AL: A Language for Procedural Modeling and Animation

Stephen F. May [*] Wayne E. Carlson [*] Flip Phillips [†] Ferdi Scheepers [‡]

[*] Advanced Computing Center for the Arts and Design, The Ohio State University
[†] The Vision Laboratory, The Ohio State University
[‡] Satellite Applications Centre, CSIR, South Africa

Technical Report OSU-ACCAD-12/96-TR5
December 1996

Abstract

We present a high-level programming language for the specification of procedural models. AL — an acronym for *Animation Language* — provides specialized programming constructs, data types, and operations for modeling, animation and rendering. In addition, it introduces a new mechanism for interaction with procedural models called *articulation functions*. AL extends and generalizes the mechanisms and features of previous animation and modeling languages. The use of AL in widely varying procedural applications is illustrated.

1 INTRODUCTION

The use of procedural techniques in computer graphics is widespread. Procedural techniques were born from the necessity to describe complex shapes or animation of phenomena too difficult to specify explicitly.

Traditionally when we think of procedural computer graphics, we think of techniques such as fractals for modeling terrain, L-systems for modeling plants and trees, rigid and elastic body dynamics for modeling hard and soft collisions between objects, and particle systems for modeling water, fire, and smoke.

However, as computer graphics models of people, animals, and everyday objects grow more sophisticated and complex, procedural techniques will play a larger and larger role in the specification of these models. Objects that we currently don't associate as being procedurally defined will be. Consider a model for a book containing hundreds of pages with text and imagery that need to be realistically flipped by a character or by the wind from an open window. Consider a model for a car which automatically follows the road and drops into potholes, a running engine under the hood, the ability to crumple and possibly explode as the result of high speed impact. Human models with detailed muscles, fatty tissue, skin, hair, clothing, and accessories will contain procedural components by necessity.

Models with a procedural definition can be more "intelligent" than the geometric models sold commercially today. They can encapsulate primary and secondary animation, modeling changes, rendering, cameras, visual special effects, sound effects, and interfaces for interactive manipulation. *Encapsulated models*[1] will provide abstraction and high-level control which will aid animators with the more tedious aspects of modeling complex objects and allow them to focus on the primary action, aesthetics, and storytelling. Not all animation is feature film, state of the art, hero character type animation. Increasingly, computer animation will be used for accident reconstruction, on-line manuals, product illustration, etc. In these instances, quick turn-around time will be the driving force for sophisticated, off-the-shelf models.

In the past, procedural techniques have usually been disassociated from interactive, visually driven approaches because of their algorithmic basis. However, more and more these capabilities are now appearing in highly interactive, commercial systems.

Some of the more established procedural techniques, like particle systems, are available in most commercial systems. The interface to a procedural effect in most software packages has a control panel with sliders, text fields, and other user interface components to control parameters of the individual special effect. Depending on the complexity of the effect, the user can modify these parameters and view the results interactively. Some software packages have specific components that are programmable (e.g., particle system where external forces can be programmatically specified or an L-system where arbitrary "growth" rules can be given).

"Plug-ins" are independent software modules that can add customized functionality to a larger system. A geometric modeling system which supports plug-ins, for example, would provide an API to the data structures and functions used to implement the modeling system. If the user needs an application specific or totally new modeling tool, the user can develop or purchase a plug-in using the provided API independently of the developers of the modeling system. In this way, the large modeling system can be quickly and cost-effectively customized to suit the user's needs.

Several animation systems have a feature called "expressions." Expressions allow animation tracks to be linked together by arbitrary mathematical expressions. These expressions are a powerful method for specifying simple mathematical relationships between object parameters.

Visual programming systems [Hae88] use dataflow visualization networks to specify scenes. Procedural elements can be constructed interactively without writing programs. However, it can be difficult to implement more complex forms of procedural animation, such as behavioral simulation, using the dataflow paradigm. The dataflow paradigm is not well suited for algorithms which require non-trivial data structures or communication between components (e.g., birds flying in a flock).

Programming languages, for better or worse, remain the most expressive and general means for the specification of procedural models. Programming languages provide a general way to describe procedural models of all types. It should also be noted that a visual programming system can be implemented on top of a programming language paradigm although the reverse does not seem as plausible. Further, non-procedural models, models where an explicit specification of geometry and motion is most convenient, are a subset of procedural models.

It is our underlying tenet in the development of AL that the data structure for tomorrow's models must be a procedural specification. The procedural specification should be in the form of a programming language and it is essential that this language provide general mechanisms for interaction.

[*] ACCAD, 1224 Kinnear Road, Columbus, OH 43212, [smay | waynec]@cgrg.ohio-state.edu

[1] The concept of encapsulated models will be described in a later paper.

Several language based systems have included mechanisms for interaction with the parameters of procedurally specified models [HS85, ROL90, Joh95].The AL language and run-time system[2] builds upon those languages and extends the set of mechanisms. In addition to the above systems, the AL language is influenced primarily by Reynolds Lisp-based language [Rey82], and the RenderMan Interface [Pix89].

The development of the AL language and run-time system is ongoing. This paper presents the major concepts involved in the design of the language and provides examples of its use. It also describes the major software components and a few of the most significant software implementation considerations. Readers should refer to [May94] for a complete definition of the AL language and and a description of the run-time system.

2 LANGUAGE OVERVIEW

AL is a programming language for specifying three-dimensional, procedural models. Models are synthetic, time-dependent, geometrically and temporally continuous representations of "real world" objects.

Models can represent simple objects, like inanimate props (e.g., a chair, a table, a steaming cup of tea, a mirror), or complex objects, such as fully clothed human characters. Large models can represent entire scenes complete with a cast of characters and detailed set. Models can contain both static and dynamic parts; procedural and non-procedural components. We don't restrict models to those which have a visible geometric representation. For example, a model for a camera may provide a convenient metaphor for "real" studio cameras which fix the camera to the floor or a track and have controls to adjust the height, pan, and zoom — but the purpose of our synthetic camera is the specification of a particular viewpoint (a camera transformation) and does not otherwise require a renderable geometric component. (This is not to say that it wouldn't be useful to have an accompanying geometric representation for viewing the camera position, field of view, or other properties from other viewpoints on the set).

In our system, all models are described the same way — as procedures implemented in the AL language. When evaluated by the AL run-time system, models produce high-level display lists for interactive display devices and input for high-quality renderers including RenderMan Interface [Pix89] compliant renderers.

The AL language provides all the features of "real" programming languages — variables, procedures, recursion, and control flow. In addition, it provides data types, procedures, and special constructs that make it easy to construct and animate three-dimensional models.

Data types, such as 3D points and transformation matrices are built-in features of the language as are mathematical operators on those special types. Procedures for creating an array of geometric primitives and mechanisms for transforming and specifying material properties for those primitives are provided. Special syntactic forms exist to group geometric shapes into hierarchically organized models and to provide both high-level and low-level animation controls.

AL provides primitives mechanisms for procedural modeling, but it does not require that models be specified using any one particular technique; in fact, it is possible to build AL models which incorporate a variety of procedural techniques simultaneously. A model animated using inverse kinematics may have geometric elements specified using a grammar-based model and contain secondary motion determined by a particle system.

[2]Currently, the "run-time" system consists of a language interpreter and suite of interactive tools. See Section 5.3.

AL is designed as an extension to the general purpose programming language Scheme. The Scheme subset provides data types, arithmetic operators, variables, procedures, control-flow constructs, and file I/O found in conventional programming languages. In addition, we feel that Scheme has several characteristics which make it conducive as a base language for procedural computer graphics (See Section 2.4).

2.1 Modeling

Reeves, *et al*. [ROL90] identify three powerful, high-level capabilities provided by procedural languages for modeling: parameterization, replication, and precision. The following examples illustrate the application of these capabilities in AL models.

Parameterized models are models that are determined by variables (model parameters). That is, the shape, position, material properties, etc. are not fixed; they are computed by the model based on the values of the parameters. In this way, a single model can be used to produce many variations, or *instances*, of a model by varying the model parameters.

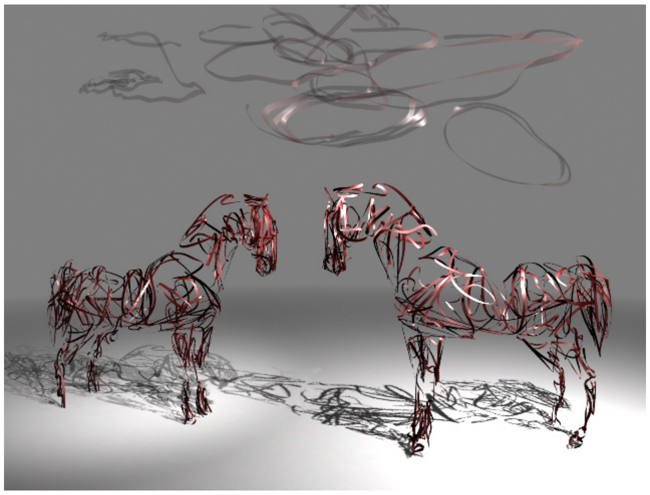


Figure 1: *Horse Play*, Charles Csuri, 1996.

Figure 1 shows the use of several instances of a parameterized AL model that creates "ribbon-like" objects. The primary parameter is a set of 3D vertices with each vertex having an associated orientation. (In the case of the horses shown, the vertices and orientations were determined from vertices and surface normals of a digitized polygonal data set.) A thin, flat, long "ribbon" (actually a b-spline patch mesh) of varying widths is generated as an extrusion that follows and is oriented by the vertex set. Secondary model parameters control the ordering and inclusion or exclusion of vertices, the range of acceptable ribbon widths, and the overall amount of randomness. The same ribbon model is used to make both the horses and the abstract cloud shapes.

A genetic approach was used to "evolve" the abstract, parameterized architectural models shown in Figure 2 [Blo95, BM95]. The architectural forms are modeled using constructive solid geometry and simple shapes including irregular cubes modeled with bicubic surface patches. Collectively, the generalized model parameters constitute a genotype which is iteratively mutated to produce successive generations of phenotypes (individual buildings). At each generation, the user provides the fitness function by selecting the most visually pleasing phenotype — that phenotype determines the genotype for the next generation. The process continues until a satisfactory form has been found.

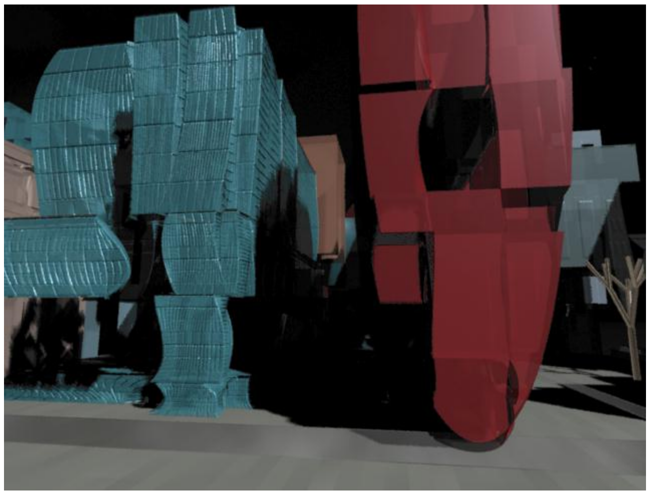


Figure 2: Genetically-based architectural models in a frame from *Tectonic Evolution* [BM95].

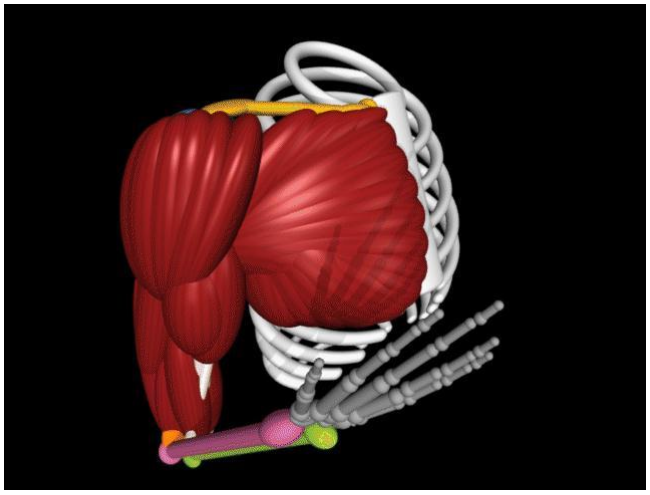


Figure 3: Anatomically-based muscle model of the upper arm.

Replication of model instances can be used to create more complex models. In Figure 3, instances of several muscle models are replicated to form a detailed model of the musculature of the upper portion of the human arm [Sch96, SPMC96]. Using the mathematical operators built into AL, the muscles can be attached precisely to the tendons and underlying skeleton.

Replication was used extensively in Figure 4 to produce a mechanically accurate model of an upright piano. The piano keys and the physical mechanisms for each key *inside* the piano including levers, hammers, and strings are replicated. With AL, it was possible to align the keys and internal mechanisms, even when in motion, with the mathematical accuracy required to produce a realistic piano model.

2.2 Animation

Both procedural and interactive animation capabilities are provided in AL. By combining the two techniques, complex motion and high-level control can be achieved.

Articulation functions, a generalization of *articulation variables* [ROL90], are used to interactively modify functions within AL models. Articulation functions include time-dependent channels or tracks which can be used to interactively animate parameters and variables within the model. These values can be used as high-level controls for the computation of more complex motion.

The muscle model of the upper arm shown in Figure 3 is built upon an articulated skeletal structure. The animator can animate the elbow using a single, articulation function which specifies the angle between the forearm and upper arm. When the elbow is animated, tendons move and muscles bulge and stretch. The individual joints of the fingers of the skeletal model can be animated using articulation functions providing low-level, detailed control. At the same time, a single high-level articulation function, named "clench," can be animated which results in all joints of all of the fingers being clenched into a fist position or releasing into an open palm position.

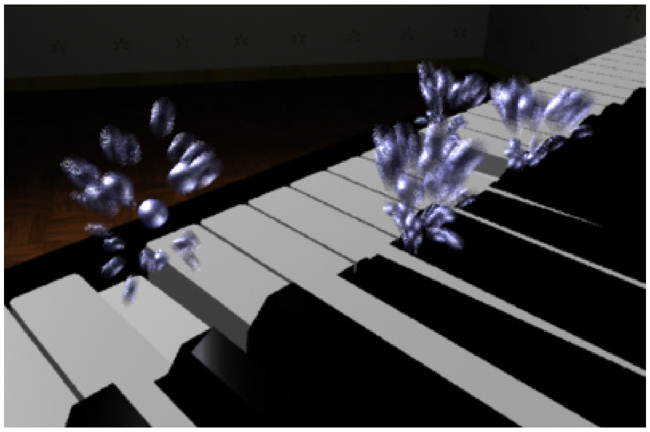


Figure 4: A frame from *Butterflies in the Rain* [FBM96]

The animation of the piano in *Butterflies in the Rain* was controlled by a MIDI-file representation of a musical score. The AL model for the piano responds to the musical events (note on, note off, velocity, etc.) to produce the motion of the keys at the correct time. In the animation, each key is struck by a splash of water spit by a talented fish in a bowl resting on top of the piano. The fish model is also controlled by the MIDI events to ensure that the water reaches the correct key at the instant when the corresponding note in the musical score must be played. A specialized particle system is used to animate the water drops and generate a splash when the drops reach the keyboard.

2.3 Rendering

The AL language and run-time system provide a flexible rendering architecture which permits both high-quality and interactive rendering to an arbitrary number of rendering devices. New rendering devices can be added easily by providing new *render packages* which are object-oriented mechanisms that bind primitives in the AL language to device-dependent rendering commands.

In practice, high-quality rendering is generally performed using the RenderMan compliant renderers *PhotoRealistic RenderMan* [Pix96] and *BMRT* [Gri96]. AL provides complete access to all features of these renderers and automates many of the more difficult specifications including: shadow map generation, motion blur, and user-defined shaders.

2.4 Scheme as a Base Language

AL is an extension of the general purpose, programming language Scheme [CR+91, Dyb96]. Thus, we call Scheme the *base language* for AL. Scheme is a lexically scoped, tail-recursive descendent of Lisp. There are two primary advantages to using a pre-existing, general purpose, programming language as a base for a "new" language.

1. Pre-existing languages have pre-existing specifications, books, compilers, interpreters, and support libraries.

2. A general purpose language will be flexible enough to solve a wide range of computer graphics problems. We want a language which not only can describe procedural models, but also perform batch rendering, image post processing, input and output to and from MIDI and other devices, and create non-trivial user interfaces.

Previous modeling and animation languages have been based on Algol-60 [Jon76], PL/1 [Pfi76], Lisp [Rey82], Fortran [OO81], C [GS88], TCL [Joh95], specialized languages [HS85, ROL90], and others. Theoretically speaking, all non-trivial programming languages are equivalent, but in practice, each language has various strengths and weakness and there were several properties that resulted in the choice of Scheme as the base language for AL.

Ease of use. AL should be as easy to use as possible. Programming is difficult enough and we want to encourage programmers and non-programmers to use AL. Scheme has a simple syntax and a small core of semantic concepts. (Scheme is frequently used as the language for introductory programming courses.) Scheme is generally interpreted rather than compiled. This reduces turnaround time and encourages experimentation. Scheme supports a rich set of high-level data types commonly used in computer graphics including strings, lists and vectors. Dynamic memory allocation and deallocation is handled automatically in Scheme. Functions are treated as first-class objects.

Extensible syntax. In AL, we want the visual structure of the program to reflect the logical structure of procedural models. So it is important to be able to extend the syntax of the language to accommodate new constructs for computer graphics. Scheme has an informally standardized method for specifying new syntax. Preprocessors can be used with other languages, but this can lead to difficulties in reporting syntax errors and in debugging.

Program vs. Data. In Scheme, programs and data share a common printed form. Keywords and variables correspond to symbols. Program constructs (structure syntactic forms) correspond to lists. This is an important concept in AL — we treat AL, a procedural language, as a data format. We manipulate functions in AL in much the same way that current systems manipulate object databases. In particular, this feature of Scheme enables us to implement articulation functions very naturally — we can store functional definitions in AL data files (See Section 5.3).

Performance. Performance is an important issue in interactive computer graphics. Scheme has a reputation for being slow, but Scheme compilers now exist which compare favorably with optimized C and Fortran compilers.

Formal Specification. Scheme is one of the few, "real" languages which has both formally specified syntax and formally specified semantics[CR+91]. Therefore a complete formal specification of AL would only require formal specification of the extensions.

3 GRAPHICS MATH DATA TYPES

Scheme provides numerous data types including lists, vectors (heterogeneous arrays), symbols, strings, characters and numbers (natural, real, rational, and complex). AL extends that set by adding types for representing points and transformation matrices. Standard math operators are overloaded providing natural and concise operations such as vector-scalar multiplication. (See Table 1.)

`(+` *vec* … `)`	Vector addition.
`(+` *mat* … `)`	Matrix addition.
`(-` *vec* … `)`	Vector subtraction.
`(-` *mat* … `)`	Matrix subtraction.
`(*` *mat* … `)`	Matrix multiplication.
`(*` *vec real* … `)`	Vector scalar multiplication.
`(*` *mat real* … `)`	Matrix scalar multiplication.
`(*` *vec mat* … `)`	Row matrix multiplication.
`(*` *mat vec* … `)`	Matrix column multiplication.
`(/` *vec real* … `)`	Vector scalar division.
`(/` *mat real* … `)`	Matrix scalar division.

Table 1: Overloaded math operators.

Additional functions that are convenient for "graphics math" that can be found in libraries for other languages [PSM95] are supplied. These functions include conversion between various spatial and color coordinate systems, interpolation functions including splines and surface patches, and the complete set of noise functions described by Peachey in [EMP+94].

Most procedures are overloaded to work on all math types — e.g., interpolation functions work just as well on 3D points as they do on scalars.

3.1 Graphics Vectors

Scheme vectors are heterogeneous arrays of arbitrary length. In computer graphics, we frequently deal with a more restricted form for representing points in three-dimensional space, surface normals, and colors (e.g., red-green-blue representation). The *vec2*, *vec3*, and *vec4* "graphics vectors" data types provide an efficient and convenient representation for points in R^2, R^3, and R^4 respectively.

Graphics vectors have a printed (literal) form that consists of the elements of the vectors (2, 3, or 4 reals) preceded by the prefix `#<` and followed by `>`. For example, a vec3 containing the elements 1, 0, and -2.5 would be written `#<1 0 -2.5>`.

Table 2 shows many of the procedures which operate on the vec3 type including a constructor, a predicate for testing whether or not an arbitrary Scheme object is a vec3, procedures for retrieving and setting components, and mathematical operations. Many of the procedures shown are overloaded for use with the vec2 and vec4 types.

Three commonly used constants `x-axis`, `y-axis`, and `z-axis` are defined in the global environment and evaluate to `#<1 0 0>`, `#<0 1 0>`, and `#<0 0 1>`.

3.2 Transformation Matrices

The *mat3* and *mat4* data types are provided in AL as an efficient and convenient representation of 3×3 and 4×4 homogeneous transformation matrices.

Transformation matrices have the same printed (literal) form as the graphics vector types (vec[2,3,4]) except that either 9 or 16 elements of the 3×3 or 4×4 matrix respectively are specified in row-major order between the prefix `#<` and the suffix `>`.

Table 3 shows a representative set of the procedures which operate on the mat4 type. Several of these are overloaded for the mat3 type as well.

`(vec3` *x y z*`)`
  Creates a new vec3 containing elements *x*, *y*, *z*.

`(vec3?` *obj*`)`
  Returns `#t` if *obj* is a vec3, `#f` otherwise.

`(xcomp` *vec*`)`
  Returns the *x* component of *vec*.

`(set-xcomp!` *vec x*`)`
  Sets the *x* component of *vec*.

`(dot` vec_1 vec_2`)`
  Returns the dot product of vec_1 and vec_2.

`(cross` $vec3_1$ $vec3_2$`)`
  Returns the cross product of $vec3_1$ and $vec3_2$.

`(distance` vec_1 vec_2`)`
  Returns the Euclidean distance between the points vec_1 and vec_2.

`(norm` *vec*`)`
  Returns the vector length of *vec*, $||vec||$.

`(normalize` *vec*`)`
  Returns *vec* normalized to unit length.

Table 2: Representative procedures for the vec3 type.

`(mat4` $a_{0,0}$ $a_{0,1}$ $\ldots a_{3,3}$`)`
  Creates a new mat4 with real elements $a_{0,0}$ through $a_{3,3}$.

`(mat4?` *obj*`)`
  Returns `#t` if *obj* is a mat4, `#f` otherwise.

`(mat4-ref` *mat4 i j*`)`
  Returns the element in row *i* and column *j* of *mat4*.

`(mat4-set!` *mat4 i j a*`)`
  Sets element $mat4_{i,j} = a$.

`(transform-point` *vec mat*`)`
  Returns *vec* (a point) transformed by *mat*.

`(inverse` *mat*`)`
  Returns the inverse of *mat*.

`(transpose` *mat*`)`
  Returns the transpose of *mat*.

`(determinant` *mat*`)`
  Returns the determinant of *mat*.

`(mat4-identity)`
  Returns the $4x4$ identity matrix.

`(mat4-rotate` *angle vec3*`)`
  Returns the transformation matrix corresponding to a rotation of *angle* degrees about the specified axis *vec3*.

Table 3: Representative procedures for the mat4 type.

4 MODELING

AL provides several efficient, high-level procedures and special forms for the construction of hierarchical geometric models. These procedures and special forms can be classified into two distinct categories: *geometric primitives* and *geometric operations*. Both types operate within a global context defined by a structure called the *graphics state*.

The graphics state in AL is similar to those used in other graphics systems including PostScript [Ado94], OpenGL [NDW93], Open Inventor [SC92, Wer94] and the RenderMan Interface [Pix89]. The graphics state maintains information used to determine, among other things, how geometry is rendered. As the execution of an AL model proceeds, the graphics state is modified. At any specific point in time, the *current graphics state* specifies both a cumulative transformation (the result of an arbitrary number of scales, translations, rotations, etc.) and a table of attributes.

Geometric primitives, or *gprims* (pronounced “gee-prims”), specify simple, geometric shapes or forms including quadrics, polygons, and surface patches. When a gprim is rendered, it is transformed by the cumulative transformation contained in the current graphics state. In addition, any relevant attributes of the current graphics state (e.g., surface color, opacity, and shaders) are also applied to the gprim.

Geometric operations, or *gops* (pronounced “gee-ops”), very simply alter the current graphics state and thus implicitly modify gprims that are declared afterwards.

High-level AL gops support solids modeling (CSG) and general instancing of models.

The AL code fragment below illustrates the use of two gops, `color` and `translate`, and the `sphere` gprim. The result is a yellow sphere translated one unit along the x-axis.

```
(color 1 1 0)
(translate 1 0 0)
(sphere)
```

In the previous example, additional gprims specified after the sphere will also be translated and colored yellow. The effect of gops can be localized using *separators*[3]. In the example below, both the sphere and torus will be colored yellow; the torus will be positioned at $(1, 0, 0)$; and the sphere will be positioned at $(1, 1, 0)$. In other words, we have a simple, hierarchical model because the first translation affects both shapes while the second translation only affects the sphere.

```
(color 1 1 0)
(translate 1 0 0)
(separator
  (translate 0 1 0)
  (sphere))
(torus)
```

The simple example below illustrates instancing and high-level control through model parameterization. A “ball” model is defined which has two parameters: “ballcolor” and “squash.” If the squash parameter is less than 1, the ball is flattened. If the squash parameter is greater than 1, the ball is stretched vertically. This single parameter controls the x, y and z scaling of the sphere and the model maintains a mathematical relationship between the three to maintain the original volume of the sphere. The ball model is then instanced twice with varying color and deformation.

```
(define (ball ballcolor squash)
  (separator
    (color ballcolor)
    (scale (/ (sqrt squash)) squash (/ (sqrt squash)))
    (sphere)))
:
(ball #<1 0 0> 0.5)   ;; squashed
(translate 2.5 0 0)
(ball #<0 0 1> 1.5)   ;; stretched
```

[3]The term “separator” is borrowed from Inventor [SC92].

One particularly nice feature in AL is that the graphics state itself is programmable. New attributes of arbitrary type can be added by the user and maintained automatically by the AL run-time system. For example, in a dynamics simulation, it might be desirable for each geometric object have a "rigidity" attribute which defines how flexible or rigid a shape is under the application of external forces. By adding "rigidity" as an attribute of the graphics state, the attribute can be easily applied and isolated to entire sub-hierarchies of the scene using a new gop and the existing functionality of separators.

4.1 The World Model

A special model, the *world model*, represents the root of the model hierarchy for the entire scene (or film). The world model is analogous to the `main()` procedure in the C programming language. All of the code which generates the scene must be specified in the ⟨body⟩ of the `world` syntactic form.

`(world` ⟨body⟩`)` syntax

Declares the *world model*. The ⟨body⟩ is evaluated and also retained in unevaluated form for later re-evaluation. Gops and gprims are valid only inside ⟨body⟩ and functions called by ⟨body⟩.

4.2 Geometric Primitives

The geometric primitives (gprims) in AL can be used to create a wide variety of surface shapes. The primitives include quadrics (cones, spheres, hyperboloids, paraboloids, cylinders, disks, and tori), polygons (convex, concave, polygons with holes), surface patches (bicubic and NURBS), lines, and text. With a few exceptions, the gprims in AL are based directly on the primitives specified in the RenderMan Interface [Pix89]. Material properties (color, opacity, shading, etc.) and transformations for gprims are specified using gops (See Section 4.3).

Gprims, for the most part, are simple procedures. Some have no required arguments. All gprims have an optional list of parameters. When optional parameters are not specified, default values are used.

`(sphere` ⟨paramlist⟩`)` procedure

Draws sphere gprim. The sphere is a semi-circle defined in the *x-y* plane swept about the z-axis. Optional parameters (and defaults) are *radius* (1.0) *zmin* (-1.0), *zmax* (1.0), and *thetamax* (360). Zmin and zmax specify clipping planes along the z-axis. Thetamax specifies the sweep angle in degrees.

The following procedure call specifies a hemisphere of radius 2. Note that if zmin and zmax values are not set to 2, the sphere will be clipped at $z = -1$ and $z = 1$.

```
(sphere 'radius 2 'zmin -2 'zmax 2 'thetamax 180)
```

Renderer specific parameters (vertex normals, texture coordinates, etc.) can also be specified. Renderers which do not support these non-standard parameters will ignore them. RenderMan compliant renderers, for example, understand a parameter "Cs" which allows color to be assigned per vertex. The following draws a sphere with different colors interpolated across its surface[4].

```
(sphere 'Cs '(#<1 0 0> #<0 1 0> #<0 0 1> #<1 1 0>))
```

4.3 Geometric Operations

Geometric operations (gops) modify the graphics state. The graphics state is actually implemented using two independent stack data structures. One stack maintains the current transformation matrix (*matrix stack*). The other stack maintains an attribute table — some of the attributes are shown in Table 4. Accordingly, most gops divide into those which modify the transformation matrix (*transformation gops*) and those which modify the attribute table (*attribute gops*).

attribute	**default value**
color	`#<1 1 1>`
opacity	`#<1 1 1>`
surface shader	–
light source shaders	–
divisions	(10,10)
bicubic basis	*Bezier*
light shadows	`#f`
casts shadows	`#t`
hierarchy	"/world"

Table 4: Several attributes of graphics state.

Two gops `separator` and `xfm-separator` are used to group gprims and gops into logical components. Both types of separators may be nested to an arbitrary depth to create hierarchical components.

These special syntactic forms localize the effect of other gops to a particular body of expressions by pushing and popping one or both of the stacks in the graphics state. When we say that a stack is "pushed," we mean that a copy of the top-most item of the stack is placed on top. When we "pop" the stack, the top-most item is discarded which leaves the stack exactly as we left it before the previous push. During evaluation of an AL model, the *current transformation* is the top-most element of the transformation stack. Likewise, the *current attribute table* is the top-most element of the attribute stack. The *current graphics state* refers collectively to both the current transformation and the current attributes.

`(separator` ⟨body⟩`)` syntax

Pushes both the transformation and attribute stacks; evaluates the expressions in ⟨body⟩ in left to right order; and then pops the transformation and attribute stacks. Nothing outside of ⟨body⟩ will be affected by the gops specified within ⟨body⟩.

`(xfm-separator` ⟨body⟩`)` syntax

Pushes the transformation stack only; evaluates the expressions in ⟨body⟩ in left to right order; and then pops the transformation stack. Attribute gops are not affected by `xfm-separator`.

[4]Quadrics are specified as parametric surfaces of u and v. Conceptually, the parametric "vertices" of a sphere then are for $(u, v) = (0,0), (1,0), (0,1), (1,1)$ with u corresponding to longitude and v corresponding to latitude.

Transformations

The complete set of transformation gops is shown in Table 5. With the exception of the `transform` gop, all gops *premultiply* the current transformation matrix (CTM) with a new transformation matrix. In other words, the first transformation specified is the last one applied. To rotate and *then* translate a box, we specify the `translate` gop first *followed* by the `rotate` gop.

```
;; rotate then translate a box
(translate 1 0 0)
(rotate 45 y-axis)
(box)
```

Points are transformed by matrices as column vectors. I.e., $p' = Mp$, where p is the original point, M is the transformation matrix, and p' is the result of p transformed by M.

`(transform` *mat4*`)`	Replaces the CTM with *mat4*.
`(concat-transform` *mat4*`)`	Concatenates mat4 onto the CTM.
`(scale` *x y z*`)`	Concatenates a scale transformation onto the CTM.
`(rotate` *angle vec3*`)`	Concatenates a rotation transformation of *angle* degrees about the axis *vec3* onto the CTM.
`(translate` *x y z*`)`	Concatenates a translation transformation onto the CTM.
`(skew` *angle* $vec3_1$ $vec3_2$`)`	Concatenates a skew transformation of *angle* degrees between vectors $vec3_1$ and $vec3_2$ onto the CTM.

Table 5: Transformation gops.

Attributes

Unlike transformation gops which *modify* the current transformation matrix, attribute gops (with the exception of light sources) *replace* the corresponding value in the current attribute table (See Table 4). In general, attribute gops affect the way gprims are drawn. This includes material properties (color, opacity, shading), basis functions for bicubic surface patches, trim curves, light sources, and other rendering controls. This section will describe a few of the more commonly used attribute gops.

`(color` *r g b*`)` procedure
`(opacity` *r g b*`)` procedure

> Specify the current color and opacity of the graphics state. Colors are expressed as red-green-blue triples where each component has a value between 0 and 1. A vec3 can be passed as the only argument to these functions instead of specifying the components individually.

AL supports all shader types defined by the RenderMan Interface: surface, light source, displacement, atmosphere, interior, exterior, and deformation. Shaders are transformed by the current transformation matrix. This is convenient for modifying the appearance of shaders and incorporating them into hierarchical models.

`(surface` *name* ⟨paramlist⟩`)` procedure
`(displacement` *name* ⟨paramlist⟩`)` procedure

> Specifies the current surface and displacement shader. In each case, the name of the shader is specified as a string. Optional parameters can be specified for the shaders in parameter list form.

Light source shaders are different from other shaders in that a list of light source shaders is maintained in the graphics state. Each light gop therefore adds a new light on to the list. Light sources can be localized (applied only to certain objects) using separators or by turning them off and on using the `illuminate` gop.

`(light` *name* ⟨paramlist⟩`)` procedure

> Adds a new light source shader to the graphics state. The name of the shader is specified as a string. Optional parameters can be specified for the shaders in parameter list form.

`(illuminate` *lighthandle bool*`)` procedure

> Turns light source shaders on and off. Lights are identified by the *lighthandle* which is an integer returned by the light source gops.

4.4 Cameras

Cameras specify the camera (view) transformation for the entire scene at the time when the scene is rendered. An arbitrary number of cameras can be specified in the world model. Cameras are transformed by gops and can be placed anywhere in a model, so they can be attached to objects or parts of a hierarchy. Cameras always view the entire scene regardless of where they are placed in the world model. AL provides two classes of cameras: "perspective" and "orthographic." New camera classes (i.e., camera shaders) can be defined by the user.

`(camera` *name type* ⟨paramlist⟩`)` procedure

> Specifies a new camera which will be identified by *name* (a string). Type specifies the camera class; either "perspective", "orthographic", or some user defined class.

5 ANIMATION

Both procedural and interactive mechanisms can be combined and used in AL models for motion specification.

5.1 Time

Central to the idea of animated models in AL is the notion of *time*. Time is a unitless measure and can arbitrarily represent video frames, film frames, MIDI ticks, seconds, or any other convenient measure for the animator. The "current time" is accessible within a model through a read-only, global variable, called appropriately, `time`. An animated model in AL is defined as a model which contains expressions directly or indirectly dependent upon `time`.

`time` variable

Stores a numerical value corresponding the the "current time." The value may be an integer or real. This variable is read-only and should not be modified using `set!`.

Let's consider a simple example of a sphere model which "bounces" up and down. In the code fragment below `bounce` is a pre-defined function which takes time and the bounce frequency (the duration of each bounce in frames) as parameters and the "height" of a perfectly elastic object bouncing under some pseudo-gravitational force (this function could be approximated mathematically as $bounce(time, freq) = |\sin(\pi \times time \times (1/freq))|$).

```
(define (bounce t f) (abs (sin (* PI t (/ f)))))

(world
  (translate 0 (bounce time 30) 0)
  (sphere))
```

Note that in the model specified, the motion is entirely procedural and no interaction is required. In general, most models incorporate a combination of procedurally driven and interactively specified animation. This combination yields complex motion with simple controls. The mechanism in AL for interactively specified motion is the *articulation function*.

5.2 Articulation Functions

Articulation functions are a generalization of the concept of *articulation variables* [ROL90]. Articulation variables, *avars* for short, are akin to what interactive animation systems call "tracks" or "channels" [Gom84]. Avars were introduced in Pixar's Menv system as "the link between modeling and animation [ROL90]." In ML (the modeling language used in the Menv system), avars look like regular scalar variables and as such can be used as arguments to functions, as conditions in if/then statements, etc. But unlike typical variables, the value is externally defined (from the program) and time-dependent. The value of an avar at the current time is determined by interpolating a spline through a set of interactively (or procedurally) specified key frames.

Because avars are defined externally, they provide a logical separation between a model and a specific instance of animation. In other words, there can be more than one set of avars associated with a single model — each set represents a unique animation of that model. Further, external definition of avars provides a mechanism for interaction — interactive programs (e.g., a spline editor) can modify avar definitions within a model. When modified externally, the model interpreter can re-execute the model to update an interactive display of the model.

In AL, the notion of articulation variables has been generalized to the concept of *articulation functions*. Articulation functions, *afuncs* for short, look and act like *functions* in an AL program and they can be used to solve a wider range of animation problems and serve as a broader mechanism for interaction.

Articulation functions are a significantly more general and powerful mechanism than avars. Afuncs can have arguments. Afuncs can return arbitrary values including points, lists of points, binary values, control meshes, images, MIDI events, etc. Afuncs may not return a value at all but rather modify the model via side effects (e.g., modification of global variables). Afuncs are composable. That is, afuncs can be used directly as inputs to other afuncs.

In the current implementation, only one type of afunc is supported — *channels*. Channels are modeled after avars[5] and in fact fulfill a similar functionality to the "channels" or "tracks" commonly found in interactive systems — hence the name.

[5]In the past, we have actually used the misnomer "avar" to refer to this particular type of afunc. As new types of afuncs are developed, the term avar will become increasingly confusing and inappropriate so we will use the term "channel" or "channel afunc" instead in this paper and in future work.

The concept of a channel afunc in AL is very broad. Channels describe an entire class of functions. A channel is any function which is time-dependent, returns a scalar, and is defined by a set of key frames. The function definition (i.e., the code that implements the function) is arbitrary. Channel afuncs may choose arbitrary methods for interpolation for example. Some channel afuncs may impose constraints. Different channel afuncs may allow different types of optional parameters to be passed to them when they are declared or when they are used.

Conceptually, the `model` syntax defines a new, named subhierarchy and declares zero or more articulation functions lexically scoped to that subhierarchy. Models can be nested to construct hierarchies of afuncs. Let's consider a re-implementation of the squash and stretch ball described in Section 4.

```
(world
  (model "ball" (squash height)
    (translate 0 (height) 0)
    (scale (/ (sqrt (squash))) (squash) (/ (sqrt (squash))))
    (sphere)))
```

Two afuncs are declared locally to the "ball" model, "squash" and "height." By default, these are channel afuncs with a default value of zero when no keys have been specified. Note that they are used as functions in the body of the model which is why they are enclosed in parentheses. An equivalent declaration of the model and afuncs where the type of afunc is explicitly specified is shown below.

```
⋮
  (model "ball" ((squash "channel") (height "channel"))
⋮
```

In the latter form, parameters to articulation functions can be specified. The potential problem of division by zero in the ball model can be avoided by specifying an optional "default" value parameter which is used only when no keys have been specified.

```
⋮
  (model "ball" ((squash "channel" 'default 1) ...
⋮
```

`(model` *name* ⟨afuncs⟩ ⟨body⟩`)` syntax

Creates a new subhierarchy in the current model and declares zero or more afuncs local to ⟨body⟩. The graphics state is pushed before evaluating ⟨body⟩ and popped after evaluating ⟨body⟩.

⟨Afuncs⟩ should have the form

$((\langle \text{afunc}_1 \rangle \ \textit{type}_1 \ \langle \text{paramlist}_1 \rangle) \ \ldots)$

or

$(\langle \text{afunc}_1 \rangle \ \ldots)$,

In the latter, shorthand form, the afunc type defaults to "channel" and optional parameters cannot be specified.

One example of the convenience provided by articulation functions (versus articulation variables) is the ability of afuncs to accept arguments. Channel afuncs take an optional "time" argument. By default, a channel is sampled at the current value of global time (Section 5.1). If an argument is given to a channel, the channel will be sampled at the time specified by that value. Therefore `(height)` is equivalent to `(height time)`.

By sampling channels at different times, AL models can effectively "look" into the past or future. Animals and people tend to look ahead in anticipation of where they want to move. If we have channels which specify the position of the object over time, we can sample those channels at some value in the future (e.g., $time + 10$) to determine the heading (the direction that the object is looking). The "squash" channel of the ball model could be replaced with a value computed from the speed (change in value from the previous frame) of the "height" channel.

```
 (- (height) (height (- time 1)))
```

5.3 The Exposure Sheet

The *exposure sheet* or *x-sheet* is a database which binds a functional definition to each instance of each articulation function in the world model. The x-sheet structure mimics the world model hierarchy, but it is a separate entity. When the bindings for afuncs change, the world model is automatically updated. An interactive tool, such as a spline editor, can modify afunc definitions in the x-sheet and this implicitly modifies the world model. Interactive display devices respond to changes in the world model and redraw the scene. For moderately complex scenes, this modify-evaluate-render loop can be executed at interactive speeds.

The exposure sheet is stored in a separate file from the model program. Multiple exposure sheets (i.e., multiple instances of animation) can be stored and retrieved for a single model.

To reiterate, the x-sheet database actually contains the code which is bound to each articulation function. This is aided by Scheme's ability to deal with programs as data and vice-versa.

5.4 Implicit Model Specification

An important concept in the construction of AL models is the concept of implicit specification. AL models are specified as continuous, implicit functions of time and the x-sheet.

Consider the motion of a sphere moving on a plane between points A and B. An explicit specification would say something like "Start the sphere at point A. Move it to point B in 30 seconds." An implicit specification of the same motion would say "The position of the sphere at frame f is $f/30$ percent of the way between A and B."

The implicit model can make model specification more difficult, particularly for models involving techniques like forward simulation. However, it makes for a flexible system of modular objects. An implicitly specified model is independent of any previous or future evaluations of that same model. The state (shape, position, surface attributes) of a model can be determined for any arbitrary value of time. This is necessary because the run-time system determines when models will be temporarly sampled. The run-time system may sample less frequently to maintain interactive speeds on a workstation display or more frequently to render motion blur. All model procedures must be capable of producing the modeled object at whatever value of time the run-time system demands.

6 RENDERING

Rendering of the world model in AL is achieved through a mechanism called *render packages*. Render packages dynamically bind gops and gprims to rendering commands to specific display devices.

There is no underlying data structure or display created when an AL model is evaluated. Evaluation of the world model results in the output of rendering commands for exactly one specific display device. The problem with generating a common display list used by all devices is the so-called "lowest common denominator" problem. The entire scene could be broken into a polygonal display list, but some renderers may provide primitives for direct, hardware rendering of NURBS. The same holds for surface shading and lighting. Some renderers may support texture mapping. Some may even support real-time evaluation of procedurally defined shaders. This does not prohibit the creation and/or sharing of display lists by a render package however. For example, *ogre* [May94] is a render package that generates a shared, OpenGL display list used by an arbitrary number of viewing windows. Another render package could generate Open Inventor scene graphs [SC92].

The world model is rendered (and thus evaluated) in two ways. Whenever the model definition, global time, or the x-sheet changes, the world model is re-rendered which results in an update in the display of interactive devices. The world model can also be rendered explicitly for non-interactive (batch), high quality renderings using the `render` procedure. The parameter list of the `render` procedure supplies all rendering controls. Some parameters apply to all render packages (e.g., which camera to use, what frames to render) and others are render package specific.

`(render` ⟨paramlist⟩`)` procedure

> Renders the world model for one or more frames. The world model is evaluated one or more times per frame.

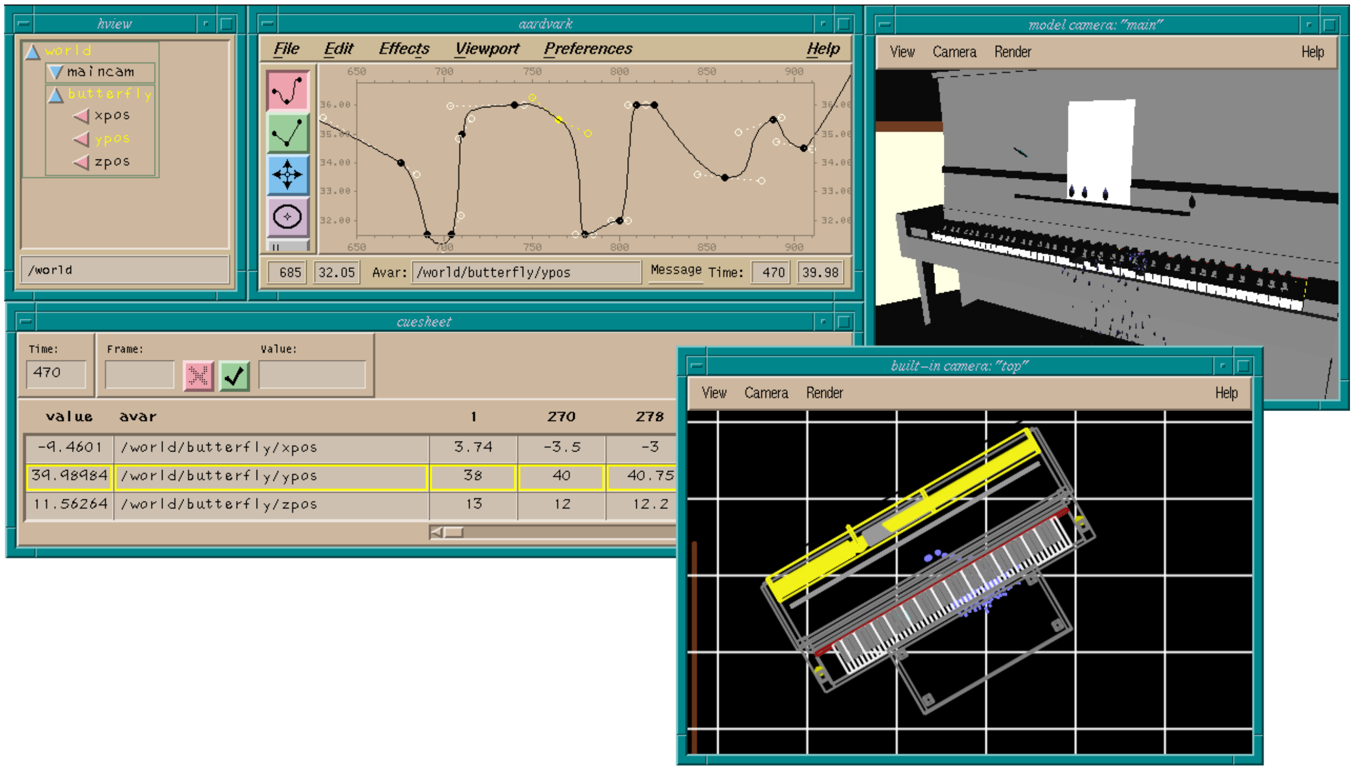
7 SOFTWARE OVERVIEW

The AL run-time system consists of the AL language interpreter and several tools for both interactive display of the world model and interactive manipulation of articulation functions. Figure 5 shows a typical workstation display with the interpreter and interactive tools.

The central piece of software is *ox*, the AL language interpreter. Ox can interpret AL commands typed at the keyboard or loaded from a file. Messages can be sent to ox from other UNIX processes for remote issuance of AL commands.

Ox is based on ELK, the Extension Language Kit by Oliver Laumann [LB94]. ELK provides a public domain, R4RS [CR$^+$91] compliant Scheme interpreter with numerous extensions including Scheme interfaces to the UNIX operating system, XLib, Xt, Motif and other useful libraries. The functionality provided by ELK can be extended through interpreted Scheme or compiled C/C++ code. Compiled object code can be dynamically linked into the interpreter at run-time; which makes it very convenient for users to enhance the interpreter with their own extensions. In addition to support for the AL language, the AL software provides general extensions of its own including Scheme interfaces to the OpenGL library, various image and object data formats, support routines for specific rendering software packages, and additional support for interface development.

The interactive tools are influenced by the tools of the Menv system and include *ogre*, *aardvark*, *cuesheet*, and *hview*. Some of the tools run directly as clients of the interpreter, while others communicate with the interpreter via message passing and data structures



685 32.05 Avar: /world/butterfly/ypos Message Time: 470 39.98

Time: 470 Frame: Value:

value	avar	1	270	278
-9.4601	/world/butterfly/xpos	3.74	-3.5	-3
39.98984	/world/butterfly/ypos	38	40	40.75
11.56264	/world/butterfly/zpos	13	12	12.2

Figure 5: AL run-time system, interactive tools. In clockwise order: hview (hierarchy browser), aardvark (afunc editor), two ogre windows with views of a piano model, and cuesheet (afunc editor).

maintained in shared memory. Each tool is independent of the other tools and may be stopped or started at any time. Multiple instances of the same tool may exist simultaneously.

Aardvark and cuesheet provide means for interactive modification of channel afuncs and global time. Both tools allow key frames to be added, modified and deleted. Aardvark provides a visual representation of channels as splines with time running along the horizontal axis and values running vertically. Cuesheet arranges the channel keys into a spreadsheet organization with a row for each channel and a column for each frame containing one or more channel keys. Modifications to channels in either of these tools results in evaluation of the world model and an updated rendering of the world in the interactive display tools.

Ogre is the primary interactive display tool. It is capable of rendering the world model in wireframe or shaded form and utilizes the local workstation hardware. The scene can be viewed from any camera defined in the world model and also from a number of auxiliary viewpoints provided within the ogre tool.

Hview displays the world model hierarchy and is used to specify which channels are currently being displayed and edited in aardvark and cuesheet.

8 CONCLUSION AND FUTURE WORK

The AL language and run-time system has proven itself to be a powerful and flexible system for implementing a variety of procedural modeling and animation techniques. It has been used for both production of completed animations and stills and as a testbed for new research. Articulation functions appear to be a powerful new mechanism for interaction with specifications of procedural models. Additional work is needed however to determine new types of articulation functions that are useful for constructing and manipulating models.

A number of extensions to the AL language and run-time system are currently in development. Research is focused around a more complete development of other classes of articulation functions and new interactive tools for their specification and manipulation. High-level language constructs for dynamic hierarchies (where subhierarchies move from one hierarchy to another), forward simulations, MIDI generation, and procedural image processing are being investigated. Models never run fast enough in an interactive environment, so we are also investigating ways to optimize evaluation of AL models and general language mechanisms for improving the speeds at which models can be manipulated interactively. Many of the performance issues cannot simply be solved with faster processors, since we will then just model more complex procedural entities. Flexible language constructs for multiple, dynamic levels of detail are one solution.

Procedural models will play an increasingly important role in computer generated imagery as the sophistication of models reaches increasingly higher plateaus. The most sophisticated of these models will not be stored and transferred in static databases or data structures — they will be specified as programs written in procedural languages. We are not suggesting that animators will write programs to create films, but animators will use procedural models; therefore, techniques for interactive manipulation and exploration of procedural models are necessary. Further, the technicians that create procedural models will need more powerful languages to construct models efficiently.

References

[Ado94] Adobe Systems Incorporated. *PostScript Language Reference Manual*, second edition, 1994.

[Blo95] Beth Blostein. Procedural generation of alternative formal and spatial configurations for use in architecture and design. Master's thesis, The Ohio State University, 1995.

[BM95] Beth Blostein and Terry Monnett. Tectonic evolution. SIGGRAPH 95 Conference Film Show, August 1995.

[CR$^+$91] William Clinger, Jonathan Rees, et al. The revised^4 report on the algorithmic language scheme. *LISP Pointers*, 4(3), 1991.

[Dyb96] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, second edition, 1996.

[EMP$^+$94] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6.

[FBM96] Mark Fontana, Kirk Bowers, and Steve May. Butterflies in the rain. Work in progress, 1996.

[Gom84] Julian E. Gomez. TWIXT: A 3D animation system. In K. Bo and H. A. Tucker, editors, *Eurographics '84*, pages 121–133. North-Holland, 1984.

[Gri96] Larry I. Gritz. Blue Moon Rendering Tools. Silver Spring, MD, 1990-1996.

[GS88] Mark Green and Hanqin Sun. A language and system for procedural modeling and motion. *IEEE Computer Graphics and Applications*, 8:52–64, November 1988.

[Hae88] Paul E. Haeberli. ConMan: A visual programming language for interactive graphics. In John Dill, editor, *Computer Graphics (SIGGRAPH 88 Conference Proceedings)*, volume 22, pages 103–111. ACM SIGGRAPH, Addison Wesley Publishing Company, August 1988.

[HS85] Pat Hanrahan and David Sturman. Interactive animation of parametric models. *The Visual Computer*, 1(4):260–266, December 1985.

[Joh95] Michael B. Johnson. *Wavesworld: A testbed for constructing three-dimensional semi-autonomous animated characters*. Ph.d. thesis, Massachusetts Institute of Technology, 1995.

[Jon76] Ben Jones. An extended algol-60 for shaded computer graphics. In Toby Berk, editor, *Computer Graphics (Proceedings ACM Symposium on Graphic Languages)*, volume 10, pages 18–23. ACM SIGGRAPH/SIGPLAN, April 1976.

[LB94] Oliver Laumann and Carsten Bormann. Elk: The extension language kit. *USENIX Computing Systems*, 7(4), 1994.

[May94] Stephen F. May. AL: Animation language reference manual. Technical Report OSU-ACCAD-11/94-TR3, ACCAD, The Ohio State University, November 1994. http://www.cgrg.ohio-state.edu/~smay/AL.

[NDW93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[OO81] T. J. O'Donnell and Arthur J. Olson. GRAMPS – A graphics language interpreter for real-time, interactive, three-dimensional picture editing and animation. In Henry Fuchs, editor, *Computer Graphics (SIGGRAPH 81 Conference Proceedings)*, volume 15, pages 133–142. ACM SIGGRAPH, August 1981.

[Pfi76] Gregory F. Pfister. A high level language extension for creating and controlling dynamic pictures. In Toby Berk, editor, *Computer Graphics (Proceedings ACM Symposium on Graphic Languages)*, volume 10, pages 1–9. ACM SIGGRAPH/SIGPLAN, April 1976.

[Pix89] Pixar. *The RenderMan Interface, Version 3.1*, September 1989.

[Pix96] Pixar. PhotoRealistic Renderman. San Rafael, CA, 1987-1996.

[PSM95] Alan Paeth, Ferdi Scheepers, and Stephen F. May. A survey of extended graphics libraries. In Alan Paeth, editor, *Graphics Gems V*. AP Professional, 1995.

[Rey82] Craig W. Reynolds. Computer animation with scripts and actors. In R. Daniel Bergeron, editor, *Computer Graphics (SIGGRAPH 82 Conference Proceedings)*, volume 16, pages 289–296. ACM SIGGRAPH, Addison Wesley Publishing Company, July 1982.

[ROL90] William T. Reeves, Eben F. Ostby, and Samuel J. Leffler. The menv modelling and animation environment. *Journal of Visualization and Computer Animation*, 1(1):33–40, August 1990.

[SC92] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, volume 26, pages 341–349. ACM SIGGRAPH, Addison Wesley Publishing Company, July 1992.

[Sch96] Ferdi Scheepers. *Anatomy-Based Surface Generation for Articulated Models of Human Figures*. PhD thesis, The Ohio State University, June 1996.

[SPMC96] Ferdi Scheepers, Richard E. Parent, Stephen F. May, and Wayne E. Carlson. A procedural approach to modeling and animating the skeletal support of the upper limb. Technical Report OSU-ACCAD-1/96-TR1, ACCAD, The Ohio State University, January 1996.

[Wer94] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.