

## Computer Animation with Scripts and Actors

by Craig W. Reynolds

Information International Inc.

### Abstract

A technique and philosophy for controlling computer animation is discussed. Using the Actor/Scriptor Animation System (ASAS) a sequence is described by the animator as a formal written SCRIPT, which is in fact a program in an animation/graphic language. Getting the desired animation is then equivalent to "debugging" the script. Typical images manipulated with ASAS are synthetic, 3D perspective, color, shaded images. However, the animation control techniques are independent of the underlying software and hardware of the display system, so apply to other types (still, B&W, 2D, line drawing ...). Dynamic (and static) graphics are based on a set of geometric object data types and a set of geometric operators on these types. Both sets are extensible. The operators are applied to the objects under the control of modular animated program structures. These structures (called **actors**) allow parallelism, independence, and optionally, synchronization, so that they can render the full range of the time sequencing of events. **Actors** are the embodiment of imaginary players in a simulated movie. A type of animated number can be used to drive geometric expressions (nested geometrical operators) with dynamic parameters to produce animated objects. Ideas from programming styles used in current Artificial Intelligence research inspired the design of ASAS, which is in fact an extension to the Lisp programming environment. ASAS was developed in an academic research environment and made the transition to the "real world" of commercial motion graphics production.

CR Categories and Subject Descriptors: I.3 [Computer Graphics];

I.3.5 [CG]: Computational Geometry and Object Modeling;

I.3.6 [CG]: Methodology and Techniques—Languages;

I.3.7 [CG]: Three-Dimensional Graphics and Realism—Animation

General Terms: Design, Languages

Additional Key Words and Phrases: Lisp, Procedural Animation Languages, Motion Picture Production

Author's addresses:

US Mail: III, 5933 Slauson Ave., Culver City, CA 90230

ARPAnet mail: Reynolds@Rand-AI

Uucp network mail: ucboxrandvax!reynolds

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

### Introduction

This paper describes the Actor/Scriptor Animation System (ASAS), which is a way of thinking about and describing computer graphic animation. ASAS is basically a *notation* for animated graphics. The notation for an animated sequence (the **script**) can be automatically read and converted into animated images by an ASAS interpreter. As in the case of musical notation being interpreted by a group of musicians—or the script of a video production being executed by a host of actors, camera, audio, lighting and video technicians—ASAS allows the creation and use of any number of simulated participants, "**actors**" each of which can control one or more aspects of the animation. The ability of ASAS **actors** to operate independently or (by communicating with each other) to act in synchronization allows a simple and unambiguous description of the function of each **actor**.

ASAS differs from "performance" based real time computer graphics systems as well as from command or "menu" based systems. Writing the ASAS notation for an animated sequence will probably take longer than the final running time of the sequence. On the other hand, an ASAS **script** is typically more compact than a simple listing of the value of all relevant parameters for each frame, as might be required in a command-menu system. This results from the fact that ASAS is a procedural notation, a programming language for animation and graphics. In fact ASAS is a "full" programming language and includes all of the typical modern structured programming features (procedures (recursive), local variables, "if then else"s loops, typed data structures and generic operators). Additionally ASAS supports independent, parallel, "animated" program structures (**actors**), and includes a rich set of geometric and photometric objects and generic operators on these objects.

The existence of a formal notation for a field of endeavor leads to a workable procedure for the development of an idea. Like an algorithm being debugged by a computer programmer, or a musical score being revised, an ASAS **script** being developed is both unambiguous and precisely modifiable. It is possible to change just one small aspect while keeping everything else exactly the same. This property of notation allows the process of progressive refinement ("tweaking") to be used to converge on the desired algorithm, music or animation.

### History

ASAS was developed at the Architecture Machine Group at MIT as two thesis projects between 1975 and 1978 [24,24]. "ASAS 0" was not a full implementation, but "ASAS 1" did actually work,

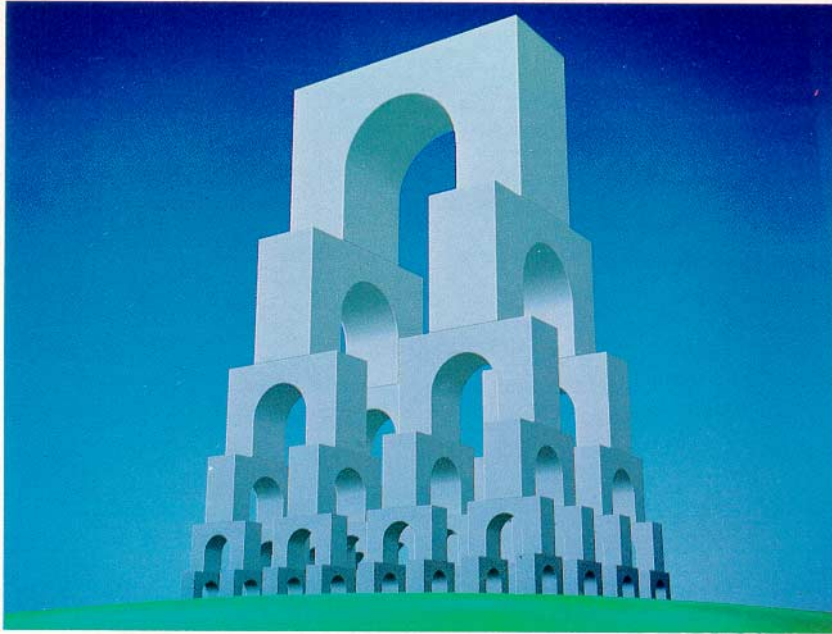


Figure 1: An Arch Fractal

despite a very slow and uninteresting display package. In 1979 ASAS was integrated into the Digital Scene Simulation system of Information International Inc. ("III", "triple-I") in Culver City, California. In this instance, ASAS is not used to make images directly, but serves as a preprocessor for III's existing 3D, hidden surface and shaded graphics system. Hence "ASAS 2" functioned as a true language compiler, translating from the animator's **script** to the command sequence for the display software. The inconvenience of having the display support in a separate software package is offset by the much wider range of graphic features made available to the ASAS user through the very advanced III software. After three years of commercial use the system was refined and became "ASAS 3". The current reference for the language is the unfinished **ASAS User's Manual 3.0**. [26]

The design of ASAS was influenced by some concepts from research in the Artificial Intelligence field. The basic concept of graphic databases and animation scripts as programs (procedural embedding of knowledge) was inspired by Terry Winograd's pioneering work in computer linguistics. In Winograd's system natural language was represented by a procedural data structure. [30] The concept of message passing **actors** was from Carl Hewitt's body of work in "actor systems" such as PLASMA. [12,13,14] (Similar concepts exist in Smalltalk [11], Simula [5] and Modula [32].)

An animation system in development at about the same time as ASAS by Ken Kahn shared some concepts with ASAS. [16,17] Kahn's system had what Hewitt calls a "uniform actor basis" and so perhaps a theoretically "cleaner" structure. Kahn's work placed more emphasis on embedding common-sense and theatrical knowledge in animation characters, and less emphasis on complex graphics.

#### ASAS and Lisp

As a programming language, ASAS "stands on the shoulders of

giants" because of its relationship to the programming language Lisp. ASAS can be considered to be either implemented in, or an extension to, Lisp. The rich programming environment of ASAS is due largely to Lisp, since all Lisp primitives and utilities are usable from ASAS (and vice versa). A Lisp interpreter plus the ASAS software yields an ASAS interpreter. ASAS was developed under MagicSixLisp at The Architecture Machine Group at MIT and was painlessly transplanted to run under MIT MacLisp at III.

For a long time Lisp has had a reputation as somewhat of a "toy" language, a powerful but quaint tool used by gnomish academic artificial intelligence researchers, but not a language really suited for commercial use. These criticisms are misdirected, Lisp is one of the most elegant and useable algorithmic notations ever devised. The bad reputation is due mainly to poor implementations or underpowered computers. Because Lisp

trades off raw computational efficiency for expressive power and usability, a well designed interpreter and a fast machine are required for a production environment. Today there are several good Lisp systems for various types of general purpose computers (MacLisp [20], InterLISP [29], ...) and currently three firms are selling specially designed Lisp machines (Lisp Machine Inc., Symbolics Inc., and Xerox)

#### ASAS Expressions

ASAS and Lisp use a simple, if unusual, notation. A "parenthesized prefix notation" is used for operators, control structures and data. An ASAS **expression** is either:

- (1) a number
- (2) a symbol ("variable")
- (3) a parenthesized list of **expressions**.

If the expression is a list, the first (leftmost) thing is the name of an operator (or "function") and any other expressions in the list are parameters for the operator. When an **expression** is evaluated (or "executed"), numbers evaluate to themselves, symbols to their currently defined value, and a list (eg: "**(plus a (abs b))**") evaluates to the result of applying the operator ("**plus**") to the recursively evaluated parameters (values of "a" and "**(abs b)**"). For example, to define the symbol "wheels" to be the number of tricycles times 3, we would write:

```
(define wheels
  (times tricycles 3))
```

Normal ASAS operators (like **times**) evaluate each of their parameters, while certain operators have special evaluation patterns (like **define**, which does not evaluate its first parameter, these are called "macros"). To define a simple operator (call it "thrice") which multiplies its single parameter ("x") by three:

```
(defop thrice
  (param: x)
  (times x 3))
```

an equivalent definition in MacLisp would be:

```
(defun thrice (x) (times x 3))
```

The first example could then be rewritten:

```
(define wheels
  (thrice tricycles))
```

### Special Symbols

Within a **script** certain aspects of the production are controlled by the values given to some special symbols. None of these symbols are actually "reserved words", but it is best to use the **script** symbols **background** and **camera** only for the purpose of defining the current **color** of the graphical background of the image and the current camera description (as a **pov**, see the section on geometric objects). The initial ASAS environment has other symbols defined to various frequently used objects (axes, colors, basic solids), it is good practice to know these and avoid redefining them.

### Geometric Objects

In addition to the data types found in most programming languages, ASAS provides a set of geometric (and photometric) objects: **vector**, **color**, **polygon**, **solid**, **group**, **pov**, **subworld**, and **light**.

The **vector** represents a position in three dimensional Cartesian space. It allows three parameters, the X, Y and Z coordinates. Trailing zero coordinates may be omitted.

A **color** object may be specified either by its Red, Green and Blue components, or by Intensity, Hue, Saturation. The two operators are called **rgb** and **lhs**, each of which accept three numbers between 0 and 1.

A simple **polygon** contains a **color** and a list of **vectors**, the "boundary". The **cut-hole** operator allows the construction of **polygons** with "holes and islands" (that is, multiple boundaries). The **color** can be a **group** of a front color and a back color. The boundary points may be listed separately or as a **group** of **vectors**. Here is a **polygon** expression for a certain blue triangle:

```
(polygon blue
  (vector 1 0 0)
  (vector 0 1 0)
  (vector 0 0 1))
```

A **solid** represents a bounded region of space, a closed polyhedron. It is composed of vertices and faces (as **vectors** and **polygons**) in addition to topological connection information.

Several geometric objects can be "glued together" into a **group** object, which is then manipulated as a whole by the geometric operators. A **group** expression allows any number of parameters -geometrical objects to be **grouped** together, including other **groups**.

```
(define houses
  (group red-house yellow-house brown-house))
```

The "point of view" object (**pov**) is used to define the point of view of an observer (for example the ASAS **camera**) or of an object. That is, a **pov** describes the three coordinate axis basis vectors and the position of the origin of an arbitrary coordinate space. We refer to such spaces by names like "eye space" and "an object's local coordinate space". Note: a **pov** plays a role very similar to a "4X4 homogeneous transform matrix" in other 3D graphics systems (there is a simple transformation from a **pov** to a 4X4 matrix) but a **pov** is a geometrical object composed of **vectors** and can be manipulated just like any other object.

A **subworld** is an object associated with a **pov**. This allows ASAS to manipulate a complex object by modifying only the **pov**, hence various "instances" of an object may share the same underlying data. **Subworlds** also allow ASAS to work with "levels of abstraction" in a graphic database, when a **subworld** is formed it notes the "overall size" and "typical color" of its contents. At display time this allows efficient tree structured clipping (when an entire **subworld** is offscreen) and handling of detail too small to see (when an entire **subworld** lies within a single pixel). [4] Hence the user can build levels of abstraction

Figure 2: How to make an Arch Fractal (given an arch element).

```
(defop arch-fractalizer
  (param: arch-element top-color bot-color levels
    fractal-ratio height width leg-width)
  (local: (total-levels      levels)
    (offset-dist            (half (dif width leg-width)))
    (sub-tower-offset-1    (vector offset-dist 0 0))
    (sub-tower-offset-2    (mirror x-axis sub-tower-offset-1)))
  (arch-tower levels))

(defop arch-tower
  (param: levels)
  (if (zerop levels)
    (then nothing)
    (else (add-arch-level (arch-tower (dif levels 1))))))

(defop add-arch-level
  (param: sub-tower)
  (grasp sub-tower
    (scale fractal-ratio)
    (move (vector 0 height 0))
    (rotate 0.25 y-axis))
  (grasp arch-element
    (recolor (interp (quo levels total-levels)
      bot-color
      top-color)))
  (subworld (group arch-element
    (move subtower-offset-1 sub-tower)
    (move subtower-offset-2 sub-tower))))
```

into a geometric database by the nesting of **subworld** objects.

Objects to be seen in shaded images are illuminated by **light** objects. **Lights** are composed of a position **vector** and a **color**.

### Geometric Operators

ASAS's geometric operators are the tools the animator uses to shape, move and orient objects. An object's shape may come directly from the action of operators, or parts encoded by hand with a digitizer can be assembled with the operators. The same operators are used both for static arrangements, or to create animated motion, by operating frame by frame under the control of an **actor**.

In many command/menu based graphics systems it is difficult to precisely specify the correct ordering of geometric transformation. For example, there will be a "rotate" command which accepts three numbers, the angles of rotation for each axis. Often there is no mention of in what order the rotations are applied, let alone a way to specify the desired order. In ASAS, the animator explicitly determines the ordering of operations by the structure of the nesting of the expressions written in the **script**.

The basic operators are "generic", they can be given any type of geometric object and operate on it as is appropriate for that object's type. ASAS operators NEVER modify the object they are operating on. The value returned by an operator is a geometrically modified copy of the original object with otherwise the same type and structure.

A notational shorthand is provided for the common occurrence of a series of operations to be performed on a single object. The object to be operated upon can be made the "current" object (using the **grasp** operator). The "**grasped**" object will then be redefined by calls to operators which do not explicitly specify an object to operate on.

Two basic types of geometric operators are provided by ASAS, "global" and "local" (sometimes called "self relative"). The ASAS global geometric operators are called: **scale**, **move**, **rotate**, **stretch** and **mirror**.

Generally these operators apply the named geometrical transform to any given geometrical object. The transforms are relative to the origin and major axes of the global coordinate space. The parameter types to each are numbers and **vectors** as appropriate. ("Stretch" is a differential scaling for each axis, specified by a **vector** of scale factors).

As an example of the usage of the ASAS global operators see Figures 1 and 2, these show how last year's SIGGRAPH cover was constructed.

The "local" operators are similar in effect to the global operators, except that they are based on an object's OWN coordinate system rather than the global coordinate system. A **subworld** carries along its own little coordinate system, its **pov**. Not only does this allow efficient modification of the **subworld** but it also provides a reference for operations in the object's local coordinate space. The local operators were inspired by the "turtle" of the LOGO graphics language [2], and are intended to be a three dimensional analog of the turtle operations (walk forwards or backwards, turn right or left). This notion of a 3D turtle (more of a deep sea swimming turtle than a land crawling

tortoise) was first used by Jim Stansfield and then refined by Henry Lieberman in a 3D line drawing extension to LOGO. A good treatment of this subject can be found in [1]. Usually objects will be defined so that the origin of their local coordinate space is at the center of the object. For this reason we will informally refer to the "origin of the local coordinate system" as the "center". Local operators are provided for moving, rotating, scaling and "zooming" relative to the local coordinate system. All of these operators accept one or two parameters, the second optional parameter is the object to operate on, if none is specified, the currently **grasped** object is redefined.

Note: the relationship between global and local operators is similar to the that of pre- and post-multiplication of transform matrices. Also note: when objects other than **subworlds** or **povs** are passed to self relative operators they are first put into an identity ("home") **subworld**, then operated on.

Local operators:

<b>grow</b>	scale up about local center
<b>shrink</b>	scale down about local center
<b>forward</b>	move along local +Z axis
<b>backward</b>	move along local -Z axis
<b>left</b>	rotate to left about local Y axis
<b>right</b>	rotate to right about local Y axis
<b>up</b>	rotate upward about local X axis
<b>down</b>	rotate downward about local X axis
<b>cw</b>	rotate clockwise about local Z axis
<b>ccw</b>	rotate counter-clockwise about local Z axis
<b>zoom-in</b>	scale up local Z axis
<b>zoom-out</b>	scale down local Z axis
<b>local-move</b>	move along arbitrary local vector
<b>local-stretch</b>	scale each local axis independently
<b>home</b>	resets back to original definition space

Examples: an operator sequence which (if evaluated each frame) will cause the AIRPLANE (that is, the sequence of objects which form the animated value of the variable AIRPLANE) to perform "barrel rolls", and one to cause the CAMERA to pan around while zooming out:

```
(grasp airplane)
(forward 0.1)
(cw 0.02)
(up 0.02)

(grasp camera)
(right pan-speed)
(zoom-out 1.01)
```

Various other ASAS operators are available but will not be discussed here. There are **recolor** and **cut-hole**, and **interp** the general purpose interpolater, **row** and **ring** which make regular **groups** of objects, and **prism** which makes solids by projecting a **polygon**. Here are some examples of some of them, an operator to make a n-sided regular polygon (inscribed within a unit radius circle), and an operator to make a prism with regular "ends":

```
(defop regular-polygon
  (param: color sides)

  (polygon color
    (ring sides
      (vector 0 1 0)
      z-axis)))

(defop regular-prism
  (param: color sides thickness)

  (prism color
    (vector 0 0 thickness)
    (regular-polygon color sides)))
```

This summary of ASAS operators suffers because of the language's extensibility; the full list is endless since the user invents new ones as needed. Beyond simple combinations of basic linear operations, there is a large class of nonlinear "bending" operators. For example consider "curl-up" which takes a long thin object and curls it into a spiral (Escher fans will know the application for that).

### Scripts and Animate Blocks

The main program an ASAS user writes is called a **script**, which is a special type of **defop**. A **script** handles the setting up and setting down needed to produce an animated sequence (or write a file for later production by another system). The **script** expression includes a name and any number of subexpressions. The effect is to define an operator with that name which opens production, evaluates each expression in the body, and closes production. There is no restriction, but the things in the body are usually either **animate** expressions ("animate blocks") or production utilities (such as "make N blank frames", "put this slate text", or "make an N second countdown").

An **animate** block is a special type of loop. Each time around the loop, after it evaluates its body, a frame of animation is produced automatically. Usually the body contains **cue** expressions ("cue at frame N, ..."). These cause objects to be made visible (with the **see** operator) or start, stop or direct **actors**. **Animate** blocks are exited when a **cut** operator is evaluated.

This example **script** contains one **animate** block, which starts two similar **actors** at different times. Both **actors** then run until the end of the block.

```
(script spinning-cubes
  (local: (runtime 96)
    (midpoint (half runtime)))

  (animate (cue (at 0)
    (start (spin-cube-actor green)))

    (cue (at midpoint)
    (start (spin-cube-actor blue)))

    (cue (at runtime)
    (cut))))
```

(Note: **cut** accepts an optional frame number, and will cut only if that is the current frame, so that third **cue** could have been written as "(**cut runtime**)") When an **animate** block is exited, all of the **actors** associated with it are stopped. Hence **animate** blocks are somewhat like the "scenes" of a movie, the coarse

structure of the action.

While an ASAS "cue" is in fact simply a number (a frame number relative to the current **animate** block) it should not be thought of as a constant. Because of the computational nature of a **script** it can be quite easy to move cues around, since all cue points can be handled symbolically (by name rather than by a literal number). For example it is a simple matter to change the overall runtime of a **script** (for a "quick run through" test) if all cue points are defined relative to one variable (e.g. "runtime"). Library macros exist to facilitate just such a scheme. The animator may find it necessary for artistic reasons to move a cue point within a **script**, again this will be quite painless if everything which is supposed to begin or end at that cue point refers to it only symbolically.

### Actors

The control structure of an animation system would be very simple if we could assume that all sequences to be produced had at most one independently animated feature at any one time. On the other hand, if we assume that there may be any number of fully independent animated features (starting and stopping at random times, happening at different rates, running in sync or not) then conventional control structures are no longer the most appropriate.

An ASAS **actor** can be thought of several ways. Most basically an **actor** is a "chunk" of code which will be executed once each frame. Usually an **actor** (or a team of them) is responsible for one visible element in an animation sequence, hence it contains all values and computations which relate to that object. In this sense an **actor** serves to modularize and localize the code related to one aspect, isolating it from unrelated code. From a formal point of view, an **actor** is an independent computing process in a non-hierarchical system with synchronized activation and able to communicate with other **actors** by message passing.

When an **actor** is "on" (between being **started** and **stopped**), it will be awakened once each frame, its local variables restored, its body evaluated, its variables saved, then put back to sleep. (Hence an **actor** has properties between a "closure" and a "process" in recent Lisp implementations.) **Actors** are put into action with the **start** operator, which takes an **actor** and returns the "actor instance id", a unique number for each active **actor**. An **actor** can deactivate itself, or can be gunned down by the **script** or another **actor**, this is done with the **stop** operator which accepts an "actor instance id". **Run** is a combination of **start** and **stop**, it starts an **actor** with a predetermined stop cue. This is the definition of the operator "spin-cube-actor" used in the **script** "spinning-cubes":

```
(defop spin-cube-actor
  (param: color)

  (actor (local: (angle 0)
    (d-angle (quo 3 runtime))
    (my-cube (recolor color cube)))

    (see (rotate angle y-axis my-cube))

    (define angle
      (plus angle d-angle))))
```

It expects one parameter, a **color**, and returns an **actor** object.

The **actor** itself has three local variables, each of which is assigned an initial value in this case: "angle" is the current angle of rotation for this **actors** cube, "d-angle" is the incremental velocity of the angle, "my-cube" is a recolored version of ASAS's predefined "cube" **solid**. Each frame the **actor** constructs the rotated version of "my-cube" and passes it to the display, the current "angle" is updated for the next frame.

#### Animated Numbers

In the last example the symbol "angle" took on a series of numeric values, frame by frame, forming an arithmetic series. But for more complex time behavior (quadratic or cubic curves) the inline code to handle and update all those linear difference terms becomes a burden. To avoid this, ASAS supports an animated numeric object called a **newton** (as in Newtonian mechanics). **Newtons** can be used any place a number would be used, such as a coordinate in a **vector** or the angle parameter for **rotate**. Between frames however, **newtons** are automatically updated to the next value in their predefined sequence. The **newton** data structure holds its future as a chain of piecewise cubic curves with selectable degree of continuity at the joints.

A **newton** can be specified in terms of position, velocity, acceleration and delta acceleration ("jerk" or "jerkiness") when those values are known. But more typically **newtons** are defined with utilities which produce curves with certain properties. Animators are familiar with terms like "slow in" or "slow out" meaning that an action should start (or end) with zero velocity (first derivative). The five most common curves (or pieces of curves) used in ASAS are: **hold**, **linear**, **slowin**, **slowout** and **slowio**. **Hold** accepts a value and a length of time, each of the others takes a starting and ending value and a time. **Slowio** (slow in and slow out) has zero derivatives at both ends. When none of the standard curves are appropriate an interpolating cubic spline fit is used.

#### Actors and Behavior Simulation

Some animation is made to match a preconceived image, especially in commercial production. Other times, animation is produced as an experiment, the answer to "what would happen if ...". In the second type, which might be called "behavior simulation", the animator sets up a little world by defining the rules of behavior and selecting the cast of characters. When the behavior simulation is run we obtain images of what went on in the little world.

A classic example of this sort of thing is to try to build a computer graphic simulation of a flock of birds. We must define the behavior of a single bird so that when a lot of instances of the bird are simulated, they flock convincingly. The flock seems to be following a leader, but each time they turn, a new bird becomes the leader. The flock changes direction like a single unit, yet it is just an assembly of individuals. The flock is a dense cluster, but the birds do not often collide.

ASAS **actors** provide a convenient way of implementing such behavior simulations. As mentioned before, one of the features of **actors** is the way they promote "separation of powers", independent modules of code which do not interfere with each other. This allows an **actor** to take the part of one character in the simulation. If the same sort of character occurs many times in the simulation (e.g. many copies of BIRD) we can use

independent "instances" of a given "class" of **actors**.

The other key feature of **actors** which makes them suitable for behavior simulation is the ability to pass **messages**. Clearly the birds in the flock are exchanging information, through the action of light and sound on the bird's senses each one is aware of where the others are and where they are going (in an intrinsically depth sorted order!) In an **actor** simulation of the flock we would not go to the extent of modeling light and sound, but we could realistically have each bird broadcasting to everyone the message "I am here (x y z) and I'm heading (dx dy dz)". In that implementation, each bird would have to put the others in order of importance, probably using a "hidden bird algorithm".

#### Message Passing

ASAS **messages** are handled by two special operators: **send** and **receive**. **Send** composes messages and posts them at the recipient's mailbox. **Receive** reads each message in the mailbox, responding to each in a manner depending on the type of message. (ASAS **actors** act once each frame, not whenever a message comes, hence mail may pile up between frames so the mailbox is implemented as a FIFO queue.)

**Send** takes an address, a "message type" (optional), and any other specific message data (numbers, geometric objects, symbols). The address is either an **actor id** or a special symbol; "all" means send to all actors and "script" and "animate" can be used to send messages to the surrounding **script** or **animate** block. The "message type" is any symbol used to describe the type of message, it must match the message type in the recipient's **receive** construct. For example, these **sends** (1) tell "bouncer" to speedup by 10 percent and (2) announce to all birds where we are:

```
(send bouncer speedup 1.10)
```

```
(send all bird-state cur-position cur-velocity)
```

The **receive** construct has a body much like a **case** construct. Each message in the mailbox is examined in turn, the "message type" of each is compared with the type of the various clauses. If one of the clauses in the body of the **receive** matches the incoming message type, the body of the clause is evaluated in response to the message. Message type "any" in a clause will match any incoming message. The contents of a message (past the type) may be accessed by specifying parameter bindings for the clause type. For example, this **actor** knows how to receive only "speedup" and "slowdown" message:

```
(receive ((speedup f) (define speed
                      (times speed f)))
```

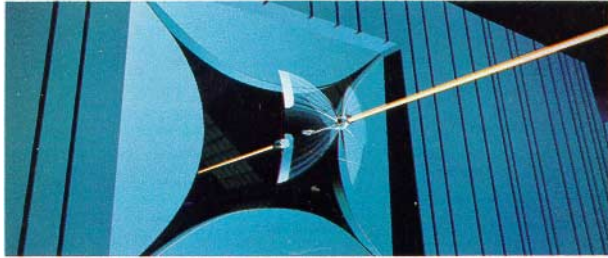
```
((slowdown f) (define speed
               (quo speed f)))
```

```
(any (print 'What?)))
```

The message passing mechanism described above is based on a more primitive operator called **in**. The **in** operator allows the evaluation of any expression "inside" the local variable space of an **actor**, thus allowing examining and setting the local variables of the **actor**. This is a useful but dangerous tool which should be used only in a well designed protocol.

### Commercial Production at III

ASAS frequently plays a central role in commercial animation production at III, although other techniques of animation control are used. Projects made with ASAS include, "MICROMA" animated logo, "LBS" animated logo, "NEWS CENTER 2" TV news show intro, two TV commercials for "TORNADO", various magazine ads, all of the theme animation for the III 1981 Sample Reel ("The Juggler"), about half of the special effects for the Ladd Company's feature motion picture "LOOKER", and all of the animation and still images III is making for the recently released Disney feature "TRON".



Copyright © 1982 by Walt Disney Productions and Information International, Inc.

Figure 3: Solar Sailer Escape Sequence from TRON

The Digital Scene Simulation Group usually works on a contract basis with clients such as film and video producers and advertising agencies. Some projects come to us carefully planned out by the client, while others come in a very vague form. If we do not get specific artistic directions (timings, storyboards and renderings) from the client, our Art Department creates these materials in consultation with the client. A team of at least three staff members (art director, designer/encoder, technical director) is formed to work on the job.

When the artistic concept is somewhat settled, work is started on its computer graphic realization. The first step is to create a geometrical model of the shapes of the objects to be used in the animation. Unless the geometry of the object is regular enough to allow it to be constructed under program control, the shape definition is done by hand in a laborious process similar to technical drafting we call "encoding". Often some mix of manual data entry and processing by various "geometrical tool" programs is used to obtain a finished object shape description.

As the objects are being finalized, the technical director begins to write the ASAS scripts and related programs. When the group is working many closely related scenes, for instance during production on a feature film, many of the "sets" or environments will be shared between several scenes. In such cases it becomes convenient to set up "libraries" of common ASAS function and object definitions. Contributions are made to these "public libraries" by all animators working on a project. Usually the motions in an animated sequence are so specifically planned out in advance that an outline of the script can be written before any of the objects are available for test pictures. At this stage the ASAS script is very abstract, references are made to symbolic constants whose values are not yet known. When object files are ready and the script is roughed out the "graphical debugging" begins.

Often design constraints are stated in such an indirect fashion ("have the camera pointing such that the logo is positioned here and oriented like in this sketch") that the only workable way to find the desired numerical parameters is experimentally with graphical feedback. After specific "key" frames have been composed, and the transitions between them defined, a motion test is made. Usually this test is made in either line ("vector") or low resolution shaded image mode. The symbolic nature of ASAS scripts make it easy to adjust the runtime of a sequence, making preliminary tests at 10:1 speed ratios allows faster turn around. Also the script can be simplified for these tests, by dropping out certain elements, replacing others with "stand-ins", all these changes can be made under the control of ASAS script flags. Often the motion test will reveal problems in the "feel" of the dynamics of the animation or unexpected behavior between the key frames.

Another pass or two is made to finalize the motion and then attention is shifted to the color, lighting, shading and other "photometric" parameters of the animation. Key frames are examined on a high resolution video display and color tests are made onto the type of color film which will be used for the final image. Often a parameter will be determined with a "wedge test", making a series of frames which differ only by a single parameter value (e.g. the amount of ambient light in the scene), with the gamut of values laid out, the final value can be easily be selected.

When all has been decided, a high resolution final filming is made. Typically when the result is screened, the client will find at least one reason to reject the work and the whole process goes back to the beginning.

### Conclusion

This paper has presented ASAS, a general purpose programming language which has been extended to include geometric objects and operators, parallel control structures and other features to make it useful for animated computer graphic applications. ASAS makes use of an abstract computing element called an **actor**, we have seen how **actors** promote modularity and how they can simulate a wide range of behavior by exchanging messages with each other. In three years of commercial use ASAS has proved itself a workable and practical tool. While the specific feature a user wants may not already be a part of the language, the extensibility of ASAS allows it to grow with its users. ASAS has expanded our "complexity barrier" another notch, allowing us to attempt work with more independently animated elements than before.

The author is not prepared to state that when the ULTIMATE computer animation system is built, it will be programming language based. But it is hard to visualize a system which allows arbitrary extensions into unexpected realms without being fully programmable. However, programming and making aesthetic judgements seem to be disjoint in most people's thinking processes. The user of a graphics programming system must always be on guard against compromising aesthetic judgements to simplify the programming! The solution used in our commercial work is to make the production a joint effort of several people, some responsible for artistic issues and others responsible for technical issues.

## References

- [1] Abelson, H. and diDessa, A. *Turtle Geometry*, MIT Press (Series in Artificial Intelligence), Cambridge, MA, 1981.
- [2] Austin, H., "The LOGO Primer", MIT A.I. Lab. Logo Working Paper 19.
- [3] Church, A "The Calculi of Lambda Conversions", *Annals of Mathematical Studies* 6, Princeton University Press 1941, Reprinted by Klaus Reprint Co., 1965.
- [4] Clark, J., "Hierarchical Geometric Models for Visible Surface Algorithms", *CACM*, October 1976.
- [5] Dahl, Myrhaug, and Nygaard *The SIMULA 67 Common Base Language*, Norwegian Computing Centre, Oslo, 1968.
- [6] DeFanti, T. "The Digital Component of the Circle Graphics Habitat", *Proceedings NCC 1976*.
- [7] Dijkstra, E.W. "Notes on Structured Programming", August 1969
- [8] Eastman, C. and Henrion, M. "GLIDE: A Language for Design Information Systems", *SIGGRAPH '77 Proceedings*, July 1977, San Jose, CA.
- [9] Futrelle, R. P. and Barta, G. "Towards the Design of an Intrinsically Graphical Language", *SIGGRAPH '78 Proceedings*, August 1978, Atlanta, GA.
- [10] Goates, G., Griss, M. and Herron, G. "PICTUREBALM: A Lisp-based Graphics Language System with Flexible Syntax and Hierarchical Data Structure", *SIGGRAPH '80 Proceedings*, July 1980, Seattle, WA.
- [11] Goldberg, A. and Kay, A. *SMALLTALK-72 Instruction Manual* Learning Research Group, Xerox Palo Alto Research, March 1976.
- [12] Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73", *Proc. of ACM SIGPLAN-SIGACT Conf.*, Palo Alto, CA, January 1975.
- [13] Hewitt, C. and Smith, B. "Towards a Programming Apprentice", MIT AI Lab Working Paper 90, January 1975.
- [14] Hewitt, C. and Atkinson, R., "Parallelism and Synchronization in Actor System", *ACM Symposium on Principles of Programming Languages* 4, January 1977, L. A. CA.
- [15] Jones, B. "An extended ALGOL-60 for Shaded Computer Graphics", *ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages*, April 1976.
- [16] Kahn, K. "An Actor-Based Computer Animation Language", *Proc. of the ACM-SIGGRAPH Workshop on User-Oriented Design of Computer Graphics Systems*, Pittsburg, PA, October 1976.
- [17] Kahn, K., "A Computational Theory Of Animation", MIT A.I. Lab. Working Paper 145, April 1977.
- [18] Larkin, F. "Computing with Text-Graphic Forms", *Conference Record of the 1980 LISP Conference*, August 1980, Stanford University.
- [19] Larkin, F. "A Structure from Manipulation for Text-Graphic Objects", *SIGGRAPH '80 Proceedings*, July 1980, Seattle, WA.
- [20] Moon, D. *MacLisp Reference Manual, Revision 0*, MIT Project MAC, December 1975.
- [21] Newman, W. and Sproull, R. *Principles of Interactive Computer Graphics*, McGraw-Hill, 1973 and 1979.
- [22] Pfister, G. "A High Level Language Extension for Creating and Controlling Dynamic Pictures", *ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages*, April 1976.
- [23] Preissler, M. "Multi-processed Music Synthesis", BS Thesis MIT EECS Department, May 1976.
- [24] Reynolds, C. "A Multiprocessing Approach to Computer Animation", SB thesis, MIT EECS Department, August 1975.
- [25] Reynolds, C. "Computer Animation in the World of Actors and Scripts", SM thesis, MIT (Architecture Machine Group), May 1978
- [26] Reynolds, C. *Actor / Scriptor Animation System User's Manual 3.0, (ASASUM 3)*, Information International Inc., in preparation.
- [27] Smoliar, S. "A Parallel Processing Model of Musical Structures" MIT AI Lab Technical Report 242, September 1971.
- [28] Sussman, G. and Steele, G. "SCHEME - An Interpreter for Extended Lambda Calculus", MIT AI Lab Memo 349, December 1975.
- [29] Teitelman, W. *InterLISP Reference Manual*, Xerox Palo Alto Research Center, 1978.
- [30] Winograd, T. *Understanding Natural Language*, Academic Press, 1974.
- [31] Winston, P. and Horn, B. *Lisp*, Addison Wesley, 1981.
- [32] Wirth, N. "MODULA: a Language for Modular Multi-programming", *Software, Practice and Experience* 7,1; 1977 pp. 3-35.

## Note

This entire paper is an example of computer graphics. All pictures were produced with Digital Scene Simulation, and directly (digitally) converted into four color halftones in the Infocolor format. The text was edited and composed on TECS, and assembled with a Page Makeup System. Camera ready, full page art (including typesetting and halftone generation) was produced with a COMp80/2 Pagesetter. All of these systems are products of Information International Inc.