages was visually configurable. Our implementation used a lightpen on a vector display with menus for programming and scripting animations.

The fundamental idea of EOM was that all programs are two-dimensional scripts, whose graphical nodes themselves stood for executable programs. Two aspects were both controlled from the screen: the data paths of the program, and the graphics of the intended animation.

Our script convention of data flow is that that lines drawn into the top of a node are inputs, and lines from the bottom outputs. Such links are the paths data flow.

First let's look at a straight program example:

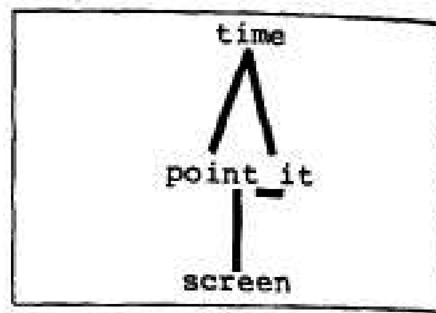


Figure 1.

The script in Figure I simply moves a point diagonally across the frame. The node *time* outputs a value — an ascending integer — as a function of absolute time in the sequence. (This value is essential a frame number.) This value takes a split path to both the x and y input of *point_it*. The object *point_it* now sends two values (in this case, its inputs) to screen, an actor which displays it.

The real advantages for graphics are to come when we add in picture elemetsL Consider the following script.

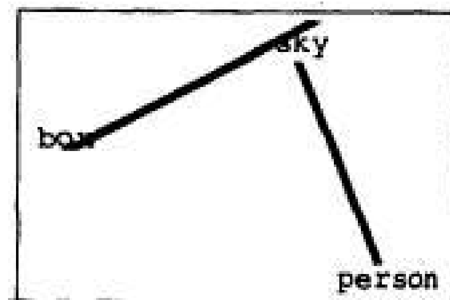


Figure 2.

This program will produce a box sailing across the frame. The outputs of box are essentially its shape and position which change. These variables are repeatedly sent to sky. The position of the word "sky" indicates the highest point the box is to reach.

The box here is like a subroutine, which might be defined like this:

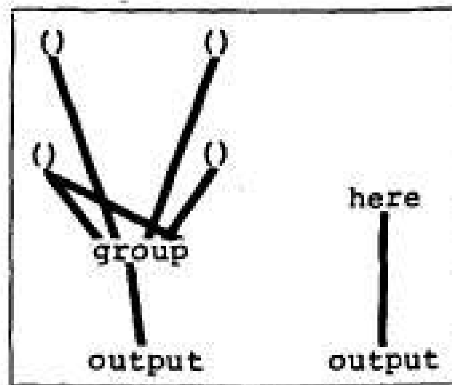


Figure 3.

The () nodes indicate positions on the screen; their output is therefore their positions x,y. Thus to change the shape of the box, you may grab any of the () nodes by lightpen and move where you like, defining the new shape.

The *group* node assembles the positions and outputs them from the subroutine. (The five connections to group complete the closed figure of four points and four lines.)

The *here* node happens to output a position also: in this case, the x,y position of the node box in Figure 2. This is passed as output out of the box subroutine, and linked to the input of sky (in Figure 3) which uses it to fix the initial position of the box that appears in the animated sequence.

Thus moving the position of the *box* node at the top level changes the initial position of the box in the animation — a simple change which does not require the usual calibrating and measuring: 'The box is too far to the left by 10%, the x,y are 0 to 512 so we must add 50 units to the x position which is currently 120 so the new position is 170, type that in . . . " None of that nonsense — just grab the box and move it to its new position. I cannot imagine anything simpler.

The position of sky in the upper level is also significant, since it is programmed to determine the precise trajectory of the box. The person node is simply a metaphor, for it is defines thus:

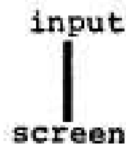


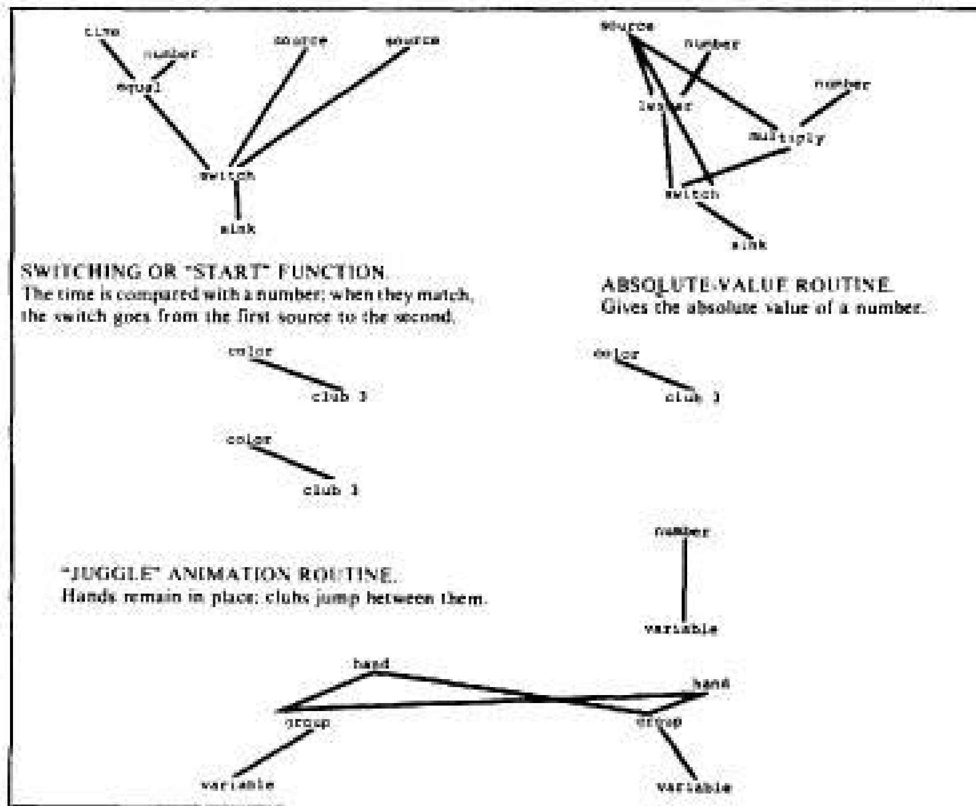
Figure 5.

This "person" takes all its inputs and displays them on the screen: I see this as answering the question of whether a tree failing in the forest makes a sound even if nobody is there to hear it.

The system was found by users to be most pleasing to interact with, and was extremely helpful in animation production. (We managed to film sequences for the science series NOVA, under the usual absurd production schedules, for which the system was superb.)

As an environment for education, EOM has the advantage that a student can perform many simulation experiments, knowing nothing about Cartesian coordinates or programming. Given a set of simulation models, all possible degrees of freedom can be expressed graphically as described above, making interaction simple. For the knowledgeable student, the models themselves can be manipulated, building on the uniform and extensible environment.

Alas, after many months of glory the system died when the hardware configuration was dismantled for newer research, and all that we learned must await another propitious time for further development. Its spirit is carried forward by many, and Henry Lieberman at the MIT Artificial Intelligence Laboratory has extended the concept substantially with his Tinker system.



SWITCHING OR "START" FUNCTION.
The time is compared with a number; when they match, the switch goes from the first source to the second.

ABSOLUTE-VALUE ROUTINE.
Gives the absolute value of a number.

"JUGGLE" ANIMATION ROUTINE.
Hands remain in place; clubs jump between them.

- end -