Computer Animation
in the World of Actors and Scripts

by
Craig William Reynolds

Bachelor of Science in Computer Science and Engineering
Massachusetts Institute of Technology
1975

Submitted in Partial Fullfillment of
the Requirements for the Degree of

Master of Science

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 19, 1978

Signature of the Author Signature redacted
Department of Architecture
May 19, 1978

Certified by Signature redacted
Nicholas Negroponte
Associate Professor of Computer Graphics
Thesis Supervisor

Accepted by Signature redacted
N. John Habraken
Chairperson, Department Committee for Graduate Students

The work reported within was supported by:

IBM (from 10-1-76 to 9-30-77)
Research Agreement with MIT
dated October 1, 1976

ARPA (from 10-1-77 to 2-28-78)
Contract number MDA 903-78-C-0039

A B S T R A C T

Computer Animation
in the World of Actors and Scripts

Craig William Reynolds

Submitted to the Department of Architecture on
May 19, 1978
in partial fulfillment of the requirements for the degree of
Master of Science

A technique is presented for producing animated video images using a special purpose computer system. The Actor/Scriptor Animation System uses a written script to sequence and direct the actions of actors. Actors are independent, animated program structures who serve as "puppeteers", invisibly controlling the formation and ongoing modification of three dimensional geometrical models, hence giving them animated motion. Geometrical operations may be specified either in terms of an object's own frame of reference, or in terms of the scene as a whole. These animated models are "seen" from a similarly animated point of view (the "camera") producing a two dimensional perspective rendering of the view as color video images. Each "still" of the animation is produced seperately and the entire sequence is assembled onto a magnetic video disk for eventual playback.

Thesis Supervisor: Signature redacted

Nicholas Negroponte

Title: Associate Professor of Computer Graphics

T A B L E O F I L L U S T R A T I O N S

I N T R O D U C T I O N

This report is a computer system user's manual posing as a Master's thesis. It describes work done at The Architecture Machine Group (of the Massachusetts Institute of Technology) over the last year and a half. The work was not directly in "computer animation" but rather in the design of tools and techniques, aids to the production of computer animation. Tne resulting "Actor/Scriptor Animation System" is a technique for producing, from written instructions, images of animated three dimensional objects. The first specific application of this work is anticipated to be in connection with ongoing research in dynamic mapping techniques. This document is intended both to describe this approach to animation, and tutor the prospective user of the system.

A N I M A T I O N

To animate means "to give life to", a lifeless ink line on a piece of paper can be made "alive" by the process of animation. The term "animation" will be used in this report to mean the illusion of motion we perceive when a series of incrementally different still images are presented to us in quick succession. We tend to blend the multitude of "snapshots" of an object into a single object in motion.

The first animated pictures were probably "flip charts" of hand drawn sketches. Later, mechanical devices such as spinning disks and drums we used as "light choppers" to regulate the time each "snapshot" or "still" was visible. With the invention of photography, and much later, the invention of flexible film base, it became possible to make animated photographs, "movies". Cinematography also became the media of the hand animator. Each hand-drawn image would be photographed onto successive frames of the movie film. This film, when projected at standard rates produces vivid, high quality animated artwork. With the introduction of "talkies", movies with synchronous sound tracks, animation could be combined

with recorded speech and music. The desire to transmit moving pictures electronically led to the development of video, and television broadcasting. But before video could be used as an animation media, there had to be a convenient way of assembling individual frames into a video sequence. Within the last ten years the development of the video disk ("slo-mo" machine) has provided this capability. More recently "floppy" video disks have brought the price of this technology down from the upper stratosphere.

By a fortuitous happenstance, computer systems able to produce color video images became available at about the same time as low cost video disks. These two advancements led to the technical ability which is the basis of the computer animation system described in this report.

The fact that the production of the stills, and the assembling of the stills into a sequence, are seperate operations leads to some of the unique features of animation. While a frame will only last for 1/30th of a second during playback, the animator may have thought and sweated and scratched his or her head for hours over the production of that one frame. This ability to "compress effort" leads to some of the most striking animated effects. Most artforms share this quality to some extent, a painter spends more time painting the canvas

than even the most ardent admirer will spend looking at it in a museum. However in animation we have complete control of how long the audience will be able to look at a sequence. A short segment of an animated sequence, though it may flash by in a few seconds, can have extremely rich texture and detail. The result can be dazzling. On the other hand, animation can be soft, and gentle, and restrained. It is this "dynamic range" that gives animation such great potential.

Because animation has an element of time in it, some features of animation are similar to music, dance, or the theater. These dynamic artforms deal largely with matters of timing (beat, tempo, pacing). To effectively produce animation, tools must be provided which allow us to talk about time structure. In music there is a notion of "beat", a synchronizing time signal that is used to pace the other aspects of the music, such as pitch and harmony. Animation is closely tied to a similar "clock" mechanism, the frame rate. But most features of an animation will last for hundreds or thousands of frames. The "characters" of an animated sequence may interact and modify each others actions, while the musical notes of a song are concurrent but largely independent of each other. Animation production is largely a matter of describing ongoing dynamic processes, and their interactions.

Why produce animation with computers?

One of the worst reasons to do animation with computers is to replace the hand animator. Something made by hand is to be appreciated for that reason. Both handwriting and typescript can be used to convey the written word, but each has a different purpose to which it is best suited. Handwritten business letters, or typewritten love letters, both seem somehow out of place. Stretching perhaps an analogy, a script based computer animation system is something like an "animation typewriter". Such a system could produce anything a hand animator could, although with perhaps somewhat less individuality. It is worth noting that handwriting has not been replaced by the typewriter.

There will always be a demand for hand animation just as there will always be a demand for painting or any other artform. Some of the work now done by hand might someday be done with computers. But some (perhaps most) of the animation that will eventually be produced with the aid of computers is simply not being produced at all now. Because of financial, technical, or artistic limitations, people who might otherwise have an

interest in animated pictures or diagrams have no means of producing them.

If making animated pictures was as easy as typing a letter, many people would come to find uses for animation. The teacher who needs instructional visual aids, the executive trying to get a plan understood and accepted by coworkers, the researcher trying to visualize the spatial relationships of experimental data, or the computer animator as artist - creating a work of art for its own sake, all would find applications of animation that do not exist today.

Computer systems in general are becoming smaller, cheaper, faster, more powerful, more widely available, and more comfortable to use. To some extent, any technique based on computers must follow these trends.

The computer is an ideal animation media for another reason. Most of the dynamic arts, music, theater, dance rely on a team of people working in concert to produce the final artform. As more and more individuals participate, more and more "events" may occur at any given instant. This increasingly complex time structuring gives a work depth and interest. When there is more going on in a performance then we can easily follow, we become drawn in, straining to catch those details we keep just

missing. In the theater, when we wish to add a new character, we hire a new actor. But with computer animation, we can merely ask the computer to split its attention, to "play" both "parts". Because a computer can "split" any number of times, the number of individual events in a computer animated sequence is essentially unlimited.

Why produce computer animation with scripts?

The Actor/Scriptor Animation System is based on the notion that it is intrinsically worthwhile to be able to describe an animated sequence in written form. A "script" is basically a long string of typewriter keystrokes (characters), and these written symbols completely describe the animation. To understand the impact of a notation for animation, consider the difference between music and dance. Music has a very old, standardized notation. Once a piece of music is committed to paper it becomes permanent and reproducible to a certain degree. Music which only exists in audible form is transitory and subject to matters of perceptual (mis)interpretation. The musical notation removes these ambiguites. Dance on the other hand does not have an accepted, precise notation (notations have been developed, but they have not been widely used). The nuances of the dancer's movements, or the advice the choreographer gave the dancer during rehearsal, are not fully described in written form. Because of this lack, a particular choreography will be lost to the future except as it may be passed from older to younger dancer. Even if we where to videotape the dance performance (with a "camera of record"),

the information about how it was made is lost.

The point is not a concern for the archieval quality of an artform, but rather the benefits that accrue from having a complete written description. If the master videotaped copy of a piece of scripted computer animation is lost, a new master can be "run off" by having the computer follow the script again. Because this is an automatic process, it can be done cheaply and easily compared to the original effort of preparing the script. Going from the formal script to videodisk requires only computer time, no people time. The cost of computer time continues to go down, the cost of people time continues to go up.

The fact that automatic production is comparatively cheap means that "retakes" can be easily made. After doing a trial production of a script the animator may see things which need to be changed. By merely altering the script (with the computer system's text editor) to reflect the desired changes, the script can be "executed" again by the animation system, producing an incrementally improved version of the animated sequence. This process of revision can be repeated until the animation meets the animators judgement. In tradtional "cel" animation, altering just one aspect of a sequence usually means completely redrawing many individual "cel"s.

The ability to reproduce a sequence from a modified script is an advantage over other types of animation, but there is an associated cost. Because a script must contain every detail necessary to specify the production, it is much more work to prepare a formal script than the loose, informal "story board" which is used to "script" a hand animated sequence. A story board is a series of sketches of "typical" frames from significant parts of the sequence. This description is "informal" because it is still open to interpretation. Leaving one detail out of the story board is not critical, someone will probably notice and fix it eventually. From the story board the animator draws "key frames", artwork which will actually appear in the finished production. The key frames are widely spaced throughout the sequence, many frames are skipped over, this serves to "rough in" the motion. Then an army of people known as "in-betweeners" draw the intermediate frames by mentally interpolating between the key frames.

The majority of the effort in hand animation is in the in-betweening phase. And it is fairly "mindless" work, it does take skill to be able to draw the frames, but there are so many of them, and they are so much alike each other. In a scripted computer animation system the large majority of the effort is in the writing of the script. Luckily scriptwriting

is not a very intrusive activity, you could do it at the beach for example.

Because we are constrained to put all of the control of the animation into the script, there can be a frustration factor caused by having only indirect control over the animation. The feeling can be a little like threading a needle while wearing thick rubber gloves. These problems are a direct result of the design of the system, if an individual animator feels unable to work in this indirect manner, they should use some technique other then script based computer animation.

Since the basic control of the animation system is through typed dialog, the ability to draw (by hand) the thing we wish to animate is not very important. While it is possible to convert hand drawn information into a form usable by the animation system, it is not necessary. Two benefits follow from this fact; people who cannot draw well can still produce animation, and animation can be produced on subjects whose visual appearence is not known by anyone. The latter is often called "modeling", deriving the probable actions of a large system by knowing only the nature of the component parts. For example, we might not know the general "shape" of a particular atom, but by knowing the properties of the subatomic particles, we can get them each to interact according to those

properties. The resulting simulation will show us what might go on in and around the atom, at least to the extent that our model of the subparts is accurate.

Perhaps the most powerful advantage gained by using a script based animation system is the notion of a "utility library", a concept from computer system design. When ever a programmer writes a program which solves a problem of general interest, the program is put in a "library" of programs. In the future anyone needing to perform the same computation need only use the program which already exits in the library. This saves the programmer time by eliminating duplication of effort. The library itself can only get larger and more useful as time goes on. Because an animation script is essentially a computer program, the same principle can be used for scripts and scripting utilities. Once we write a description of how to make an object "bounce" (for example), the "bounce" program goes into the library. From then on, no one need ever rethink the process of bouncing. In "cel" animation the only advantage to having done a bouncing object before is the personal experience, which is of no use to your coworkers.

P A T H S N O T T A K E N

Our work consists both of what we do and what we choose not to do. This section notes some other approaches to computer animation and how they differ form the Actor/Scriptor system.

The first computer animation was produced by programs written in standard programming languages (assembler, Fortran) which controlled refresh vector-drawing displays. [32] The programs where fast enough so that they could redraw the scene 20 to 40 times a second. By slight modification of each "frame" an animated motion is produced. More recently techniques have been developed to produce animated, shaded color raster-scan images of three dimensional objects at these rates. [33]

But these "real time" approaches have a "ceiling". If the complexity of the animation crosses a specific limit, the system will become overloaded. At that point the system will either "fail" (produce an erroneous image) or slow down, thus ceasing to produce real time animation.

The Actor/Scriptor Animation System recognizes this problem

and presumes that in general it will not operate in real time. This is why the system includes a video disk. As the complexity of the animation increases, the time taken to produce each frame also increases. Hence the rate of finished frames going to the video disk slows down, but does not stop. When played back, the result will always be "real time" regardless of complexity.

One of the most useful "shorthand" techniques in traditional animation is "key framing". This is a process by which the animator can specify an animated sequence by an initial and final frame and some kind of interpolation rule for going from one to the other. An assistant animator called an "in-betweener" then creates the intermediary frames. Computer animation systems have been built based on this technique. [3, 22] The main problem with these key-frame systems is that they fail to do the same job as the in-betweeners. The computer interpolaters operate at too low a level, they are given only geometrical information about the first and last frames. They have no information on the physical constraints of the motion, whereas the human in-betweeners "know" how objects should move from everyday experience. Consider the example of an animated automobile driving around a curved section of road. A naive interpolater will drive the car straight from the start to the end of the curve. The in-betweener knows to drive along the

road.

User-oriented computer systems often consists of a fixed number of "commands" which can be used in any combination or order. Such a system is usefull only if everything the user wants to do is well described by the set of available commands. A more flexible system design would allow new features (new commands) to be added to the system, describing them in terms of the existing features. Because the Actor/Scriptor Animation System is such an "extensible" system it can be expanded to include any number of new techniques. For example, a naive (linear) interpolator has been written for this system. Hence the animator is free to use the key frame technique (or any mixture of key framing and other techniques), but is not limited to it. The commercial Synthavision computer animation system [30] is an example of a system which is made bulky and hard to use because it is not extensible. Every new script for Synthavision must be started from scratch, it cannot build on previous work. However Synthavision does yield beautiful images to the patient animator.

When computer animation scripting languages are based on conventionial programming languages the control structure of the language is imposed on the script, and hence the

animation. Most programming languages do not provide a way of talking about concurrent events. Hence it is unwieldy to write a program/script which directs actions which are both concurrent and yet independent. Many authors have noted the applicability of the notion of "parallel processes" to such concurrent/independent control problems - in animation [15, 16, 17, 29] and other areas [10, 11, 12, 13, 31].

More recently, programming languages have been designed which allow descriptions of concurrent control structures. The SIMULA "class" [2, 5], the SMALLTALK "class" [7, 8], the MODULA "module" [34], and the PLASMA "actor" [10, 12] are all linguistic techniques for describing concurrent systems.

Both the Actor/Scriptor System and Ken Kahn's animation system [17] are based on this notion of "actors". The two systems differ in emphasis. Kahn's system focuses on issues of "common sense" intelligence on the part of the actors, to allow then to "participate" in the script in a mannor befitting their role. (A car in Kahn's system would know enough to drive around the curved road.) The system described in this report has been much more concerned with three dimensional geometry and motion, and straight-forward ways of describing them, it assumes that the majority of the intelligence will be provided by the script writer. This has led to the development of a

three dimensional analogue to LOGO "turtle geometry" [1, 9, 19], sort of a "flying turtle". Two dimensional turtle geometry has been shown to be much easier to learn (particularly by children) then more traditional geometric techniques. [26, 27, 28]

T H E

A C T O R / S C R I P T O R

A N I M A T I O N

S Y S T E M

U S E R ' S

M A N U A L

March 8, 1976

revised:
Friday May 19, 1978

The Architecture Machine Group

Massachusetts Institute of Technology

Cambridge, Massachusetts

COMMENTS, SUGGESTIONS, COMPLAINTS (OR PRAISE)

This is the first generally available version of the majority of this material. Any feedback on specific errors, typographical glitches, suggestions for further work, or comments on the general workability of the animation system would be greatly appreciated. Letters may be sent via United States Postal Service to:

Craig W. Reynolds
The Architecture Machine Group
Massachusetts Institute of Technology
Room 9-516
77 Massachusetts Avenue
Cambridge, Ma. 02139

or via MagicSix mail (at the Architecture Machine):

cwr @ AM (the author)
hbb @ AM (current maintainer, appointed expert)
asas @ AM (mailing list of ASAS users)

or via the ARPAnet mail to:

amg.ar @ mit-multics

HOW TO MAKE IT GO

This manual describes an implementation of the Actor/Scriptor Animation System which runs under the MagicSix operating system at the Architecture Machine Group. The animation system is entered by typing the asas command with the directory >demo>asas-dir in your search rules. Alternately, one may log in as "asas".

G E N E R A L I N F O R M A T I O N

This section of the manual explains the concepts used in the Actor/Scriptor Animation System. If you are reading this for the first time, you probably do not want to read this section all of the way through. Perhaps the best scheme is to skim over this section, just getting familiar with the ideas being used, but not going into them in detail. The rest of the manual concerns the actual scripting (programming) constructs and expressions. In each section, just after the section title, there is a list of topics to "see also". These will reference other related scripting features, and will also refer back to the General Information section for the concepts underlying a particular scripting feature.

The eight major topics of the manual are arranged (more or less) in order of interest, and within each topic the subtopics are also listed in order of interest, and so on.

O V E R V I E W

The Actor/Scriptor Animation System is a technique for producing color video computer animation from formal written "scripts". The script is essentially a computer program written in a special purpose "scripting language". The animation system will execute a script without intervention producing as output a series of animation "stills" which are assembled onto a video disk for eventual playback and transfer to video tape.

The script is usually a collection of either "animate blocks" or video production utilties. The animate block is basically a list of cues, these are pairs of "what-to-do" and "when-to-do-it". Most of the cues start, stop or give directions to actors. These are "little gremlins" who pop in between frames to make changes to the picture. All of the animated motion seen in a production is controlled by actors.

The actors main job is to create and modify "geometrical objects" to be seen by the animation system "camera". The geometrical objects are models of three dimensional colored

shapes. In general these objects are "plane faced" like the surface of a geodesic dome, the overall object may be curved, but the surface is made up of flat plates. Some geometrical objects come predefined, and the animator may confine the animation to use only these shapes. Or the animator may define new shapes or combinations of other shapes as part of the script.

S C R I P T S A N D A C T O R S

The terms "script" and "actor" come from the real-world concepts of a theaterical or television script and the human actors who play it. It is the actions of the actors (and crew) of a production which form the substance of a drama. The others involved in the production (the writer, the director, the choreographer, the costume designer) do their work "off-line". During the play their presence is felt, after all it is they who decided how the actors should move, what they should say - and when, and how the actors should appear to the viewer. But on the night of the performance the choreographer (for example) could be just as well be in another city.

These two types of participation form the model that is used in the Actor/Scriptor Animation System. The animation scripter

(analogous to the writer, director, costumer, and choreographer rolled into one) writes a formal description of what is to go on the during the animation (a "script"). This is then given to the computer system, which then produces an animated videotape corresponding to the specifications in the script. As far as the script writer is concerned, no intervention is needed (or allowed) during production. It could for example, be done late at night when demand on computer systems is lessened.

The scripter's time during the day could better be used either deep in thought, or interactively using the system to try out, visualize, and "rehearse" sequences. I include the notion of being "deep in thought" because frankly, I have never seen any computer system which can directly aid the thought process (in fact they often seem to serve as distractions). This is not to say that a "thought provoking" computer system could not be built, but the nature of such systems is poorly understood at this time. The Actor/Scriptor Animation System makes no pretense of being able to aid the animators in their deep thinking, in fact it is expected that the majority of complicated script writing would be done "off line". Every room where people are expected to think should be equipped with one blank white wall, devoid of any detail, in order to encourage "staring at a blank wall".

Possibly the largest advantage to structuring the task of animation production in terms of scripting and automated production (although some would call it a disadvantage) is that we effectively transform the art of animation into the art of computer programming. This is not to say that "traditional" animation is somehow inferior to programming, merely that it is very different, and that is good. If we open the world of animated pictures to those who are at ease with programming, we will have greatly augmented the pool of talent from which animators are drawn. This can only be a positive contribution to the overall quality and usefulness of animation.

G E O M E T R I C A L O B J E C T S A N D O P E R A T I O N S

The images we will see in an animated sequence from this system will be views, as seen by the camera, of the geometrical objects which the script is manipulating via the actors. Geometrical operators are the "tools" used to manipulate the objects. The Camera (itself, a geometric object) is also under the control of the script and actors. No other component of the animation system is ever visible. We

cannot see actors or programs for example, because they have no geometrical description. Certain geometical objects are also defined to be invisible for convenience sake. "Cameras" are presumed to be too small to see, so that if we happen to have a few spare cameras in the middle of everything they will not be seen in the animation.

The geometrical objects are best thought of as models (or descriptions) of "real" three dimentional solid objects. They are similar to a "blueprint", telling the graphics subsystem how to "build" an object. Some objects come predefined by the system, others can be defined by the user. But all geometrical objects are handled by the same geometrical operators. The move operator can be used to move a vector, a camera, or a group which contains a model of an entire city. The operator itself decides how to treat the object it is given.

The basic sorts of geometrical operations which can be performed are; growing and shrinking, changing position, rotations, stretching, and changes in color. Operators are also available to group seperate objects together into one conglomerate object. Because the scripting language is extensible, new operators may be added by the user simply by writting programs using the predefined operators. The geometrical operators take in one object and return another

object. The operators NEVER modify the particular object they are given as input, the result of the operation is a modified copy of the original object.

N O T A T I O N

See also: Expressions, Evaluation

The scripting language used in the Actor/Scriptor Animation System is what is called an "extension" to the programming language Lisp. This means that the animation system was implemented by writting Lisp programs (they are all called "functions" in Lisp). The scripting constructs are just calls to these animation system functions.

The Lisp notation is a little unique in format, but the format is very consistant. Basically Lisp expressions consist of words and parenthesized lists of words or other lists. Hence the scripting expressions used as examples in this manual tend to consist of lots of nested parenthesis, with words intermixed. The words will either be specific animation system keywords, or they will be examples of user's variables. All keywords in this manual will be underlined. This is only done for the sake of this manual, the computer system does not require the keywords to be underlined.

In each section of this manual which describes a program there

will be a "general usage format" or diagram of how the program is used:

(program-name <input1> <input2> ...)

In these diagrams the underlined keyword is at the left of a list, the rest of the things in the list will either be words in <angle-brackets> or an ellipsis "...". The name in angle brackets is a "meta-name", a name for a class of objects. For example, if a certain program (call it "frob") expected two inputs, a number and a color description, we would write its usage format as:

(frob <number> <color>)

This means that we should call the program "frob" by writing a list with three things in it; the word "frob", an expression whose value is a number, and another expression whose value is a color.

An ellipsis ("...") in the usage format means that the meta-name to the left may be repeated as many times as desired. For example the group program will accept any number of expressions whose value is a geometrical object:

(group <gobj-1> <gobj-2> ...)

If a meta-name is in {curly brackets} it means that it may, but need not appear in the program call.

E X P R E S S I O N

See also: Evaluation, Program, Variables, Numbers, Data-types

Expression is the general term which refers to all of the elements of the Actor/Scriptor language (e.g. programs, actors, lists, variables ...). The evaluator (the function eval) assigns a value (or meaning) to expressions. These values are also expressions.

Because the parts of a complex expression can themselves be expressions the definition of an expression is said to be recursive.

an expression	is an:	atom
	or its a:	parenthesized list
an atom	is a:	variable
	or its a:	number
a parenthesized list	is a:	"("
	followed by zero or more:	expressions
	followed by a:	")"

The value of the various expression types, are explained in detail in the section on evaluation; but numbers evaluate to themselves, variables to their defined value, and lists indicate that a program (whose name appears at the left of the list) is to be run to obtain the value.

A "valid expression" usually refers to one that could be

evaluated without error. For example, the use of a variable which is not declared by any of the expressions which surround its use is defined to be an error. In fact it is an "invalid global variable reference" error. Programs containing such expressions are detected and trapped at program definition time.

The term "<something>-expression" (e.g. "cue expression", "lambda expression", or "loop expression") usually refers to a parenthesized list whose first (leftmost) element is that <something>:

```
(cue (at 10) (cut))

(lambda (x) (plus x 1))

(loop (until (empty snodge))
      (drib dralb)
      (klitz klork klump))
```

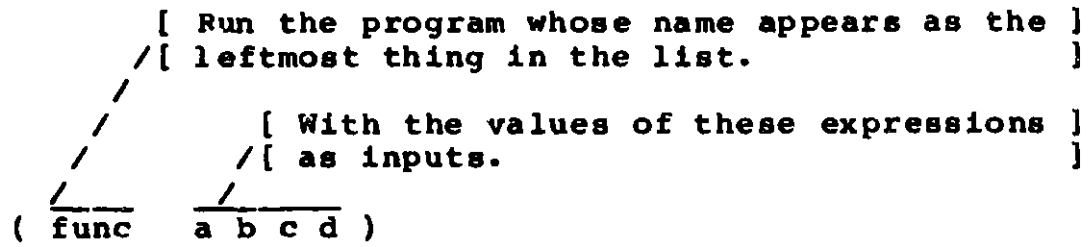
E V A L U A T I O N (a n d E X E C U T I O N)

See also: Expression, Defprog, Actor

The fundamental operation of (LISP and hence) the animation system is called evaluation, the process of assigning a value to an expression. The function which performs evaluation is called eval, it expects one expression as input and returns its value (which is an expression itself). Here is a list of

how various expression types are evaluated (the symbol "=>" is read "evaluates to"):

numbers	=>	themselves.
variables	=>	their defined value.
parenthesized lists	=>	the result of running a specified program with the inputs listed:



A list indicates that a program is to be run to obtain the answer. The name of the program to be run is the first (leftmost) thing in the list, its inputs are formed by evaluating each of the rest of the elements of the list. The input variables of the program are defined to these values, in order, from left to right. When (if?) the program finishes, it returns a value (sometimes called the "return value" or "functional value" or just the "result"). From then on the evaluation proceeds as though the functional value had been written rather then the parenthesized list.

Example (assume "i" is defined to be 48):

```
          (plus  5 (times  (quotient  i  2) 3))
=>
          (plus  5 (times  (quotient  48 2) 3))
```

```
=>
          (plus  5 (times   24              3))
=>
          (plus  5 72)
=>
          77
```

There are two reasons to evaluate a program, for value, or for effect, (or for both). All programs return a value, although when we run a program for effect we often simply ignore the returned value, this type of evaluation is sometimes refered to as execution. Generally, programs without effect (those used solely for value) are easier to understand, debug, and use.

The order of steps in the evaluation process are:

(1) Lookup of actual function from function's name.

(2) Evaluation of the input expressions (left to right).

(3) Saving of old values of program variables, (pushed onto stack).

(4) Binding of new values of program variables.

(5) Evaluation of program body, (and possible side effects).

(6) Old values of program variables are restored, (poped from stack).

(7) Functional value returned.

"Normal" functions always have all of their input expressions evaluated But we can define special functions which will evaluate their own input expressions in the order they chose,

this allows selective, repetitive, or conditional execution. Most control functions (e.g. loop, if, animate, define) are of this type.

N U M B E R S

See also: Numeric operators

Numbers are used throughout the animation system. Here is a list of the ways that numbers are used:

To measure or specify time, usually in terms of animation frames, or seconds of animation time.

To measure or specify distance, as in vector coordinates.

To measure or specify angles, in terms of revolutions.

To measure or specify size changes, scale factors, and zoom factors.

To measure or specify colors in terms of either; intensity, hue, and saturation of the color; or the individual intensities of each of red, green, and blue.

To uniquely identify each of the active actors (integers only).

As repitition counts, the number of times to repeat something.

As inputs to numerical operators.

We can write numerical constants in several different ways.

Numbers may be positive or negative, and they may be integers or floating point numbers (fixnums and flonums in Lisp terminology). There are two ways to write floating point values.

Integers are used to count things which do not come in fractional parts. Take people for example, we can have 0 or 1 or 2 people, but having one-and-a-half people in a room with you is very messy.

Floating point numbers are used to represent values which may take on any value, even the in-between values. Temperature is an example of this, we can have a tempreture of "64" or of "65" or of "64.8329". Floating point numbers can be written in each of two formats, one is called "decimal point notation", the other is "fraction notation". In decimal point notation, a decimal point "." is used to seperate the integer digits from the fractional digits. In fractional notation the value is written as a ratio of two integers, with a fraction sign "/" in between. Note that in fractional notation there are many ways of writting the same value, since any fraction which can be simplified will be equal to its simplifications.

Integer Notation	Decimal Point Notation	Fractional Notation
0	0.0	0/1
	0.5	1/2
	0.66666	2/3
1	1.0	385/385
-48329	-48329.0	
	25.9	259/10
	1.001	1001/1000

A N G L E

See also: Numbers, Rotate, Left, Right, Up, Down, Cw, Ccw, Ring

The description of a rotation has two parts, the "axis" and the "angle" of the rotation. If we think of rotating our own head, the neck (actually the spine) is the axis, the amount by which we turn our head is the angle. In the animation system we will often be rotating geometrical objects about various axes, either their own self-relative axes, or those of the external coordinate system.

The angle by which an object is to be rotated is given as a number. Rotations are measured in "revolutions" (units which

are easy to understand, but non-standard among mathematicians). One revolution is one complete turn, or 360 degrees, or (2 * pi) radians. Hence an angle of 1/2 means "half way around", the angle between numerals on a clock face is 1/12. This handy chart should help you traditionalists:

Table of equivalent angles in different units.

revolutions		degrees	radians
0	(0.0)	0	0.0
1/4	(0.25)	90	1.570796
1/3	(0.333)	120	2.092299
1/2	(0.5)	180	3.141592
2/3	(0.666)	240	4.188784
3/4	(0.75)	270	4.712388
1	(1.0)	360	6.283184

The number used to specify an angle in "revolutions" is therefore usually between 0.0 and 1.0, exclusive. If a value outside this range is used, it will have its integal part ignored (for example: 4.267 revolutions is the same as 0.267 revolutions). Using an integer value for an angle is not an error, but it is somewhat useless, since all integer values are equivalent to a rotation of 0.0, or "none".

C O O R D I N A T E S Y S T E M S

See also: Vector, Subworld, Self-relative and Global Operators

Because the geometrical objects in the Actor/Scriptor Animation System are three dimensional, we must have standard ways of describing positions and orientations in three space.For humans, perhaps the most natural direction to conceptualize is "forward", the direction we walk. We can easily turn to either side, and hence we have freedom to move in two dimensions. But because of gravity, we cannot easily move up and down. We generally do not have as much personal experience with three dimensional motion. In thinking about geometry in the third dimension it may be helpful to think of spaceships, zipping around in the void of space. In a free-falling spaceship there is not enough gravity to effect the motion of objects. If we place a box in the middle of the cabin it will not fall anywhere, it will just "sit" there floating in the middle of the cabin. SCUBA divers and fish live in a much more three dimensional world than most of us, since they can also move unencumbered in the vertical direction.

Coordinate geometry is a method of specifying positions by means of numbers. "Cartesian coordinates" is one such technique which measures positions in terms of distances along predefined "coordinate axes". We are all familiar with the two dimensional version of this system, many modern cities are arranged on a rectangular grid of streets. To specify an area of the city we use phrases like: "Near the corner of 'B' Street and Tenth Avenue". This specifies a location by telling how many blocks along each coordinate axis ("A" Street and First Ave serve as the axes). Hence in two dimensional Cartesian coordinates we use two "rulers" to measure position, these "rulers" indicate both a distance and a direction ("10 blocks along 'A' Street"). Note that the "rulers" are perpendicular to each other, and that the length of units ("blocks") along each axis may be different.

Three dimensional Cartesian coordinates are just an expansion of this concept into three dimensions. A three dimensional position can be specified as some given height above a two dimensional location. We simply add a new "ruler" which is perpendicular to the other two. In our city example, this would be as though we described how to get to our office as: "Go to the corner of 'B' and Tenth, go into the Schmooha building and take the elevator to the fourteenth floor."

Since we must differentiate between the three coordinate axes, we give them names. The easiest way of visualizing this is to imagine youself at the center of the coordinate system. If you hold your right arm out to the side it will be pointing in the "positive X" direction, your left arm points in the "negative X" direction. The top of your head points in the "up" or "positive Y" direction, "down" is the "negative Y" direction. The direction your nose points is the "forwards" or "positive Z" direction, your tail (if you had one) points in the "backwards" or "negative Z" direction. Since a "negative" direction is merely the opposite of the positive, we can specify a coordinate system's orientation in terms of three "rulers" (or "coordinate axis definition vector") called "x", "y", and "z"; or "right", "up" and "forward". We need only specify where we are and how our three "rulers" are oriented in order to completly describe the coordinate system.

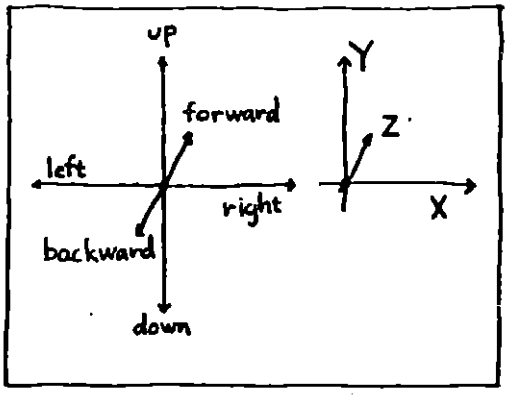


Figure 1

Self-relative directions and the positive coordinate directions.

To specify a location in our own coordinate system (that is, a location "relative" to us) we tell how far away it is in terms of distance along each of our coordinate axes. If we are sitting in a seat in an airplane for example, the person two rows in front and three seats over would be at a relative position of "3 to the right, 0 up, and 2 rows forward". The vector data-type is used to contain this type of information, and as a vector that person's position would be:

(vector 3 0 2)

The upper tip of the tail would be approximately at location:

(vector 0 10 -20)

That is: "none to the right, 10 units above my head, straight back 20 rows".

Two flavors of coordinate systems are used in the Actor/Scriptor Animation System, they are called "self-relative" and "global". The coordinate system discussed in the airplane example is an example of a "self-relative" system. In such a system, the object or agent under discussion is always at the center of its own coordinate system, as they rotate or change size their coordinate system rotates or changes size with them. The rule is sort of "as you go, so goes your coordinate system". By a "global" coordinate system we actually mean "someone else's" coordinate system. An

example of the difference between these types of coordinates would be to refer to some object as "the big red thing to my left" or as "the big red thing in the northeast corner of the room". The first is a self-relative specification, the second uses an external coordinate system (that of the room). The major difference is that a "global" or external coordinate system is not modified as we modify the objects within it.

D A T A - T Y P E

The term "data-type" is used occasionally throught this manual, by it we mean a Lisp data structure which represents some component of the animation system (such as a program, an actor, or a geometrical object). Lisp structures which do not fit into one of these categories are considered "random cruft" and will cause most of the animation system's operators to choke.

The data-types used in the animation system are of the sort known as "self-evident" data types. This means that given a object we can determine what type it is. Consider two pieces of paper covered with digits, a "self-evident" piece of paper might have the notation "The first 10,000 digits of Pi" thus telling us how to interpret the data. The piece of paper with

no title, just digits, is an example of untyped data.

Because the compenents of the animation system are "self-evident" the programs and operators which deal with t..em can do "run time type checking". This means that an operator which is expecting a vector as an input can check what it actually gets to be sure it is a vector. Operators which receive inputs of an incorrect type will stop and complain. For hints on what happened after such an error, see the back-trace and release operators.

C A M E R A

See also: See, Pov, Animate, Script, Geometrical Objects, Geometrical Operators, Local Variables, Define

The animation system provides actors to provide the animated control of geometrical objects, and the geometrical operators with which to manipulate the objects. The end goal of all of this creation and modification of objects is to produce "subjects" for the animation "camera" to take pictures of.

The "camera" is a geometrical object itself (although it is not visible) and so can be manipulated in just the same way as

the other objects. For example (if the "world" contained only the camera and one object) we could move the object closer to the camera or we could move the camera closer to the object, to achive the same effect. A camera is represented by a point of view (pov) geometrical object, and since we can save an object by name (with a "variable"), we can keep several camera definitions around and switch between them.

A point of view has three parameters, a position in three dimensional space, an orientation (which way is up, and right, and forward?), and a "magnification" or "zoom" factor. We think of this as a video camera floating somewhere in space, aiming off in some direction, and with its "zoom lens" set at the specified magnification. One advantage of using an imaginary camera, over a real one is that the zoom lens on a real camera will not have a "zoom ratio" of more than 20 to 1, the animation system's camera has a zoom ratio of about one million to one.

THE camera is a local variable owned by the active animate block. To reference the current camera we just use the variable of that name. To change the current camera we redefine that variable. This sequence of commands saves the current camera (as "cam-1"), modifies the current camera (saving it as "cam-2"), then switches back to the original

point of view:

```
(define  cam-1 camera)
(grasp  camera)
(forward  10)
(zoom-in  2)
(define  cam-2 camera)
(define  camera cam-1)
```

B A C K G R O U N D

See also: Animate, Color, Local variables, Define

The two dimensional image produced by the camera will often contain areas where no geometrical object is visible - what should we see there? The system uses the concept of a "background" color. This is much like a cyclorama in the theater, a large hemicircular, featureless curtain which is behind all of the scene pieces. In the animation system we can consider the background to be a huge sphere which completely surrounds the camera and all of the geometric objects. The color of the background can be changed merely by defining the variable "background" to a new color object.

C O L O R

See also: Global variables, Numbers, Polygon, Solid, Data types

In order to display things on a color display we must be able to specify colors. The animation system uses a data type called color to represent them. Two different functions are provided for specifying colors, they are called rgb and ihs ("red, green, blue" and "intensity, hue, saturation"). Note that the hardware display unit in use allows only 1024 distinct colors per frame.

R G B

IN rgb space we specify a color in terms of how much red, green, and blue light is to be mixed together to get the desired color. Each of the primary's levels may be adjusted from 0.0 (none) to 1.0 (full intensity). The particular color display unit in use allows 256 distinct levels for each primary. Usage format:

(rgb <red-fraction> <green-fraction> <blue-fraction>)

I H S

In ihs space a color is specified in terms of three different parameters:

Intensity: How bright is the color?

Hue: Where is the color on a circular color scale?

Saturation: How vibrant (vs. washed out) is it?

Intensity runs on a scale from 0.0 (none) to 1.0 (full intensity). Hue values form a cycle, and the spectral hues fall between 0.0 and 1.0:

color:	hue value:	
red	0	0.0000
yellow	1/6	0.1666
green	1/3	0.3333
cyan	1/2	0.5
blue	2/3	0.6666
magenta	5/6	0.8333
red	1	1.0000

The saturation values run from 0.0 (no saturation, the color is a gray tone) to 1.0 (fully saturated, vivid color).

Here are some common colors specified as rgb and ihs for comparison. For a full list of predefined colors see the section on Global Variables.

black	(rgb 0.0 0.0 0.0)	(ihs 0.0 0.0 0.0)
gray (50%)	(rgb 0.5 0.5 0.5)	(ihs 0.5 0.0 0.0)
white	(rgb 1.0 1.0 1.0)	(ihs 1.0 0.0 0.0)
cyan	(rgb 0.0 1.0 1.0)	(ihs 1.0 0.5 1.0)
yellow	(rgb 1.0 1.0 0.0)	(ihs 1.0 1/6 1.0)
pink	(rgb 1.0 0.3 0.3)	(ihs 1.0 0.0 0.5)

P R O G R A M S

See also: Defprog, Variables

The features provided by the Actor/Scriptor Animation System are intended to be those which would be most useful to the "average" user. However since no one considers themselves average, there will be times when the user wants to do something which is not provided as a basic feature. The solution to this is to write new programs. It is this ability to add new features to the system which gives it real expressive power. In a non-extensible system, where it is not possible to define new operations the user must be content to use the set of operations provided by the system designer. There is no way however, for the designer to ensure that all of the useful features have been thought of beforehand.

Doing something which would require a combination of lots of basic operations is the usual reason to write a program. This accomplishes several things:

(1) We can name the combination operation with one concise name which from then on will refer to the whole thing. This name is usually chosen so that it clearly identifies the process carried out by the

program it represents.

(2) By refering to the large combination expression by a name we can eliminate the big expression wherever it would have appeared, hence shortening the calling program.

(3) A program serves as an "abstract" solution to a general problem. For example, the program which adds numbers has the rules of addition built in. Once it "knows" these, it can add ANY two numbers. Once a program is written for one particular set of input values, it will work for any others of the same type.

A very useful way to design programs is what is called "top-down" design, this means that while we are writing our script we may decide that the action to take place at one spot in the script is too complicated to explicitly write down all of the details. The solution is to make a new program to do all of the dirty work. But rather then worry about the details of that program (after all we were trying to write the script originally) we just make up a name for the program-to-be and call it from the script just as though it already existed. Later, after we have finished the "top level" script (or program) we will come back and write the "sub-program". Note that there is no real difference between a program and a subprogram.

V A R I A B L E S

See also: Evaluation, Program

To save information produced at one point in a script to be used elsewhere, we need some kind of storage or memory. At the machine level, individual storage cells have numeric "addresses" (e.g. memory location #52,647), but in most programming languages we are allowed to refer to the cells by name (the programming system figures out which name corresponds to which memory cell).

Since the name "stands for" the value stored in the memory cell, we can change the "meaning" of a name by changing the contents of the cell. Hence we say that the name is a variable, standing for one thing now, another thing later. Usually the name is chosen so that it suggests the usage of the value of the variable. For example, in a program which constructs something, the variable "size" may specify how big the something is. We would not need to know the specific value of "size" in order to understand the general workings of the program.

To allow us to be free to use the same name in different programs (to mean the size of a triangle one place, the size of a ball elsewhere) we specify which programs "own" which variables. When a program is entered, space is reserved for each of its own variables, when the program is done that space is recycled for use later. Within the definition of a program (a defprog construct) we can only look at, or change the variables of our own program, or those "global" variables which we explicitly declared to be accessable. This no-global-references rule is also enforced for these constructs: script, animate, and actor.

There are four "classes" of variables in the Actor/Scriptor Animation system, they are listed below along with the constructs which may "bind" them:

(1)	global	defprog, script, loop, animate, actor
(2)	input	defprog, script
(3)	local	defprog, script, loop, animate, actor
(4)	private	actor

From the user's console only global variables are available for use. Within a program only variables which are "declared" may be used. A variable is declared by appearing in a global, input, local, or private expression. These "variable declaration" expressions have the same form as a function, but

they are not executed at run time, they are there to provide information about the program at definition time. The format is a parenthesized list of variables to be declared with the variable's "class" as the first thing in the list. The "class" of a declaration is followed by a colon ":" to differentiate declarations from functions:

(<type>: <var1> <var2> ...)

G L O B A L

Variables which exist but do not belong to the current program are called "global variables". As an example, certain geometric models come predefined. The global variable "cube" is one such. We can use the symbol "cube" from the console (e.g. "(display cube)"), but if we wish to reference "cube" in a program we must specify (for the program defining program) that "cube" is a valid symbol to use in the program, and that the program should not reserve space for it (as it would for a local variable). This is done with a global variable declaration expression:

(global: <var1> <var2> ...)

For example, to declare that the program "thorf" wants to access the global variable "cube":

```
(defprog  thorf

          (global:  cube)
          ... )
```

Globalness is a relative thing, it is like the layers of an onion, each layer is "global" (surrounding, outside) to the ones below it. If the we are in program A and a call is made to program B, the variables of program A are part of the global environment of B. Program B may examine the variables of A by declaring them to be global.

This is a list of the global variables that come predefined when the Actor/Scriptor system is entered:

Name	Value	
t	t	("true")
nil	nil	("false" or "none")
x-axis	x-axis	
y-axis	y-axis	
z-axis	z-axis	
objects	(tetrahedron cube octohedron cylinder pipe ball)	
colors	(black white red red-yellow yellow yellow-green green green-cyan cyan cyan-blue blue blue-magenta magenta magenta-red)	

The variables on the list "objects" are geometrical objects suggested by their names, those on the list "colors" have been defined to be the indicated colors.

I N P U T

See also: Defprog, Evaluation

The most flexible way of getting values into a program is not global variables, but input variables. This class of variable belongs to the program that declares it, space is set aside for it. Input variables get their initial values when the program of which they are a part is "called" (or invoked) from another program. The input expressions in a call to a program are evaluated, and the values are assigned to each of the input variables of the program, matching them up left to right. Consider this calling expression and partial definition for "wacko":

```
...
(wacko  (plus  1 2)
        (vector  1/2  1/4))
...

(defprog  wacko
          (inputs:  size vec)
          ...)
```

After the call, "size" would have the value 3, and "vec" would have the value: (vector 0.5 0.24 0.0).

Because input variables belong to their program, they may be altered during the execution of the program if desired. This will not modify values in the global environment.

I N P U T

See also: Defprog, Evaluation

The most flexible way of getting values into a program is not global variables, but input variables. This class of variable belongs to the program that declares it, space is set aside for it. Input variables get their initial values when the program of which they are a part is "called" (or invoked) from another program. The input expressions in a call to a program are evaluated, and the values are assigned to each of the input variables of the program, matching them up left to right. Consider this calling expression and partial definition for "wacko":

```
...
(wacko  (plus  1 2)
        (vector  1/2  1/4))
...

(defprog  wacko
          (inputs:  size vec)
          ...)
```

After the call, "size" would have the value 3, and "vec" would have the value: (vector 0.5 0.24 0.0).

Because input variables belong to their program, they may be altered during the execution of the program if desired. This will not modify values in the global environment.

L O C A L

See also: Evaluation, Actors, Defprog

Local variables are "extras", temporary values used only during the lifetime of the program. Note that programs defined with defprog to be "called" have a lifetime of just one call, actors on the other hand have very long lifetimes. It is the local variables of an actor which provide its "memory". In a local variable declaration expression we can specify initial values for the variables. If no initial value is given, it is assumed to be nil. The format for specifying initial values is to write:

(<variable-name> <initial-value-expression>)

in the place where we would normally write just the name of the variable. The initial value expression is evaluated at definition time.

Examples of use:

```
(defprog  funf
          (input:  funfee)
          (local:  funfer)
          ...)
```

Defines a program named "funf" with one input, and one local variable.

```
(start  (actor  (local:  (time 0))
```

```
                (inc time)
                ... ))
```

Starts an actor with a local variable named "time", note that the actor increments its "time" each frame.

P R I V A T E

See also: Actor, In

Because actors are an independent lot, they insist on having hiding places where they can keep secret information. These are called private variables. Local variables of actors can be changed from the "outside" by use of the in operator, private variables cannot.

Imagine for example that we wanted to make a "sluggish" actor, when you told it to be somewhere, it would casually mosey over to that location. The actor must prevent outside agents from actually setting its position. To do this we give the actor a private variable to store its real position, the public variable "position" will be used only as the target towards which it moves:

```
(define  sluggish
         (actor  (local:  position object)
                 (private:  real-position)
                 ...))
```

S C R I P T

See also: Animate, Defprog, Video utilities, Local Variables

The top level controller for the production of animation is the "script". The script construct is similiar to the script for a motion picture or television drama. Both serve as a written description of the production, both specify the sequence of the elements of the action. In the animation scripting language, the elements of the script are usually animate blocks or one of the "video production utilties" (such as: black, color-bars, test-pat, and count-down). Add the script specifies the name of the production, and may declare variables for use by the elements of the script.

General usage format:

```
(script  <script-name>
         <declarations-of-variables>
         <script-element-1>
         <script-element-2>
         ... )
```

or:

```
(script  <script-name>
         <script-element-1>
         <script-element-2>
         ... )
```

Examples of use:

```
(script  grunch
```

```
(black       10)
(color-bars  10)
(black        5)
(count-down   5)

(animate   ...)
(animate   ...)

(black   10))
```

Now "grunch" is defined as a "scripting function", it consists of 10 seconds of black, 10 seconds of "color bars" (a standardized color test image), a 5 second count down, two animated scenes (whose details have been omitted), and then more black.

If a script wants to declare that it will use variables, the declarations fall between the <script name> and the <script elements>:

```
(script  frizz
         (local:  velocity
                  acceleration
                  (size 28))
         ...)
```

The scripting function "frizz" declares that space must be reserved for three local variables, "velocity", "acceleration", and "size". "Size" has also been given an initial value of 28 (the other variables will have the default initial value nil).

At the time a script is read in by the animation system (from

a file or from the users console), it is not "produced", it is merely defined. To cause the production to begin, we type the name of the script as a function (that is, surrounded by parenthesis):

```
(grunch)
```

or:

```
(frizz)
```

A N I M A T E

See also: Script, Cue, Actor, Cut, Animation-mode

The animate function is used to control one animated "sequence". But the length and complexity that constitutes a "sequence" is chosen to suit the animator's taste. A script may contain just one sequence, or it can have as many as desired.

An invocation of animate is sometimes refered to in this manual as an animate block. The execution of the contents of the contents of this block (the "body" of the animate) is similar to the execution of a loop. An animate block will execute each of the expressions in its body, and produce one frame of animation every "cycle". Usually the expressions in the body are conditional cues. These are used to start, stop,

or direct the actions of the actors. It is the actions of these actors which underlie all animated action. Generally there is at least one actor (some of whom are more like "invisible stagehands") responsible for every ongoing change in the animated sequence. Sometimes more than one actor will co-operate to accomplish some joint goal.

If we think of the eventual playback of the animated sequence now in production, one particular animate block will be active, each frame it would complete a cycle (30 times a second in playback time). During that cycle each of these things happen:

(1) Each expression in the body is executed.

(2) All of the actors are awaken and allowed to "do their thing", occasionally asking the displayer to make objects visible.

(3) The frame is cleared to the background color.

(4) All of the visible objects are displayed.

(5) An entry is made at the console for each actor being traced.

(6) The end-of-frame message is printed on the console.

(7) The frame of animation is stored (onto an analogue video disk).

(8) The animation clock is advanced one frame.

Steps 3 through 7 are optional, to control which are executed we use the animation-mode function. Steps 3 and 4 are

controlled by the "display" mode flag, step 5 by the "trace" mode, step 6 by the "eof" mode, and step 7 by the "record" mode.

As the animated sequnce continues (as the animate block continues to loop) the time for each cue comes and goes. These cues initiate various changes in the progress of the animation, such as creating and starting new actors or directing those actors already running to modify their actions.

An animate block can contain declarations of local scratch variables, used to store information about the state of the ongoing animation. The cue and give-cue functions make special use of these variables as "cues", markers of points in time, used to signal actions by the actors. Variables may be used as cue-names, or as temporary storage, but not both.

The looping of the animate block will continue until somewhere, within its body it executes a cut expression.

Examples:

```
(animate  (cue  (at  10)
                (cut)))
```

Produces 10 frames of blank background color.

```
(animate (cue (at 20) (start quack))

         (cue (at 40) (start quux))

         (cue (at 60) (cut)))
```

This produces a 60 frame animated sequence, with the actor named "quack" entering at 20 frames into the action, and actor "quux" starting at frame 40.

```
(animate (local: actor)

         (cue (at 50)
              (define actor
                      (start frob)))

         (cue (at 100)
              (speed-up actor 1.25))

         (cue (at 150)
              (cut)))
```

This animate block declares a variable named "actor", and uses it to save the idenification code of the actor it is running. This is so it can refer to that particular actor later (in this case, to speed up its action by a factor of 1.25). Note that any actors still active at the end of the animate block are automatically stopped.

C U T

See also: Animate, Script, Loop

The cut function does nothing other then terminating the current animate block. After an animate block is entered, it will continue to loop (and produce frames of animation) until it executes a cut. The term "cut" is from the filmmaking

usage. Cut is to animate as loop-exit is to loop.

Example of use:

```
(animate  ...
          (cue  (at  1000)
                (cut))
          ...
          (cue  (at  quitting-time)
                (cut)))
```

This skeleton of an animate block will continue to loop until either; it reaches the 1000th frame, or the cue "quitting-time" is given by someone.

C U E

See also: At, Script, Animate, Give-cue, Until-cue

A "cue" is an event which serves as a landmark in time. We use cues in everyday life:

"When I wave my arms, you pull on the rope."
"I'll see you in April."
"On your mark, get set, GO!"

The wave of the arm, the coming of April, and the shouted "GO!" all serve as cues, indicating the time at which something starts.

In the Actor/Scriptor Animation System a cue is indicated by the settings of special program variables within an animate block. Normally a cue is "off" or "uncued", but the give-cue function can be used to set the variable to "cued". The

variable will automatically reset (to "uncued") one frame later.

Usually the cue function is used to test variables used as cues. The execution is conditional, as in if:

(cue <when> <what>)

The <when> part is evaluated, if it is true (it asks a question whose answer is "yes"), then the <what> part is executed. Said another way: the <what> test serves to cue the <when> part.

The cue function is usually used with the 'at' function as the <when> part. The <what> part is often one of functions which direct actors: start, stop, run, or in. It can also be define, give-cue, or any expression desired.

Examples:

```
(cue  (at  0)                    Call the program "startup"
      (startup))                 on the very first frame.

(cue  (at  midpoint)             Whenever the cue "midpoint"
      (start frobber))           is given, start the actor
                                 "frobber".

(cue  (at  (plus  beat delay))
      (give-cue back-beat))
```

Everytime the cue "beat" is given, give the cue "back-beat" after a pause of "delay" frames.

Note that these two expressions are equivalent:

```
(cue  (at  5)
      (start  zapper))

(if   (at  5)
      (then  (start  zapper)))
```

A T

See also: Cue, Animate, Give-cue

The at function answers the question "Is it time yet?" At is usually used with cue or if. The at function expects one input, which can be a frame number or a cue-name.

If the argument is a cue name, then at returns:

true	if the cue has been given during the last frame.
not-true	otherwise.

If the cue is a frame number, then at returns:

true	if that is the number of the current frame.
not-true	otherwise.

Examples of use:

(cue (at 105) ...)	Do the "..." only on frame number 105.
(cue (at restart) ...)	Do the "..." whenever the cue named "restart2 is given.
(if (at (plus 5 move-start)) (then a) (else b))	If it is five frames since the frame whose number is stored in "move-start" then return the value of "a", otherwise return the value of "b".

A C T O R

See also: Script, Animate, Variables (Local and Private)

The word "actor" is actually used in three slightly different ways in this manual. In one sense, we wish to suggest an analogy with the real-world concept of a theaterical or television actor. While performing in accordance with the intent of the director (of the play or video production), a human actor is "on their own" during the actual performance. The director has planned and rehearsed the roles of the actors beforehand, but it is the actor's job to play their role without constant intervention by the director.

Within the animation system we use the term actor to describe an entity in the computer system with some of the same properties as a human actor. This usage is attributed to Carl Hewitt of MIT, and his "actor model of computation" [10, 11, 12, 13]. In this model, the program units are called "actors" (things which act). Actors are only allowed to send "messages" to one another, one actor cannot force another to take any action.

The Actor/Scriptor Animation System is a hybrid of the actor

model and the traditional "recursive function theory". To produce an animated sequence, we write a formal description of what is going to happen when. This is called the script, essentially a normal computer program. The main task of the script is to sequence and direct the actors. Once an actor has been started (gone "on stage" or "on camera") it will continue to perform its "task" or "action" without any intervention from the script. If, for example, we have an actor which causes a big red box to sail across the screen, the script need only specify when we would like the action to start (in terms of animation time or cues). If the actor knows how to do its job, no further attention needs be paid to it. The script might have specified when the actor should stop (go "off stage", or "off camera"), or the actor might decide for itself (for example, when it realizes that the big red box is off the screen, and hence no longer visible).

Just as in the theater, more than one actor may be active or "on stage" at one time. These actors may simply ignore one another, each going about its own business. Or, the actors may cooperate and work in a communal effort toward some common goal.

While the analogy with human actors is useful, it should be kept in mind that the actors in the animation system are not

directly visible. Perhaps the notion of a "puppeteer" is closer then a theatrical performer, since we see not the puppeteer but the puppet that they control. In the example above, we saw the big red box, not the actor who was moving it. We could also think of an actor as an "invisible stagehand" who moves, replaces, or repaints pieces of scenery, costumes, or props without being seen themselves.

Figure 2

A scene from the production of the "Hanna - Barbara Happy Hour" (a children's TV program seen weekly on NBC). The marionette/puppets (known as "Honey" and "Sis", the stars of the show) are being controlled by the puppeteers behind them. Only the puppets will remain after the video mixer replaces everything blue in the picture, including the blue background and the puppeteers in their blue tights. (this tracing is from a photograph which appeared in the April 15, 1978 issue of TV

Guide magazine, pp. 10-11)

As mentioned above we use the term "actor" has three slightly differnt usages; the theatrical analogy, the notion of an independent computational process which communicates by message sending, and the actor function. This function is used in the scripting language to specify and construct the actors that the script will use. The actor data-type itself (the thing returned by the actor expression) is an element of the animation system which can be handled in the usual ways. For example, this defines the variable "John" to be the specified actor:

```
(define  John  (actor  ... ))
```

where the "..." is replaced by the actual specification of the actor. The actor expression returns the actual actor data-type, and so may be used in a define or directly as an input to an other function which expects an actor as an input. Either of these two cue expressions could be used to start a specified actor at frame number 10:

```
(cue  (at  10)
      (define  Jane
               (actor  ... ))
      (start  Jane))

(cue  (at  10)
```

(start (actor ...)))

An actor is made up of two parts, a memory and a role. Its memory is a collection of data, think of it as a set of memos, each with a topic name at the top and an expression written below. At the programming level this means that an actor has a set of bound variables. The "role" of an actor is represented by a program. An actor acts by running this program once betweeen each frame, using its memory to keep track of its progress.

To specify an actor we must indicate:

(1) the names of all variables to be used by the actor
(and optionally, initial values for them)

(2) the expressions which make up the body, the "role"

The general form of an actor expression is:

(actor <declarations-of-variables> <body>)

Examples of use:

```
(actor  (local:  (age 0))

        (inc  age)
        (print  age)))
```

This actor has one local variable called "age" which it increments, and then prints each frame.

```
(define  blinker
         (actor  (local:  (on-time  5)
                          (off-time  10)
                          (object    nil))
```

```
(private:  (time    0))

(inc  time)

(if  (less  time on-time)
     (then  (see  object)))

(if  (greater time
                (plus  on-time off-time))
     (then  (define  time 0))))))
```

The variable "blinker" has been defined to be the specified actor. It causes a geometrical object called "object" to alternate between being visible and invisible. The times it is in each of these states are controlled by the variables "on-time" and "off-time". The actor keeps its own private "time" so that it will not forget what time it is.

S T A R T

See also: Actor, In, Stop, Run

The start operator takes an actor object and "activates" it. This means that an "instance" (or copy) of it is added to the active actor list. It is these active actors which are awaken each frame, the ones who are "on stage".

Start returns a functional value, an integer number. This number is known as active actor "id". It uniquely identifies an individual active actor. Because we may wish to have several identical actors active at one time, we use these numbers to refer to a particular instance of an actor. Rather

then deal directly with the id numbers themselves, we usually define a variable to the result of the start:

```
(define  Hamlet
         (start  ... ))
```

From that point on, when we refer to the variable "Hamlet", we are refering to this particular instance of the actor.

The start operation allows additional input expressions which are not immediately executed, but are sent in to the newly created instance of the actor for initialization (see in).

Usage:

```
(start  <actor>  <in-exprl>  <in-expr2>  ... )
```

The <in exprs> must be of the type allowed for use with in; basically define expressions for variables declared by the actor.

Examples of use:

```
(start  (actor  ... ))
```
Create an actor and start it.

```
(start  blinker
        (define  object cube))
```
Starts an instance of the actor "blinker", defining its blinkee "object" to be a "cube".

```
(define  floink-1
         (start  floink
                 (define  size 1)))

(define  floink-2
         (start  floink
                 (define  size 2)))
```

This starts two instances of an actor ("floink"), saving the

individual active actor ids (as "floink-1" and "floink-2"). The two instances have each been given different initial sizes.

S T O P

See also: Actor, Start, Run

The stop operator will deactivate an active actor. The dear departed instance is lost (ultimately to be recycled and born again). Stop allows one input which should be an active actor id, the one to stop. If this input is omitted, the currently active actor stops itself. Hence an actor may be stopped by anyone who knows their id, or they can stop themselves.

Usage:

(stop)

(stop <active-actor-id>)

Examples of use:

(stop Hamlet) — The actor instance "Hamlet" is stopped.

```
(actor (local: (time 0))
       (if (less time 100)
           (then (frobulate time)
                 (inc time))
           (else (stop))))
```

This is an actor expression containing a use of the no

input form of stop. The actor calls the function "frobulate" every frame for the first 100 frames after it is started, then stops itself.

R U N

See also: Actor, Start, Stop, For, Until-cue, Give-cue

The run function is a combination of start and stop. It takes advantage of the fact that we often use these other operations in certain combinations, hence we develop a shorthand. Run is used either;

(1) to run an actor for a certain period of time (some number of frames for example).

(2) to run an actor until a particular named cue is given.

Run expects at least two inputs; the actor to run, and a "termination actor". The termination actor is usually constructed by either the for or until-cue functions. Additional input expressions may be given which will not be immediatly executed, but are sent to the newly created actor by the in operator.

Usage format:

(run <actor> <termination-actor>)

(run <actor>
<termination-actor>
<in-exprl>

```
        <in-expr2>
         ...)
```

The functional value returned by run is the same active actor id that is returned by start when the main actor is started. Hence we can define a variable to be the result of the run function, to allow us to refer to that instance later in the script:

```
(define  Ophilia
         (run  ... ))
```

By the use of for and until-cue we start actors with predetermined fates. The "termination actor" serves as a grim reaper who will do away with the main actor at the anoited moment. The for function expects one input, a number of frames. It creates an actor which will stop another actor in that number of frames. The until-cue function takes a cue-name and makes an actor which will do in the other when the cue of that name is given. The cue-name is a variable and must have been declared in a surrounding expression (the script, animate, or defprog containing the give-cue expression). The cue name supplied to give-cue will not be evaluated, the name itself is used.

Usage format:

```
(run  <actor>  (for  <frame-count> )  ... )
(run  <actor>  (until-cue  <cue-name> )  ... )
```

Examples of use:

```
(run  (dwonk  dcount)
      (for  300))
```

The actor formed by the "dwonk" function is run for the next 300 frames.

```
(run  mover
      (for  1000)

      (define  speed 30))
```

The actor "mover" is run for 1000 frames, its initial "speed" is set to 30

```
(define  b7
         (run  blunk
               (until-cue  stop-blunks)))
```

An instance of the actor "blunk" is started. It will run until someone else gives the cue named "stop-blunks". This particular instance of "blunk" will be known as "b7".

I N

See also: Actor, Start, Variables, Private, Define

The in function is used to communicate with actors after they have been started. When the script wants to send directions to an actor, or wants to ask an actor a question, it uses the in function. Additionally, active actors may communicate with each other by using the in function.

An actor is composed of two parts, its "role" and its "memory". The communication facility provided by the in function deals solely with the memory part (local variables of

the actor). We may examine or modify the non-private variables of an actor. The name "in" is used because the variables are examined or modified "within" or "inside" the world of the specified actor.

In expects two or more inputs, the first specifies the actor we wish to communicate with, this value should be an "active actor identification number" as returned by start or run. The rest of the input expressions are evaluated AFTER we jump into the world (binding context) of the actor.

General usage format:

(in <active-actor-id> <expr1> <expr2> ...)

The <expr>s must be one of the two allowable types. They may either examine or define a variable, hence the expression may be either:

(1) the name of the variable, or

(2) a define expression for the variable.

In either case, the variable must be both:

(1) declared by the specified actor, and

(2) not declared to be private.

Any other type of expression will cause a run time error condition.

If the expression is a define for variable "foo", the second input of the define (the value part) may not contain any references to undeclared variables (as usual) with the exception of "foo". For example, the last in expression in the following example is valid even though "foo" is not declared in the program "klork" which contains the in:

```
(defprog  klork
          (inputs:  actor-id  new-bar)

          (in  actor-id
               (define  bar  new-bar))

          (in  actor-id
               (define  foo  (times 2 foo))))
```

The proceeding program will cause a run time error unless the actor indicated by "actor-id" has variables (memory cells) with the names "foo" and "bar".

To read information from an actor we supply (through the in expression) the name of the variable containing the desired information. For example, to ask for the "speed" of an active actor whose id number is held in the variable "actor2" we would write:

```
(in  actor2  speed)
```

or

```
(define actors-speed
        (in  actor2  speed))
```

In returns as its value the result of evaluating its LAST

input expression inside the context of the specified actor. In the previous example, the last expression is "speed"; so the value of "speed" in "actor2" is returned.

To send information to an actor we must supply both the information itself and the "name" of the message. It is as though the actor had not just one mailbox, but one for each type of message it might possibly recieve (one for bills, one for paychecks, one for love letters, one for magazines, ...). So if we wish to send a message of the form "Change your 'speed' to 100", for example, we would write:

```
(in  actor2
     (define  speed  100))
```

Using the mailbox analogy, this would correspond to putting the value 100 into the actor's "speed" mail box.

We can either use the in function directly from the script or an actor, or we may use it as a primative operation with which to construct more complex operations. One powerful result of the structure of actors is that we can make programs which modify a particular named parameter of ANY actor, regardless of the particular structure of that actor. Here is a program called "speed-up" which will change the "speed" of any specified actor by a given scale factor.

Example of use:

```
(defprog  speed-up
          (inputs:  actor  factor)

          (in  actor
               (define  speed
                        (times  speed  factor))))
```

G I V E - C U E

See also: Cue, Until-cue, Script, Variable

Events in the course of an animation script are marked by "cues", points in time at which significant happenings occur. The signaling of these "cues" is done by the give-cue operation. It expects one input which is the name of the cue to signal (the name is not evaluated).

The effect of signaling a cue is as though we had shot off a skyrocket. Everyone who is interested can see that it has happened. One give-cue may start any number of actions, cue expressions in the main script may be triggered, active actors may stop or change what they are doing because of the cue being signaled.

A cue is a short lived phenomenon, just as the skyrocket, the cue soon burns out and is no longer visible. A cue lasts for

exactly one frame of animation. Because of the way actors and scripts are defined, this means that everybody who might care gets at least one chance to see the signaled cue.

Usage format:

(give-cue <cue-name>)

Examples of use:

(give-cue Now)	The cue named "Now" is signaled.
(cue (at master-cue) (give-cue subcue1) (give-cue subcue2))	In this cue expression from an animate block, the cue named "master-cue" is used to signal two other cues named "subcue1" and "subcue2".

S E E

See also: Camera, Geometrical objects, Subworld, Script

The see operation is used to indicate which of all of the existing geometrical objects are to be "seen" by the camera of the animation system. See expects one input, the object to be seen:

(see <geometrical-object>)

The exact copy given to see is placed on the visible object list. At the end of the frame all of the visible objects are

made intc a subworld with the camera as the point of view, it is that subworld which is actually displayed. The visible object list is emptied before each new frame.

Examples of use:

(see box)	The object "box" is made visible for this frame.
(see (rotate 1/3 Joe))	A rotated copy of "Joe" is made visible this frame.

G E O M E T R I C A L O B J E C T S

The objects which make up the Actor/Scriptor Animation System can be divided into two main classes:

(1) Programming constructs (control structures)

(2) Geometrical objects (data structures)

All of the programming features exist ultimately to create, modify, and display the geometric models used to represent the visible objects in an animated sequence. Both conventional programming features and special "animated" programming constructs are provided to allow the most convenient way of manipulating the geometrical objects. This section deals with the nature of the geometrical objects, and how they are created by programs.

The geometrical objects are so called because their common property is a description based upon coordinate geometry. The objects contain information about their type (e.g. vector, solid ...), information about their specific description (positions, orientations, colors), and information about their structure (such as specification of the members of a group).

V E C T O R

See also: Numbers, Coordinates

A vector is a group of three numbers, they are interpreted to represent distances along each of three coordinate axes (called "x", "y", and "z", in that order). A vector may specify either a relative distance from a base location, or an absolute position (by being based on the "origin", the position where the three coordinate axes intersect). To create a vector with a given set of coordinates we would use the vector function:

(vector {x} {y} {z})

The three inputs to the vector function are optional, missing coordinates will be set to zero. Hence these pairs of expressions are equivelent:

(vector)	=>	(vector 0.0 0.0 0.0)
(vector 1)	=>	(vector 1.0 0.0 0.0)
(vector 1 2)	=>	(vector 1.0 2.0 0.0)
(vector 1 2 3)	=>	(vector 1.0 2.0 3.0)

The coordinate expressions used in a call to the vector function can be any expression as long as it returns a number. Any type of number (fixnums or flonums) may be used, although the vector function always returns a vector with three flonum coordinates.

Examples of use:

(vector (plus a b))	Forms the vector: (vector a+b 0.0 0.0)
(move (vector a b c) (vector q r s))	Forms a vector which is the vector sum of the two given.
(define position (move position velocity))	Redefines "position" by adding an incremental "velocity" into it.

P O L Y G O N

See also: Vector, Color, Cut-hole

A polygon is very much like a thin piece of cut-out sheet metal, hanging suspended in three space. A polygon is flat and has straight sides, it can be painted with a color (or 2 different colors for the front and back), and can have holes or "windows" cut through the otherwise opaque surface. One of the possible colors for a polygon's side is the "no color" color nil, which corresponds to "invisible". If the face of a polygon which is toward the camera is colored nil it will not be seen. This is the mechanism used to remove "back faces".

To create a polygon object we use the polygon function. It expects at least four input expressions. The first input is a color or a group of two colors. The rest of the inputs are vectors specifying the positions of the vertices of the

polygon. The order of the listing of the vertex vectors is the order we would encounter the vertices if we started at any vertex and walked around the polygon in a clockwise direction:

(polygon <color> <vector1> <vector2> <vector3> ...)

At least three vectors are expected, because the polygon with the smallest number of vertices is a triangle, which has three. A full <color> specification includes a front, back and edge color, in that order, omitted colors will be set to nil:

<color> becomes: (group <color> nil)

(group <c1> <c2>) becomes: (group <c1> <c2>)

Hence the default back face color is "invisible".

The polygons produced by the polygon function are "simple polygons", they have just one boundary, and so no holes. To form polygons with holes we make one polygon the shape of the outside, and one polygon the shape of the hole, the holey polygon is formed from these two simple polygons with the cut-hole operator.

Example of use:

```
(polygon  red                  Forms a isosceles right
          (vector  1 1)        triangle in the x-y plane,
          (vector  1 4)        which is red on its front,
          (vector  4 1))       and invisible from behind:
```

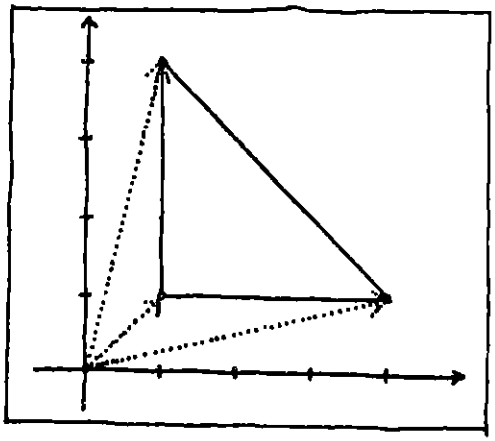


Figure 3

G R O U P

The group is the facility that is used to "glue" simpler objects together into more complex objects. Although the objects need not actually "touch" to become rigidly attached to each other. Group takes any number of input expressions, and returns the group of the values of all the input expressions.

(group <object1> <object2> ...)

Once objects are grouped together, they will be treated as a single object by all of the geometrical operators. For example, if we had already defined two objects "ding" and "dong":

```
(define  both                    This defines "both" as a group
         (group  ding dong))     containing "ding" and "dong"
                                 glued together into one object.

(define  rot-both                Defines "rot-both" to be a
```

(rotate 0.3 both)) rotated copy of "both".

In this example, the rotate operator handles the group of "ding" and "dong" as though they were both glued onto a tabletop, and the entire table was rotated. Because of the "copy, then modify" rule that the operators obey, rotating the group has no effect on the original "ding" or "dong" or "both".

S O L I D

See also: Color, Vector, Polygon, Local variables

Using groups of polygons we can construct objects which would appear as solids in three space. For example, six squares could be positioned and grouped to form a object which would appear as a cube. However if the aim is to construct a three dimensional bounded region of space, the most direct way is to use the solid function. In addition to specifying the vertex and face information, the vertices of a solid are shared among the faces which contain them, thus avoiding replication. Also the description of a solid allows geometric operators to act on the object as a solid chunk, when solids are represented as grouped polygons this is not possible.

To create a solid we use the function of the same name, the input expressions are evaluated in a nonstandard manner. The general format of a solid expression is:

(solid <color> <vertex-declarations> <poly1> <poly2> ...)

The <color> is evaluated to yield a standard color object. The <vertex declarations> are contained in a vertices expression.

The face <poly>s have the same form as a polygon expression. The format of a vertices expression is exactly the same as the format of a local variables expression (except for the substitution of the word "vertices" for the word "local"):

```
(vertices:  (  <vertex-name-1>  <initial-value-expression-1>  )
            (  <vertex-name-2>  <initial-value-expression-2>  )
            (  <vertex-name-3>  <initial-value-expression-3>  )
            ... )
```

The vertices expressions serves to assign a name to each vertex of the solid, and to specify the vector position of the vertex. The <initial value expression> is the only part of the vertices expressions which is evaluated.

There must be one polygon expression for each face of the solid. The color and the vertices (listed by name, in clockwise order) which make up that face are specified. The color expression is the only part that will be evaluated at definition time. If the color expression for a face is "*", then the solid's default color will be used.

As an example of the use of the solid function, there follows a program which constructs a tetrahedron (one of the five regular polyhedra, a pyramid with a triangular base) with three red faces and one yellow face. The yellow side lies in the x-y plane, the red sides "point" in the positive "z" direction. The program expects one input, the size of the

tetrahedron to construct:

```
(defprog  yellow-red-tet

          (input:  size)
          (global:  yellow red)
          (local:  (vec    (vector  0 size))

                   (point (vector  0
                                   0
                                   (times  size
                                           (sqrt 2.0)))))

          (solid  red
                  (vertices:  (a  vec)
                              (b  (rotate  1/3 vec))
                              (c  (rotate  2/3 vec))
                              (d  point))

                  (polygon  yellow  a  b  c)
                  (polygon     *    a  d  b)
                  (polygon     *    a  c  d)
                  (polygon     *    c  b  d)))
```

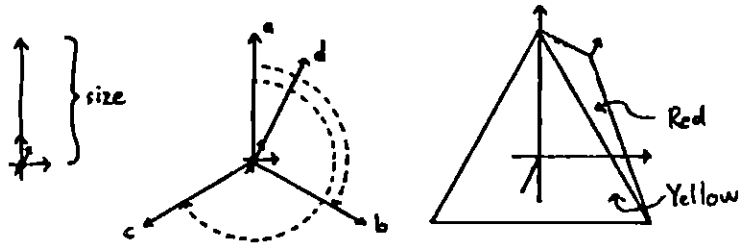


Figure 4

Construction of a yellow and red tetrahedron.

P O V (Point of View)

See also: Coordinates, Self-relative, Camera, Subworld

The point of view geometrical object is used to represent the relationship between an object's local coordinate system and the external, global coordinate system. A pov can be used to describe the appearent changes in an object's position, orientation and size as "seen" from differing points of view. For example, the camera of the animation system is a pov object. The subworld data-type consists of a pov and another object which is to be situated AT the pov.

The parameters which make up a point of view description include:

(1) The position of the local origin. (As an absolute position vector.)

(2) The orientation of the coordinate system. (As direction vectors along the local coordinate axes.)

(3) Three scaling factors. (One for each local coordinate axis.)

This information is held as four vector objects. One is the base position vector, the other three are vectors parallel to the local coordinate axes. The magnitudes of these three vectors are the scaling information. It is worth noting in passage that the 12 parameters of a pov (4 vectors times 3 coordinates each) correspond to the 12 parameters of a 3X4 transformation matrix which is often used as a formalism of this notion. (A 4X4 matrix is used in homogeneous

coordinates.)

The pov function is used to create pov objects. It expects zero or four inputs. If no inputs are given, the "home" pov is returned, this is the orientation of an object at the global origin, pointing down the "z" axis. Otherwise the four inputs are the vectors which describe the pov:

(pov)

(pov <position> <x-axis> <y-axis> <z-axis>)

S U B W O R L D

See also: Pov, Self-relative operators

The subworld data-type allows us to think of geometrical objects as having their own view of the world, by allowing an object to have its own personal coordinate system. The "origin" in such a "centered on one's self" coordinate system is usually located at the object's center. The positive coordinate axes ("x", "y", and "z") correspond directly to the object's directions of "right", "up", and "forward". The self-relative geometrical operators are used to manipulate an object with respect to its own coordinate system.

A subworld has two major parts:

(1) A description of the relation of the internal coordinate system to the external coordinate system (a point of view ("pov") object).

(2) Another geometrical object (frequently a group) which is to be situated at the point of view. Hence the object is put int the same "frame of reference" as the POV.

The usage of such a data-type is perhaps best thought of in terms of the animation system's camera, the imaginary TV camera which is used to take pictures of the imaginary geometrical objects. If the script calls for the camera to "tilt up", we could accomplish this in two different ways. We could "tilt up" the camera, or we could "tilt down" everything the camera was looking at. Surely it would be easier to tilt one camera then to tilt the rest of the "world" (even if both the camera and the "world" are just figments of the computer's imagination). At the end of each frame, all of the visible objects are seen through the currently defined camera. This means that that we may change our view without operating on all of the visible objects.

However in the case of a subworld, its contents are PLACED AT the pov rather then SEEN FROM it. The pov serves as a platform

on which to situate the contents. The subworld's contents are modified at display time to conform to the position, orientation, and scale of the pov. Usually the subworld object is used to contain a complicated object so that we may modify just the point of view part and leave the rest alone. If we had constructed a geometrical model of a airplane, we would have described its position in whatever terms were most convenient (although usually we want to have the origin of the local coordinate system to be coincident with the "center" of the object). However to make the plane fly, we need to reposition and reorient it. Again we could either modify each part of the description of the airplane (move the wing, move the tail, move the landing gears ...) or we could have constructed a subworld object whose content is the model of the airplane. Then to fly the airplane we need only modify the subworld itself, the original model remains unmodified.

An object's own "self-relative" coordinate system may be, but need not be, the same as the "global" or external coordinate system. When they are the same, the object is said to be "home". We usually create a subworld object in its "home" position and orientation. The subworld function expects both the pov description, and the object which will become the contents of the subworld:

(subworld <pov> <object>)

If the <pov> expression is "(pov)", then the subworld object is created in its "home" position, otherwise the supplied coordinate system is used.

Additionally, these parameters are automatically determined at the time the subworld is constructed:

(1) A "maximum radius", the distance from the local origin to the most remote part of the contents of the subworld. (Used in screen and detail clipping.)

(2) A "typical" color (which can be nil). (Used to display the entire object (as a "dot") when it would appear too small on the display screen to show any detail.)

Example of use:

These two expressions construct the subworld containing the airplane as discussed above (omitting the details of the airplane itself):

```
(define  airplane-model
         (group  ... ))

(define  airplane
         (subworld  (pov)
                    airplane-model))
```

G E O M E T R I C O P E R A T O R S

See also: Geometrical objects, Coordinate systems

The Actor/Scriptor Animation System provides two way to manipulate geometric objects. We can express motion either from the point of view of the object involved (with self-relative operators), or in terms of an external coordinate system or "frame of reference" (using global operators).

Self-relative operators express a new position or orientation for an object relative to its current position and orientation. This is somewhat like spaceflight, the self-relative operators are similar to the controls of a spaceship. The direction in which the spaceship is pointing establishes the "forward" direction (also refered to as the "nose vector"). Using the ship's steering jets the pilot can alter its heading. The ship's nose can be turned right or left (just as in an automobile), tilted up or down (like a see-saw), or the ship can be "rolled" about the nose vector in either the clockwise or counterclockwise direction. The pilot can fire the main thruster to move in the forward direction (along the "nose vector"). We will also assume our imaginary

spaceship has a thruster at its nose so that it can be flown "backwards".

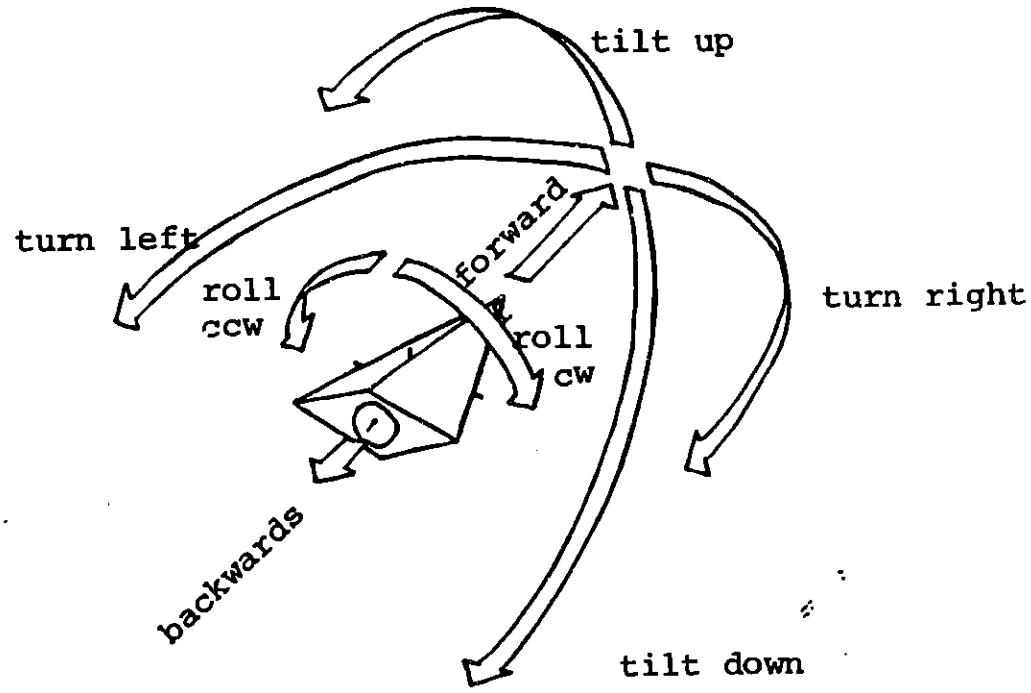


Figure 5

Self-relative operators as spaceship maneuvers.

This type of operator is called <u>self-relative</u> because measurements of changes (such as how far to go forward, or how much to turn right) are made from the objects own "frame of reference" or coordinate system. It is as though we put ourselves in the place of the object we wish to change, and described how we would move from that position to the desired position.

For those learning geometry for the first time this "personal" frame of reference is easier to understand than a fixed, external system. Seymore Papert (and the LOGO Group at MIT) have successfully used this approach to teach two dimensional geometry to young school children. [26, 27, 28] Using a programming language called LOGO, students as young as 6 years have learned how to program a computer-controlled graphical device called a "turtle" to draw very complicated and interesting designs. In "turtle geometry" as it is called, the turtle can walk forwards and backwards, and turn right and left. Upon command the turtle can also lower a marking pen attached to its belly, so that its path is traced on the large sheets of paper spread on the floor. [1, 9, 19] The self-relative operators of the Actor/Scriptor Animation System are a three dimensional generalization of the ideas of "turtle geometry".

The global operators provide another way of specifing geometrical changes. The major difference between the two types is that in global operations, the "center of the universe" is defined by the coordinate system in use. When we use the self-relative operator right we expect the center of the rotation to be wherever the object is. When using the global operator rotate the center of the rotation is the center of the coordinate system. This location in three

dimensional space, also known as the "origin", is represented as the position vector: (vector 0 0 0). Any other position is measured in terms of its distance from the origin, along three mutually perpendicular directions known as "coordinate axes". The axes are named "x", "y", and "z" (similar to "right", "up", and "forward"), which intersect at the origin.

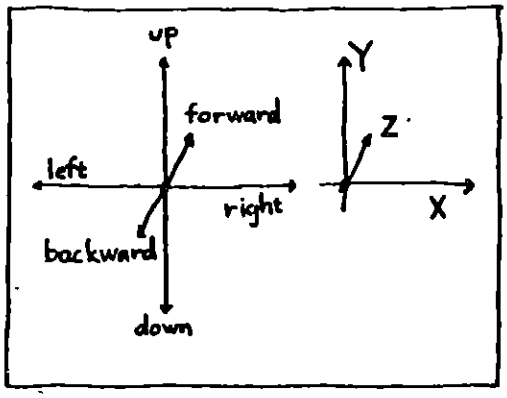


Figure 6

The positive coordinates axes.

Just as self-relative operators are similar to flight, the global operators could be compared to afixing the objects involved to invisible machanical linkages. The rotate operator has an effect like placing an object on a phonograph turntable, the move operator is akin to placing the object on a conveyor belt, the scale operator does what a pantograph does. The set of global operators are similiar to the machine tool operations familiar to machinists.

G R A S P

See also: Variables

The grasp function causes no geometrical changes itself, but is used to control other operations. The individual operators can work in either of two ways. If passed an object to operate on, they will produce an altered copy of the object, and return the copy as their functional value. Otherwise we can use the grasp operator to specify the "current object" and from then until the next use of grasp, operations will operate by default on the "current object". One way to think about this is that the operators all use an imaginary "hand" to change objects. We can tell the operator where to put the hand (by specifying an object in the call). Or, we can indicate that the operation should use the object already in the "hand", by omitting the object specification.

To use the grasp function we pass it one input, the name of a variable which is defined to be a geometrical object. Grasp does not evaluate the variable, it remembers the name itself. From then on operators which are not passed an object, operate on the object attached to the "grasped" variable. Hence these two chuncks of code do equivalent things:

```
(grasp thing)          |    (define thing
                       |            (right .02 thing))
```

```
(right .02)
(forward 30)
```

```
(define thing
        (forward 30 thing))
```

Warning: since the grasp operator uses NAMES rather then VALUES there is an opportunity for the "funarg" monster to raise its ugly head. If a program, which does not use the grasp operator, references the "grasped" object (e.g.: (forward 10)) there is a possiblity of name conflicts between the name of the grasped object and the variables of that program.

F O R W A R D , B A C K W A R D

See also: Number, Grasp

The forward operator moves an object along its own "forward" direction. Forward expects one or two inputs, the first is the distance to move along the "forward" direction. The second input, if given, is the object on which to operate. If the second input is ommited, the currently grasped object is used.

The backward operator differs from forward only in that the motion is in the opposite direction.

R I G H T , L E F T

See also: Angle, Grasp

These operators cause an object to rotate about its own vertical axis, to the right or left by an angle of rotation specified as the first input. If an object is specified as a second input, it is operated upon, otherwise the object currently grasped is used.

U P , D O W N

See also: Angle, Grasp

These operators cause an object to rotate about its own horizontal ("through the ears") axis, causing its "nose vector" to go up or down by an angle of rotation specified as the first input. If an object is specified as a second input, it is operated upon, otherwise the object currently grasped is used.

C W , C C W

See also: Angle, Grasp

These operators cause an object to rotate about its own

horizontal ("nose vector") axis, in the clockwise (cw) or counterclockwise (ccw) direction by an angle of rotation specified as the first input. If an object is specified as a second input, it is operated upon, otherwise the object currently grasped is used.

Examples of usage of self-relative operators:

```
(grasp  camera)
(forward  20)
(down  1/4)
```

These three steps cause the object currently defined as "camera" to be moved forward 20 units, then to tilt straight down.

```
(defprog  chase-tail
          (input:  object)

          (left  0.01
                 (forward  2 object)))
```

This program, if used to redefine an object each frame, would cause it to chase its tail around in a circle, like dogs sometimes do.

```
(defprog  barrel-roll
          (inputs:  speed object)

          (forward  speed
                    (cw  1/100
                         (up  1/100 object))))
```

This program will make the incremental change in an object which would produce a "barrel-roll" flight path if used to redefine an object each frame. Both the "airspeed" and the object to be modified are specified.

H O M E

See also: Pov, Coordinates, Grasp

The home function will cause an object to be restored to its "home" position and orientation. This has the effect of undoing any self-relative operations which the object has undergone. If given an object as an input, home will reset that object. Otherwise the currently grasped object is reset.

Example of use:

(grasp schmoo)	Grasp the object named "schmoo"
(cw 1/3) (up 1/4)	Modify it ...
(define schmoo2 schmoo)	Save the modified version under another name.
(home)	Reset the state of "schmoo".

G R O W , S H R I N K

See also: Number, Scale

Grow and shrink cause an object to change size (about its own center) by a specified amount. Both of these operators expect one or two inputs, the scale change factor (a number), and optionally, the object to change. If the second input is omitted, the currently grasped object is used:

(grow <growth-factor> <object>)

(grow <growth-factor>)

(shrink <shrinkage-factor> <object>)

(shrink <shrinkage-factor>)

Example of use:

```
(define big-flower
        (grow 10 flower))
```

Defines "big-flower" to be the result of growing "flower" by a factor of 10 in size.

Z O O M - I N , Z O O M - O U T

See also: Number, Scale, Camera

Zoom-in and zoom-out both basically do the same thing, they are used to change the "magnification" ("zoom") of a point of view ("camera") object by a specified amount. The camera of the animation system is a variable whose value is a point of view object. Both of these operators expect one or two inputs, the zoom change factor (a number), and optionally, the pov to change. If the second input is omitted, the currently grasped pov is used:

(zoom-in <zoom-factor> <object>)

(zoom-in <zoom-factor>)

(zoom-out <zoom-factor> <object>)

(zoom-out <zoom-factor>)

Examples of use:

```
(grasp  camera)
(forward  36)
(right  1/3)
(zoom-in  1.5)
```

The "camera" is made the current object. It is moved forward, turned to the right, and then zoomed in.

```
(define  cam2
         (zoom-out  2 cam1))
```

A new camera ("cam2") is defined to be like "cam1" but with twice the angle of view.

R E C O L O R

See also: Color, Polygon, Solid

The recolor operator will change the color of an object without changing its shape, orientation, or position. Recolor expects one or two inputs, the first is the new color for the object. The second input is the object to be recolored, if it is omitted the currently grasped object is used:

(recolor <color>)

(recolor <color> <object>)

The object returned by recolor is a copy of the original, except that any colors occuring in the object (as part of a polygon, solid, or subworld) will have been replaced by the new color. The <color> used may be a simple color or a group of two colors.

Examples of use:

(recolor red town)	Paints the town red.
(define blue-ball (recolor blue ball))	Defines "blue-ball" as a blue copy of "ball".

S C A L E

See also: Numbers, Grasp

The scale operator is used to change the size of an object, without changing its shape or color. We specify a number (a "scale factor") an an object, and scale returns an enlarges (or shrunken) copy of the object. A scale factor of "2" means to make the copy twice as large, a scale factor of one-half (0.5) means that the copy should be just half the original size. The difference between scale and the self-relative grow and shrink is that the scaling is done about the global, rather then local If the second input is omitted, the currently grasped object will be used.

General usage:

(scale <scale-factor>)

or:

(scale <scale-factor> <object>)

Examples of use:

(scale 2 house)	Makes a double sized copy of "house".
(define tower (scale 10 tower))	Redefines the variable "tower" to be ten times as large as it was previously.
(define tiny-ball (scale .001 ball))	"tiny-ball" is defined to be 1/1000 the size of "ball".

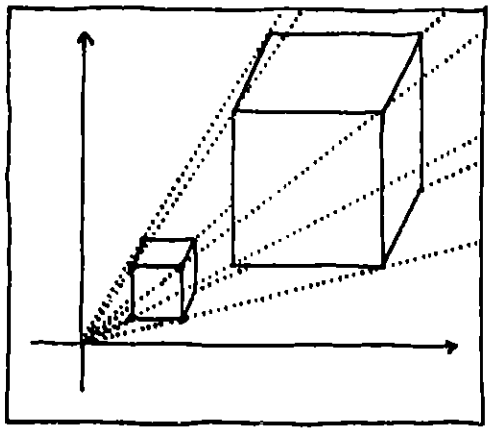


Figure 7

Note that when an object is scaled,
and it is not centered about the origin (vector 0 0 0),
it will appear to move away from the origin during a enlargment
(or towards the origin for a reduction in scale).

M O V E

See also: Vector, Coordinates, Grasp

The move operator is used to change the position of an object without changing its orientation or other properties. When we pull the drawer out from a desk or a dresser, we are performing a "move" operation on the drawer. The shape and color of the drawer are the same, and it has not been rotated. In English the word "move" has a more general meaning, but here in this manual we will restrict it to mean this particular type of "sliding without turning".

position vector". This is just one way of interpreting a vector. The three coordinates of a vector specify "how far" in each of three directions, "left", "up", and "forward" (also known as "x", "y", and "z"). When we consider a vector as "relative" we measure each of the distances from "where we are now", parallel to the axes of the current coordinate system. Move expects one or two inputs, a relative vector, and an object to be moved. It returns a moved copy of the object. This means that each part of the copied object will have been moved along the vector from its previous position. If the object specification is omitted, the currently grasped object is operated upon.

General usage:

(move <vector>)

(move <vector> <object>)

Examples of use:

```
(move  that-a-way  bad-guys)        Makes a copy of "bad-guys"
                                    which is moved along the vector
                                    called "that-a-way" from its
                                    original position.

(define  friggle                    Redefine "friggle" to be one
        (move (vector 0 0 1)        "unit" further in the z or
              friggle))             forward direction, then it was.


(move  elevator-shaft-location
       (move  elevator-altitude elevator-car))
```

or equivalently:

(move (move elevator-shaft-

location elevator-altitude)

elevator-car)

These expressions produce a copy of "elevator-car" with compound motion based on the sum of two vectors, the location of the elevator shaft, and how high the elevator currently is within the shaft.

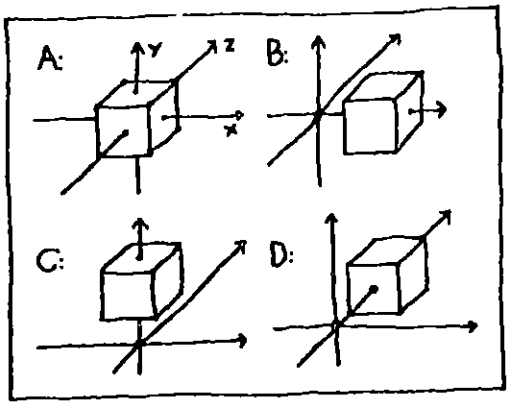


Figure 8

A cube shown before, and after several moves, along each positive coordinate axis.

R O T A T E

See also: Angle

The rotate operater is used to turn an object "around" an axis. For example, the motion of a wheel around its axle is a rotation. Rotate is also used to form anything with rotational symmetry, such as the placement of petals of a flower about its stem. Rotation does not effect shape or color, it will

however change the position of an object which is away from the axis of rotation (rotating the wheel changes the position of the valve stem, for example).

Rotate can take one, two, or three inputs. We must always specify the angle of rotation (how much to rotate) as the first input. The second input is the object to rotate, if it is omitted, the currently grasped object is used. The third input is the axis about which to rotate the object, if it is omitted, the z-axis is used.

General usage:

(rotate <angle>)

(rotate <angle> <object>)

(rotate <angle> <object> <axis>)

The z-axis is the line perpendicular to the center of the display screen (as seen by the initial camera), thus not specifying a third input to rotate indicates that a rotation of the x-y (screen) plane is desired. Rotation about the z-axis is sometimes called a "two-dimensional" rotation (since the z coordinates remains unchanged), and is the most commonly used of any rotational axis. Axes may be specified as one of the standard, named axes. These are x-axis, y-axis, and z-axis. the convention for positive angles is that as we look (in the positive direction) along the axis of rotation, a positive rotation is clockwise.

The angle by which the object should be rotated is given as a number. Rotations are measured in "revolutions", one revolution is one full turn.

Examples of use:

(rotate 0.1 wheel)	Form a copy of "wheel" rotated one tenth of a revolution.
(rotate 1/2 wheel axle)	Rotate "wheel" half way around the axis "axle".
(define krakle (rotate .001 krakle))	Redefines the object "krakle" to be slightly rotated about the z-axis.

```
(rotate (times angle step)
        (move reposition frob)
        current-axis)
```

An example of an expression which calculates all of its inputs "on the fly" thus allowing them to change every time the expression is re-evaluated.

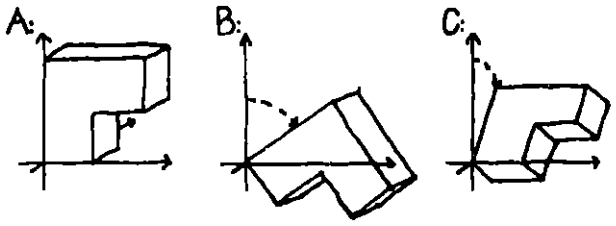


Figure 9

(a) The original "glork".
(b) result of: (rotate 1/6 glork).
(c) result of: (rotate 0.2 glork x-axis).

S T R E T C H

See also: Scale, Coordinates, Grasp

The stretch operator is a more general case of the scale operator. In scale we specify one scale-factor and this is used to multiply all coordinates. In stretch we are allowed to specify different scaling-factors for each coordinate axis ("x", "y", and "z"). If, for example, we wish to make an object "tall and skinny", we would like to increase its height, while decreasing its width and thickness. This would correspond to a "scaling factor vector" similar to:

(vector 0.1 5.0 0.1)

The "scaling factor vector" indicates that "x" and "z" coordinates should be reduced by a factor of 10 (multiplied by 0.1), while the "y" coordinates should be increased by a factor of 5. When we wish one of the coordinate directions to remain uneffected by a stretch operation, the corresponding scale factor should be 1.0.

The stretch operator expects one or two inputs, a scaling factor vector and optionally, the object to stretch. If the object is omitted, the currently grasped object is stretched.

General usage:

(stretch <scale-factor-vector>)

(stretch <scale-factor-vector> <object>)

Examples of use:

(stretch (vector 100 1 1) quux)

Produces a copy of "quux" which is 100 times wider then the original.

```
(define wire
        (stretch very-long cylinder))
```

Defines a "wire" to be a very long "cylinder".

```
(define short-people
        (stretch (vector 1 0.75 1)
                 average-people))
```

"Short-people" are just "average-people" with less "y" scaling.

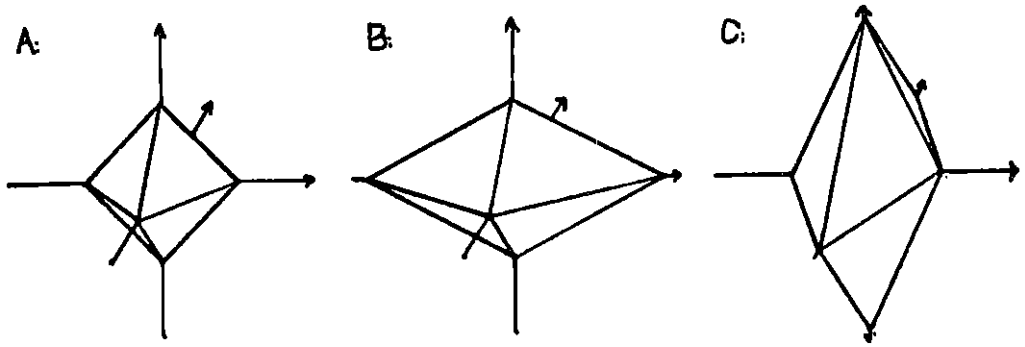


Figure 10

(a) The original octahedron.

(b) The result of: (stretch (vector 2 1 1) octahedron)

(c) The result of: (stretch (vector 1 2 2) octahedron)

M I R R O R

Your left and right hands are very similar in overall structure, but in a sense they are exact opposites. The mirror operator changes the "handedness" of an object, just like being reflected in a looking glass. Mirror needs no parameters, it expects one input (the object to reflect) but if it is omitted, the currently grasped object is used:

(mirror)

(mirror <object>)

The object will be reflected about the y-z plane. As seen from the camera's home position, this means that things on the left hand side of the screen will be reflected to the right hand side.

Examples of use:

```
(define  right-hand                Defines "right-hand" to
         (mirror  left-hand))      be a mirror image of
                                   "left-hand".
```

```
(define  ship-hull
         (group  half-hull
                 (mirror  half-hull)))
```

Defines "ship-hull" as two mirror symmetrical halves.

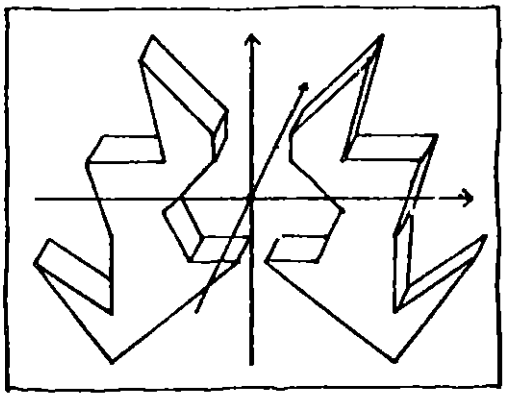


Figure 11

(group gwiz (mirror gwiz))

C U T - H O L E

See also: Polygon

When constructing polygons with the polygon function we are limited to forming polygons with just one boundary. To form multi-boundary polygons we use the cut-hole operator. Cut-hole allows any number of inputs, which all must be polygons. It returns one polygon which contains the boundaries of all of the input polygons, and the color of the first input polygon. General usage format:

(cut-hole <polygon1> <polygon2> ...)

Polygons are filled by "parity" (also known as the "checker-board" rule), this means that if we where to crawl across the plane of the polygon, every time we crossed a boundary the "filled-in-ness" of the polygon changes. If a polygon has two boundaries, one inside the other, the inner boundary forms a hole in the polygon. If there was yet another boundary inside the other two, it would surround a filled-in part, an "island".

Examples of use:

```
(define wall
        (cut-hole  rectangle  window-1  window-2))
```

Defines "wall" as a rectangle with two specified windows cut through it.

```
(define  board
         (cut-hole  board knot-hole))
```

Redefines "board" as having a "knot-hole" through it.

P R I S M

See also: Color, Vector, Polygon, Solid, Move, Cut-hole

In geometry, as in the animation system, a prism is a solid with two identically shaped, parallel polygons as "ends", and between the corresponding edges of the ends, "sides" in the shape of parallelograms. The prism operator expects three inputs; a color for the sides, a vector representing the relative positions of the "ends", and a polygon which will become one of the "ends". The other "end" is formed by moving the polygon along the vector (and reversing its front and back colors). Prism returns a solid which has the specified prismoid shape. Note that a polygon with "holes" may be used to make a prism, and the resulting solid will have holes through it. Usage format:

(prism <color> <vector> <polygon>)

Examples of use:

```
(defprog  pentagon-doughnut

          (inputs:  color size thickness hole-ratio)
          (local:   pent)

          (define   pent
                    (reg-poly  color 5))

          (define   pent
                    (cut-hole  (scale  size pent)
                               (scale  (times  size hole-ratio)
                                       pent)))

          (prism    color
                    (vector   0 0 thickness)
                    pent))
```

This is a function which forms a "pentagon-doughnut", this is the shape of the Armed Forces Headquarters in Washington D.C. The parameters represent; the color of the whole thing, the size from the center out to the edges, the thickness of the prism, and how much of the bulk is hole. The "reg-poly" function returns a regular polygon and is defined in the section on "Low Level Data Base Interface".

R E P L I C A T I N G O P E R A T O R S

See also: Group

Geometric operators like forward and rotate give us back an object with exactly the same structure as the object we passed to the operator. The group operator takes many objects and returns a single object whose structure contains all of the others. The programs known as replicating operators take one object, and return a group of many modified copies of the original object. Replicating operators function much like an

office photocopier, you give it an original, it gives you back a group of copies.

R O W

See also: Vector, Move

The row operator is used to make a row of copies of an object. Row expects two or three inputs, the first is the number of copies to make, the second is the object to copy. If a third input is given it is the vector we would use to move the original object to the position of the next object in the row.

```
(row  <count>  <object> )
(row  <count>  <object>  <vector> )
```

If the third input is not specified, it is assumed to be a unit vector in the "x" direction: (vector 1 0 0).

Typical uses for row include: making a multi-story building from a model of just one story, making rows of rows (columns) to form rectangular arrays, or to construct a picket fence from one picket:

```
(define  building
         (row  20
               single-story
               (vector  0 10 0)))

(define  rect-array
         (row  5
```

```
(row  5
      array-element
      (vector  50 0))
(vector  0 50)))
```

```
(define  fents
         (row  100 fent))
```

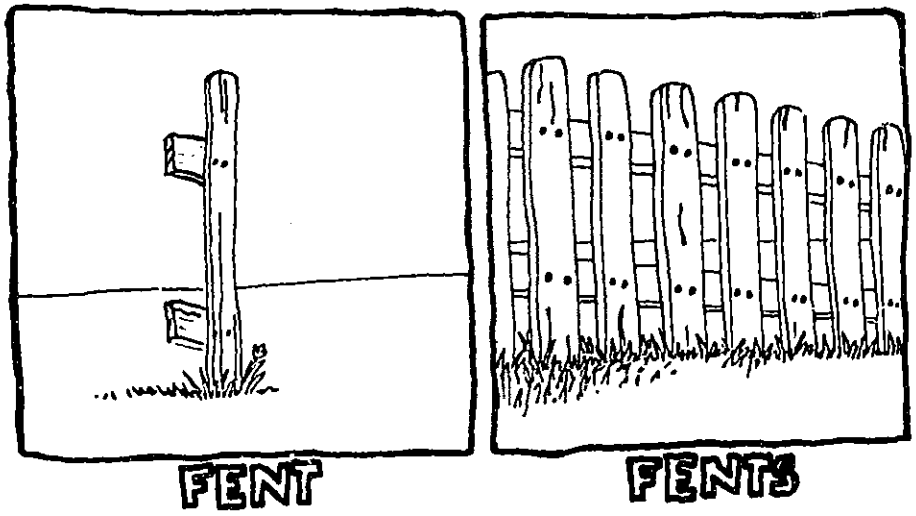


Figure 12

Adapted from "Never Eat Anything Bigger Than Your Head & Other Drawings" by B. Kliban.

R I N G

See also: Angle, Rotate

The ring operator forms multiple copies of an object, arranged along a circle. We specify the number of copies, and the object to be copied. If a third input is given, it is the axis about which to rotate the copies, otherwise the z-axis is used.

General usage format:

(ring <count> <object>)

(ring <count> <object> <axis>)

Examples of use:

```
(define  stonehenge              Defines "stonehenge" as a ring
         (ring  30 stones))      composed of 30 upright "stones".

(define  wheel

         (group  hub
                 rim
                 (ring  40 spokes x-axis)))
```

Defines "wheel" as being made up of a "hub", a "rim" and 40 "spokes" arranges about the hub.

R E P O P

See also: Row, Ring

Repop is a generalized replicating operator. Just as with row and ring, we specify the number of copies, and the object to be copied. In addition we specify an expression which tells how to form the "next" element of the group from the previous element. General usage format:

(repop <count> <object> <next-expr>)

The <next-expr> is any valid expression, and may contain references to the variable named "**". The <next-expr> is not evaluated until we are inside the repop. But when the <next-expr> is evaluated, the variable "**" will contain the previous element of the group. For example, to indicate that each element should be twice as large as the previous one we

would use this <next-expr>:

```
(scale  2 *)
```

As an example of the use of the repop operator, here is a definition of the two input version of row. We will call it "row2":

```
(defprog  row2
          (inputs:  count  object)

          (repop  count
                  object
                  (move  (vector  1 0 0)  *)))
```

C O N T R O L S T R U C T U R E S

The parts of the scripting language described in this section are those which make it a full programming language rather then simply a collection of predefined actions. The scripting features described in the second section are intended to provide for about 90% of the scripting that might be done with this system. For the oddball 10% however, it is possible to create your own programs to do what the predefined programs cannot.

To allow general programming, features are provided for attaching program definitions to names (defprog), for conditional evaluation (if, then, else), and for repetitive evaluation (loop). Programs can have parameters (input variables) and auxiliary storage (local variables). Parameters are passed just as in Lisp (if that does not mean anything to you, ignore it). All programs, like any other expression, return values.

D E F P R O G (Define Program)

See also: Program, Variables, Evaluation, Expressions

The defprog function is used to name and define programs. Once a program has been defined by defprog it can be invoked ("called") from scripts or from other programs (in fact a program can call itself).

```
(defprog   <prog-name>
           {declarations-of-variables}
           <expr1>
           <expr2>
           ...    )
```

Let us assume for a moment that we found that in some script we were writing, that we frequentally needed to double the size of various objects. We might then want to define a program to do this:

```
(defprog  double
          (input:  object)

          (scale  2 object))
```

This defines the program "double", making the name stand for the operation of scaling an object by 2. The defprog expression also declares that the program has one variable, an input variable (or "formal parameter" as they say in the formal computer science biz) named "object". This means that when we call this program from somewhere, we will pass it one input expression, which is evaluated to form the object to have its size doubled (that object is sometimes called the

"actual parameter").

The "body" (or executable portions) of the defprog follow the name and any program variable's declarations. The body may contain any number of expressions, during execution each expression is evaluated in turn. The value returned by the entire program (to the caller) is the value of the last expression in the body. In the example above, "double" only has one expression in its body, a call to the scale function, so the result of the scale is returned as the result of the invokation of "double".

To use the program "double" from somewhere else we would write the same kind of calling expression we would use for a predefined program, that is: a parenthesized list with the name of the program as the first thing in the list, followed by any input expressions for the program. Since "double" has just one input, it would have just one input expression. For example to make a copy of a cube twice the size of "cube":

```
(double  cube)
```

Or to double the size of a group of "cube" and "tube":

```
(double  (group  cube tube))
```

To define "c-and-t" to be the double sized group of "cube" and tube":

```
(define  c-and-t
```

```
(double  (group  cube tube))
```

I F (If-Then-Else)

See also: Cue, Evaluation

When a decision has to be made, a choice made between alternate actions, we use either the cue function or the if function. If expects 2 or 3 inputs, an expression to test for truth or falseness, plus a then clause and/or an else clause. These clauses are lists of expressions, starting with either then or else. The first input is evaluated, if it is true the body of the then clause is evaluated, otherwise the body of the else clause is evaluated. The value returned by the entire if is the value of the clause which was evaluated. Either of the clauses may be omitted (if an omitted clause is selected, its value is nil).

Usage:

```
(if  <test>
     (then  <then-body> )
     (else  <else-body> ))
```

or:

```
(if  <test>
     (then  <then-body> ))
```

or:

```
(if  <test>
     (else  <else-body> ))
```

Examples of use:

```
(if  (less  x 0)                          If "x" is non-positive,
     (then  (define  x 0)))               redefine it to be zero.

(if  (equal  a b)                         When "a" and "b" are
     (then  (start  zipper))              the same, activate the
     (else  (start  zapper)))             actor "zipper".
                                          Otherwise activate the
                                          actor "zapper".
```

L O O P

When we produce animation, there is an ongoing loop structure. Each frame we repeatedly go through the same set of cues, repeatedly activating each of the actors, and so on. However the implicit loop in an animate expression is not the only place where it is convenient to use the notion of a "loop". For this reason, the loop construct is provided. A loop is the programming equivalent of saying "and so on ...".

When there are several similar tasks to be performed, we could either repeat the expression needed to accomplish the task several times, or we can use the loop construct to execute the same expression several times.

For example, assume we want to print the value of the variable "slunk" six times, we could write:

```
(print slunk)
(print slunk)
(print slunk)
(print slunk)
(print slunk)
(print slunk)
```

or we could use a loop:

```
(loop  (repeat 6)
       (print slunk))
```

A loop expression is a parenthesized list of expressions starting with the word "loop". Each expression in the body of the loop is executed in turn, and in the absence of any loop control expressions (see below) this will repeat over and over. The looping will go on until the end of time, or when the program is interrupted from the user's console (whichever comes first).

Normally a loop is exited when any of the loop termination conditions occur. The programs which can cause these conditions are: repeat, while, until, and loop-exit.

Repeat expects a number and optionally a return value expression:

(repeat <count>)

(repeat <count> <return-value-expr>)

When a repeat expression is executed for the (<count> + 1)th time, the surrounding loop will be exited, if a return value

expression was specified, it is evaluated and returned for the value of the entire loop.

The while expression is similiar, except the first input is an expression to test for "true". If the value is "true" then the loop will continue on. But if the expression is "false", the surrounding loop is exited, and the return expression (if any) is returned as the value of the loop:

(while <test>)

(while <test> <return-value-expr>)

The until expression is just about the same as the while expression, except that the sense of the test is reversed. The loop will exit when the <test> is "true":

(until <test>)

(until <test> <return-value=expr>)

Loop-exit causes an unconditional exit of the surrounding loop, an optional input will be evaluated and returned by the loop:

(loop-exit)

(loop-exit <return-value-expr>)

One other loop control program is provided loop-top, which expects no inputs and simply jumps, to the top (start) of the

loop in which it occurs:

(loop-top)

In addition to the loop control expressions, and any other executable expressions, the first thing in a loop body may be a local variables declaration expression. These variables may be given initial values, and will disappear when the loop exits.

Examples of use:

```
(loop  (local:  (i  0))
       (while  (less  i 20))
       (inc  i)
       (run  zipper
             (for  1000)
             (define  speed i)))
```

Starts 20 instances of the actor "zipper" with assorted "speed"s.

```
(loop  (define  x
               (get-next-one))
       (until  (equal  x 0))
       (process-it  x))
```

This will repeatedly call "get-next-one", and if its value is not zero, it will "process-it". As soon as an "x" is encountered which is zero the loop will exit. Note that the until expression is embedded in the loop body, just as any of the other loop control programs may be. This is a solution to the "loop-and-a-half" problem, which is common to many programming languages.

N U M E R I C O P E R A T O R S

See also: Numbers

The numeric operators are functions for performing basic arithmetic operations on numbers. Functions are supplied for addition, subtraction, multiplication, division, and square rooting. Programs are also provided which increment or decrement a variable's value.

The plus function takes any number of numerical inputs and returns their sum.

The difference function (also called dif) takes any number of numerical inputs, dif of two inputs is normal subtraction. Dif of other numbers of inputs are handled like this:

```
(dif  a)                  ==     -a
(dif  a b)                ==     (a - b)
(dif  a b c d e ...)      ==     ((((a - b) - c) - d) - e) ...
```

The times function takes any number of numerical inputs and returns their product.

The quotient function (also called quo) takes any number of numerical inputs, quo of two inputs is normal division. Quo of

other numbers of inputs are handled as in dif.

The square-root function (also called sqrt) takes one numerical input and returns its square root. Remember that using real numbers (as we are), the square root of a negative number does not exist.

The increment and decrement operators (inc and dec) are used to redefine a variable's value as having had one added to, or subtracted from it. These are used to manipulate "counters", variables used to keep track of how many times we have done something. For example, every time somebody passes through a turnstile, a mechanical counter is incremented. Inc and dec are just shorthand for the use of define and plus, these two expressions do exactly the same thing:

```
(inc  sheep-count)        (define  sheep-count
                                   (plus  1 sheep-count))
```

Example of use:

```
(defprog  magnitude
          (input:  vector)
          (sqrt  (plus  (sq  (vx  vector))
                        (sq  (vy  vector))
                        (sq  (vz  vector)))))

(defprog  sq
          (input:  x)
          (times  x x))
```

These programs serve as an implementation of the vector operator magnitude which calculates the length of a vector. The functions vx, vy, and vz are used to extract the coordinates of a vector. Note the use of an auxiliary function "sq" used to square numbers.

C O M P A R I S O N A N D T E S T I N G O P E R A T O R S

See also: Cue, If, While, Until, Low Level Data Type Predicates

These testing functions answer questions about their inputs, such as "is A equal to B?" or "is A numerically less then B?". They return a value which represents "true" or "false" (t or nil). These values are used by various control programs (such as if) to determine which path to take through the user's program.

The empty function tests to see if its one input is an empty

list, that is "()" or nil.

The less function tests its two numerical inputs to see if its first input is less then its second input.

The less-eq function tests its two numerical inputs to see if its first input is less then or equal to its second input.

The greater function tests its two numerical inputs to see if its first input is greater then its second input.

The greater-eq function tests its two numerical inputs to see if its first input is greater then or equal to its second input.

The equal function tests its two inputs (which may be of any type) to see if they are equal.

The and function takes any number of expressions (usually comparison operators) and returns "true" only if all of the input values themselves where "true".

The or function takes any number of expressions (usually comparison operators) and returns "false" only if all of the input values themselves where "false".

Examples of use:

```
(less  a b)                          Is "a" less than "b"?

(greater  x 0))                      Is "x" positive?

(defprog  less-eq                    This is an implementation of
          (input:  p q)              the less-eq function, using
                                     less, equal, and or.
          (or  (less   p q)
               (equal  p q)))
```

G E N E R A L P U R P O S E U T I L T I E S

These programs are not strictly part of the Actor/Scriptor Animation System, in fact they are part of the Lisp system in which ASAS is implemented. They are however often useful for the animator to know about.

Help is used while you are at the console to get information about the system. The usage is:

(help <topic-name>)

The <topic name> is not evaluated, the name itself is used. If the topic name is "*", a list of all available topics is printed. To get information about a specific topic (say actor) type "(help actor)" and the documentation will be printed at your console. For the most part, the documentation will be the same as the corresponding section in this User's Manual.

Read is a function of no inputs, which returns the next full expression from the user's console. It waits until the expression is complete (all parenthesis must balence for example).

Print expects one input, and will print the value of that expression on the console.

Back trace (or bt) is used to help describe what happened after an error condition occured. After the error message, you are not back to where you started, you are still "inside" the program which got the error. To see how you got there, type:

(back trace)

It will give a list of the programs which you have entered, but not yet exited. Hence the list "a, b, c" means the error occured in "a", which was called from "b", which was called from "c".

(bt t t)

This will list the entire expression from which each program was invoked.

Release (or rl) is used to get out of the error handler, and down to the previous level. Pending program variables will then be restored to their previous value. Note that error

handlers can "stack" so that if you get two errors in a row without releasing you will be on the second level.

Quit gets you out of the Lisp interpreter, and hence out of the animation system. It does a "release all levels" before exit. Quit is one of the rare functions that does not return a value, it just puts you back into the MagicSix operating system.

V I D E O P R O D U C T I O N U T I L I T I E S

See also: Script, Animation

The programs described in this section are used only in the final stages of animation production. They are used to control the graphical output of the animation system, and to specify the production time options. Programs are provided to produce test patterns, count downs, and black leader.

A N I M A T I O N - M O D E

The animation-mode program sets system parameters which effect the action of the system as it produces an animated sequence. This program has two names, for typing convenience, the name amode may be used. The program expects two inputs, the first is not evaluated and specifies the name of one of the animation mode flags. The second input is the new value for for that mode flag, which may be either the word "on" or "off" or any other expression to be evaluated:

(animation-mode <mode-name> <new-mode-value>)

Modes:

Name:	Description:	Allowable values:
display	(display enable)	on / off
eof	(end-of-frame message enable)	on / off
number	(display frame number on each frame)	on / off
record	(video disk recording enable)	on / off
pause	(wait on console each frame)	on / off
trace	(active actor tracing)	<actor-id> / off
untrace	(removes from tracing list)	<actor-id>

The on/off modes are straitforward, they act like light swithes. Pause mode is used to "step through" the execution of a script, each frame the system will pause and listen to the user's console. At that time the animator may either just tell it to go on, or may examine the state of program variables.

The trace mode is used to watch the action of particular active actors. The value "off" stops any ongoing tracing. If the value is a number, it is presumed to be a actor identification number (as returned by start, or run). If it is a valid id number, that actor is added to the tracing list. Then at the appropriate time during the animate loop, each traced actor is listed at the console.

Examples of use:

(animation-mode record off)	Tells the system not to record frames on the disk.
(amode trace actor1)	Adds "actor1" to the tracing list.

The black program is used to produce frames with nothing on them, this is usually put at the beginning and end of a production, and between any sequences which must be seperated. Black expects one input, the number of seconds of black frames to produce:

(black 5)	Puts out 5 seconds, or 150 frames of black.

C O L O R - B A R S

Color-bars is a standard test pattern used to calibrate color balence information on video recordings. It consists of patches of the primary and secondary colors, as well as gray, "I" and "Q". The program expects one input, the number of seconds of "bars" to produce.

T E S T - P A T

Test-pat is a color, gray level, linearity, and resolution tester. It also includes the date produced. The test-pat program expects one input, the number of seconds to produce.

C O U N T - D O W N

The count-down function is used to mark the actual beginning of the animated production. It is used by anyone who will mix your animated sequence with anything else (for example, a television program about computer animation). The start of the count down is a "warning" that the real start is coming up. The count serves to allow the person doing the mixing to synchronize the end of the previous scene with the start of your animated sequence.

S L A T E

A slate is the information frequently put at the start of a piece of film or tape, giving the date produced, title of the production, and the people involved in the production. The slate program expects one or more input expressions, the first input is the number of seconds of slate desired. The next two input expressions will not be evaluated, but are considered to be the title and the animator's name. These are specified as either a single word, or a parenthesized list of words. Any additional input expressions are treated as more text for the slate. The date and time of the production are automatically added to the slate. General usage format:

(slate <seconds> <title> <animator>)

or:

(slate <seconds> <title> <animator> <textl> ...)

Example of use:

```
(slate  5
        (Actor/Scriptor Animation System Demo)
        (Craig W. Reynolds)
        (The Architecture Machine Group - MIT)
        (Cambridge, Massachusetts))
```

L O W L E V E L G E O M E T R I C A L O P E R A T O R S

See also: Vector, Polygon

These functions provide low level manipulation of certain geometrical objects. They are often useful when defining your own geometrical operators.

The vadd and vsub functions add and subtract vectors, the result is another vector. Each expect two inputs, both vectors.

The magnitude function expects one vector as input, and returns the magnitude or length of the vector.

Three functions are provided to extract the "x", "y", and "z" coordinates from a vector. The functions are called vx, vy, and vz; they each expect one vector input.

The area function expects one polygon as input, and returns the area of the orthagonal projection of the polygon onto the x-y plane. If the area is positive, the "front" of the polygon points towards the x-y plane, otherwise it points away. The area is expressed in units of pixel areas; hence if a polygon has an area of 123.4 this means that it would cover about 123 pixels if projected onto the display screen.

L O W L E V E L D A T A B A S E I N T E R F A C E

See also: Data types

Occasionally we cannot construct the objects we desire using the standard operations. The data base primitives are essentially used to convert between Actor/Scriptor Animation System data types and Lisp lists.

The color-of function returns the color of its one input, a geometrical object.

The lofcof function takes one geometrical object and returns a list of the contents of the object.

The bld- functions take a list of "contents" and build the specified type of object. For example bld-vec takes a list of coordinates and builds a vector data type from them. Here is a list of the bld- functions and expected args:

```
(bld-color  <red>  <green>  <blue> )                    Color

(bld-grp  <list-of-objects> )                           Group

(bld-poly  <color>  <list-of-vectors> )                 Polygon

(bld-pov  <list-of-vectors> )                           Pov

(bld-sld  <color>                                       Solid
          <list-of-vertex-names>
          <list-of-vectors>
          <list-of-faces>    )

(bld-sub  <pov>                                         Subworld
          <color>
          <size>
          <list-of-objects> )

(bld-vec  <list-of-coords> )                            Vector
```

Example of use:

```
(defprog  reg-poly

          (inputs:  color sides)

          (bld-poly  color
                     (lofcof  (ring  sides
                                     (vector  0 1)))))
```

This is a program to create regular polygons, as inputs it takes the number of sides desired on the polygon and its color. It works by creating a ring of vectors, (which is a group or vectors), taking the ring's contents, and using them to build the new polygon.

Because the "type" of an object can be determined by examining it at run type, a set of functions are provided to test the

type of a given object. These "predicate" functions test for a particular type and return a "true" or "false" result. Each of the predicate functions expects just one input, the object to test. The names of the predicates are the name of the type, followed by the letter "P", for predicate. Additionally the gobjectp predicate test to see if its input is any of the valid geometrical objects. Usage formats:

(actorp <object-to-test>)

(colorp <object-to-test>)

(groupp <object-to-test>)

(gobjectp <object-to-test>)

(numberp <object-to-test>)

(polygonp <object-to-test>)

(povp <object-to-test>)

(solidp <object-to-test>)

(subworldp <object-to-test>)

(vectorp <object-to-test>)

B I B L I O G R A P H Y

1
Austin, H., "The LOGO Primer", MIT A.I. Lab. Logo Working Paper 19.

2
Birtwistle, Dahl, Myhrhaug, and Nygaard, SIMULA Begin, Auerbach 1973.

3
Burtnyk, N. and Wein, M. "Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation", CACM, October 1976, p. 564.

4
Church, A "The Calculi of Lambda Conversions", Annals of Mathematical Studies 6, Princeton University Press 1941.

5
Dahl, Myhrhaug, and Nygaard "The SIMULA 67 Common Base Language", Norwegian Computing Centre, Oslo, 1968.

6
Dijkstra, E.W. "Notes on Structured Programming", August 1969

7
Goldberg, A. "SMALLTALK and Kids -- Commentaries", Learning Research Group, Xerox Palo Alto Research, 1974

8
Goldberg, A. and Kay, A. "SMALLTALK-72 Instruction Manual" Learning Research Group, Xerox Palo Alto Research, March 1976.

9
Goldstein, I., Abelson H., and Bamberger, J. "LOGO Progress Report 1973-1975", MIT A.I. Lab Memo 356, March 1976.

10
Greif, I. and C. Hewitt "Actor Semantics of PLANNER-73", Proc. of ACM SIGPLAN-SIGACT Conf., Palo Alto, Ca., January 1975.

11
Hewitt, C. and Atkinson, R., "Parallelism and Synchron-

ization in Actor System", ACM Symposium on Principles of Programming Languages 4, January 1977, L.A. California.

12
Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages", A.I. Memo 410, MIT A.I. Lab, 1976.

13
Hewitt, C. and B. Smith, "Towards a Programming Apprentice", IEEE Transactions on Software Engineering SE-1, March 1975.

14
Jones, B. "An extended ALGOL-60 for Shaded Computer Graphics", ACM SIGPLAN/SIGGRAPH Symposium on Gr Languages, April 1976.

15
Kahn, K. "An Actor-Based Computer Animation Language", Proc. of the ACM-SIGGRAPH Workshop on U Design of Computer Graphics Systems, Pittsburg, P October 1976.

16
Kahn, K., "A Computational Theory Of Animation", MIT A.I. Lab. Working Paper 145, April 1977.

17
Kahn, K., MIT Doctoral Thesis, to be submitted this year.

18
Kitching, A. and Emmett, C. "The ANTICS Computer Animation System", ACM-SIGGRAPH Newsletter: Computer Graphics, Winter 1976.

19
Lieberman, H. "The TV Turtle: A Logo Graphics System for Raster Displays", ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages, April 1976.

20
Lieberman, H. and Kahn, K. "Computer Animation: Snow White's Dream Machine", Technology Review magazine, MIT, October-November 1977.

21
McCarthy, Abrahams, Edwards, Hart, and Levin, "Lisp 1.5 Programmer's Manual", MIT Press, Cambridge, Mass. August 1962.

22

Mezei, L. and Zivian, A. "ARTA: An Interactive Animation Animation", Technical Report, University of Toronto, 1976.

23
Moon, D. "Maclisp Reference Manual", MIT Project MAC memo, December 1975.

24
Negroponte, N., "The Return of the Sunday Painter or The Computer in the Visual Arts", Future Impact of Computers and Information Processing, Dertouzos and Moses editors. 1978.

25
Newman, W. and Sproull, R. Principles of Interactive Computer Graphics, McGraw - Hill, 1973.

26
Papert, S., "Teaching Children Thinking", MIT A.I. Lab, Memo 247, October 1971.

27
Papert, S., "Teaching Children To Be Mathematicians vs. Teaching About Mathematics", MIT A.I. Lab, Memo 249, 1971.

28
Papert, S., "A Computer Laboratory for Elementary Schools", MIT A.I. Lab, Logo Memo 1, October 1971.

29
Pfister, G. "A High Level Language Extension for Creating and Controlling Dynamic Pictures", ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages, April 1976.

30
Rivlin, R. "Electronic Image Creation and Manipulation System", Millimeter magazine, October 1976.

31
Smoliar, S. "A Parallel Processing Model of Musical Structures" MIT A.I Lab. AI-TR-242, September 1971.

32
Sutherland, I. "Sketchpad: A Man-Machine Graphical Communication System", MIT Lincoln Lab TR-296, 1963.

33
Watkins, G. "A Real-Time Visible Surface Algorithm", Univ. of Utah Tech. Report, UTECH-CSc-70-101, June 1970.

34
Wirth, N. "MODULA: a Language for Modular Multiprogramming", Software, Practice and Experience 7,1; 1977 pp. 3-35.