SCRIPTS: A COMPUTER ANIMATION SYSTEM

Luiz Velho†

The Media Laboratory
Massachusetts Institute of Techology

Abstract

The problem of specification of temporal transformations for Computer Animation production is investigated. Based on this analysis, an interactive animation language is developed which supports both procedural and key-frame animation. It is a flexible software environment for the design and prototyping of animation programs and interfaces. The language is implemented in C within the Unix operating system, and consists of C-like expressions, built-in functions and animation constructs. There is also an escape mechanism to run Unix commands. A small set of animation tools is also developed to exemplify the system's utilization. These include a three dimensional model interface library, a spline library, and simple mechanics, collision detection and inverse kinematics functions.

Key Words: Computer Animation , Simulation, Actor Systems, Interactive Systems.

1. Introduction

Animation design or scripting is the description of the temporal transformations in a scene. The way in which this information is specified, depends on the control modes and interfacing techniques used. Control modes can be classified as interpolated and algorithmic animation. They correspond, somehow, to the subtle difference between data and programs.

Interpolated Animation specifies a sequence of data elements describing the state of the scene at successive points in time. In general, all the details must be specified, and although the animator has complete control of the transformations, complex animation is usually difficult to describe. Interpolating systems can be subclassified further according to the input methods. Motion tracking, score-based, and key-frame systems are some examples. In motion tracking systems, actual movements recorded in real time serve as the input for the animation [Maxwell 83]. In score-based systems, movements are described in an alphanumeric choreographic notation, very much like a musical score [Badler 79]. In key-frame systems the state of the scene is described at key frames, and the inbetween frames interpolated automatically by the system [Stern 83] [Gomez 85].

Algorithmic animation specifies, through a set of procedures, the rules that regulate transformations in the scene. This is a powerful method that can solve

---

† Author's current address: Globo Computação Gráfica, Rua J. Carlos, 101, Rio de Janeiro.

complex animation problems, but, on the other hand, usually requires intensive software development. Algorithmic systems, according to the degree of abstraction supported, range from a general programming language enhanced for animation, to a task level knowledge-based system [Zeltzer 84b].

Several authors, however, have recently acknowledged a need for interpolated and algorithmic control modes under the same system [Fortin 83] [Zeltzer 84a], and for an integration of control modes [Hanrahan 84], because those approaches individually are unable to provide an effective interface for animation development.

This calls for a model of interaction that associates the power of algorithmic control with the high level of manipulating expression provided by interpolated animation, and that also, incorporates flexibility to describe a wide range of dynamic situations, and extensibility for creating animation abstractions.

The script language integrates algorithmic and interpolating animation control, while being compatible with the criteria of extensibility and flexibility. The animation control mechanism built into the language is designed to support synchronized independent events, similar to actors in ASAS [Reynolds 78] and MIRA [Thalmann 83].

In comparison with ASAS and MIRA, Script differs mainly in three distinct aspects. The first is related to the programming base language. Script is implemented in C, following its main characteristics, while ASAS and MIRA are based respectively in Lisp and Pascal. The second is concerned to the way events are handled: Script allows the definition of nested parallel processes, that is supported neither by ASAS nor by MIRA. This closure property is very important, because it makes possible to break complex animation problems in simpler ones, easier to program and mantain. Third, the Script language provides a more general scheme for the description of dynamic animation parameters.

2. The Script Language

Script is an animation language based on concurrent synchronized events. It has C-like expressions, control of flow statements, C built-in functions and animation constructs.

The language is meant to complement C at a higher level, providing an interactive mechanism for the description of animation processes, features that C doesn't have. As a consequence of this objective, a mechanism to link C code is incorporated into the language, allowing the simultaneous use of script and C. These buit-in functions behave the same way as regular functions, and should be used whenever faster or lower level control is necessary.

The basic actions of the language are specified by statements, in the form of expressions, compound, control of flow and animation constructs.

The animation description constructs are scripts, events and tracks. Scripts are time programs that are executed in parallel to generate animation. Tracks are time variables used to define dynamic animation parameters.

Scripts and Events

The script construct is the primary element for animation specification in the language. Scripts are static algorithmic descriptions, that can be instanced into dynamic events to perform animation actions. They can be abstracted as temporal functions that will be active for some period of time. Script variables

are local to each instance, and last while that instance exists. They are prototypical, in the sense that one script description may originate several instances of distinct events.

Events are the run-time instantiation of scripts that model their temporal and algorithmic properties. They are composed of a body of instructions and a private memory that registers the uniqueness of each instance in relation to the others. Events can generate other events, defining a hierarchy of procedural instances implicitly ordered and synchronized by activations at time boundaries, during their active periods.

Once a script is started, the event instance lasts until explicitly stopped, being activated for every time interval. If an event is stopped, its dependent sub-events are stopped as well. Synchronization between events is guaranteed by the parallel activation of events at time interval boundaries. Each event has incorporated in its memory a local time, that is automatically updated by these activations. The activation consists of the time update, the evaluation of all expressions in the event body, and the recursive activation of dependent sub-events.

The periodic activation of the event hierarchy originates at the highest level of the event tree. Time acts as streams of updating rates that flows through events at each activation cycle. In this way instantiated scripts are played, in very much the same manner as film projectors or videotape players.

Intercommunication of events is possible, in a limited way, by means of global variables. This solution provides a coarse level of communication adequate for simple interactions.

Tracks

Tracks are data structures that hold the necessary elements to describe time variable parameters. They constitute a flexible mechanism that accommodate the needs of various different parameter types. Tracks are specified by a list of values and a set of manipulating functions. Each type of track may have different configuration, as well as, different functions assigned to it. Track values at particular instants in time are accessed through some of these functions. Several techniques can be used to generate them, and, in general, but not necessarily, there will be some kind of interpolation, such as spline, parabolic or linear functions. Other functions will perform various manipulating operations. The basic ones are: insertion, deletion and modification of elements; and access of the first, last, previous and next elements in the track structure.

Script algorithmic descriptions, event control mechanisms, and track data objects, make up the set of primitive animation entities in the language. Together they form the basis upon which other animation abstractions can be built, extending the repertory of operation within the language.

3. Language Syntax

The script language syntax, with few exceptions, is patterned after the C programming language. Comments, identifiers, operators, numerical and string constants are specified in the same way as in C.

The basic data types are real and string variables, track and event structures. Real variables are double precision floating point values. String variables hold arrays of characters terminated by a null character. Track structure is a list of marks and Events are references to script instances. The data declaration grammar is:

```
data_decl:  REAL namelist;
          | STRING namelist;
          | TRACK namelist;
          | EVENT namelist;
```

Functions and scripts are the primary procedural objects in the language. The procedure declaration grammar is:

```
proc_decl: FUNCTION name ( param_decl )
           localvar_decl
             { stmtlist }
           | SCRIPT name ( param_decl )
           localvar_decl
             { stmtlist }

param_decl:data_decl_list
localvar_decl:data_decl_list
```

Expressions are combinations of primary expressions and operators. Primary expressions are numerical constants, reference to variables, function calls, and parenthesized expressions. Binary operators in decreasing order of precedence are (^ * / + - = < => == != && || =) with the same meaning as in C. Unary operators are the arithmetic and logical negation, respectively - and !. Function calls may be regular functions or C builtin functions. Arguments are a possible empty list of expressions separated by commas. The expression grammar is:

```
expr:       NUMBER
          | variable
          | ( expr )
          | expr BINOP expr
          | variable ASSIGNOP expr
          | UNOP expr
          | function ( arglist )
          | built-in ( arglist )
```

Statements specify the elementary actions in the language, that can be expressions, compound, control of flow and animation constructs. The statement grammar is as following with optional definitions inside angle brackets.

```
stmt:       expr ;
          | { stmtlist }
          | RETURN [expr] ;
          | IF ( expr ) stmt [ELSE stmt] ;
          | FOR ( [expr]; [expr]; [expr] ) stmt ;
          | PLAY script ( arglist ) ;
          | [event = ] START script ( arglist ) ;
          | STOP [event] ;
```

4. The Computing Environment

We propose as a model of interaction, an open mechanism based on layers of functional abstractions and multiple interfacing processes. Functional abstractions are specified in script or C language, and developed with conventional

programming tools, such as interpreters, compilers, text editors and debuggers. The interface between the script language and C facilitates the integration of levels in the development of animation tools. Layers of functional abstractions are created with those primitive algorithmic constructs to define a hierarchy of animation entities that manipulate object parameters at different levels of detail. At the top level the user interacts with a complex of simulation machines associated to several interfacing processes. They generate the appropriate controlling parameters, and the animation is produced.

The script language supports the prototyping of these animation interfaces, assembled from a set of pre-defined building blocks. The animation and utility tools developed establish a basic interaction protocol between the script language and other components of the animation system, and also, define a primitive set of animation control procedures. They are related to general input/output, three dimensional object modeling, event/track manipulation, interpolation, collision detection, viewing and display control. These tools are C procedures, implemented as built-in functions into the script language.

The user interface is one of the most critical elements in the animation system, mainly because it is the accessing channel to the system's capabilities. The interface, besides human factors considerations, has to be complete, in order to allow full use of the system's resources.

In algorithmic systems, like Script, users interact at different levels, and because it is extensible, new elements that require interfacing may be frequently added to the system. This means that a complete user interface, in this case, will actually be a meta-interface - a development mechanism for prototyping and refinement of animation interfaces. Furthermore, multiple interfaces may coexist, in the context of a multiprogramming system, sharing global data objects.

The methodology adopted here for the development of user interfaces is known as the building block approach [Foley 82] [Green 82]. In this approach, interfaces are assembled from basic modules that implement common interaction techniques. Interfaces may also be created by the selective addition/modification of existing prototype interfaces.

Three main interfaces are used most of the time in conjunction with other interfacing modules developed with these tools. They are: the script interpreter, a script editor, and a track editor.

Emacs, a powerful screen-oriented text editor, is used as the script editor. Script files are read, modified and written back whenever necessary between script execution cycles.

The track editor is analogous to the motion editors used in Mutan [Fortin 83] and Bbop [Stern 81], and supports full d-spline [Kochanek 84] track manipulation. The editor maintain a list of tracks that can be accessed and modified interactively.

5. Examples

In this section, simple animation scripts are presented to give a flavor of the language and to demonstrate the system's usage. Some examples are complete working scripts, while others are simplified versions of the actual ones, with the distracting details taken out for clarity in the presentation.

Example 1: A cylinder rolls in one direction with its displacement calculated from its rotation. One track controls directly the cylinder rotation, while the cylinder position is derived from it radius.

```
track cr;

script cylroll()
     real rot, pos;
{
     rot = linear(cr, t);
     setobj("cylinder", "rx", rot);
     pos = pos + (rot * getobj("cylinder", "radius"));
     setobj("cylinder", "px", pos);
}
```

Example 2: A prototype for a cycle script that uses the built-in function stopped to inquire the status of a script instance, starting it again when it finishes.

```
script cycle()
     event e;
{
     if (stopped(e))
          e = start scriptname(arguments);
}
```

Example 3: A ball bouncing inside a cubic space. The ball trajectory is calculated from a initial position and direction vector, using accelerated motion interpolation to account for the effects of gravity. A collision detection test determines the necessary trajectory reorientation whenever the ball is about to cross one of the cube boundaries. The functions newtraj and collision, not shown, were written in C for efficiency reasons. Newtraj modifies the trajectory track parameters for a new trajectory begining at the collision point. Collision tests if there is any intersection between the ball trajectory and the six planes of the cube. It returns a parametric value in the interval [0.0, 1.0] if there is any intersection, and returns 1.1 if no intersection.

```
track px, py, pz;

script inittraj(tkptr tx, ty, tz;
     real x, y, z, vx, vy, vz;)
          real gr;  /* gravitational constant */
{
     gr = - 0.98;
     tkinsert(tx, 0, x, vx, 0.0);
     tkinsert(ty, 0, y, vy, 0.0);
     tkinsert(tz, 0, z, vz, gr);
}
```

```
script bounce(string env, obj;
     real x, y, z, vx, vy, vz;)
     real x0, y0, z0,/* current center */
     x1, y1, z1,  /* new center */
     cp;  /* intersection param */
{
     if (t==0) { /*   initialization*/
          ldobj(env); ldobj(obj);
          attachobj(env, obj);
          inittraj(px, py, pz, x, y, z, vx, vy, vz);
          x0 = accmotion(px, t);
          y0 = accmotion(py, t);
          z0 = accmotion(pz, t);
     }
     x1 = accmotion(px, t); /* calculate new position */
     y1 = accmotion(py, t);
     z1 = accmotion(pz, t);
     if((cp = collision(x0, y0, z0, x1, y1, z1)).1){
          newtraj(cp, px, py, pz, x0, y0, z0, x1, y1, z1);
          t = 1 - cp;
          x1 = accmotion(px, t);
          y1 = accmotion(py, t);
          z1 = accmotion(pz, t);
     }
     set(obj, "pxyz", x1, y1, z1); /* move the object */
     x0 = x1; y0 = y1; z0 = z1; /* update curr position */
}
```

Example 4: The demo script combines the bouncing ball example with the a robotics manipulator function to produce the animation sequence shown in figure 1. The robot arm grasps the ball, lifts it up, and throws it. The ball hits the wall several times and bounces back until it stops completely. The ball initial trajectory, elasticity and gravitational coeficient are controlled by tracks and global variables. The script launch guides the arm's motion.

```
script launch()
     real x, y, z;
{
     x = spline(lx, t);
     y = spline(ly, t);
     z = spline(lz, t);
     movearm(x, y, z);
     if (t > tknext(lx, 0) && t < tkprev(lx, tklast(lx)))
          setobj("ball", "pxyz", x, y, z);
     if (t == tklast(lx)) stop;
}

script demo()
     real x, y, z, vx, vy, vz;
{
     if (t == 0) start launch();
     if (t == tkprev(lx, tklast(lx))) {
          x = tkget(lx, t, 0);
          y = tkget(ly, t, 0);
          z = tkget(lz, t, 0);
```

```
            vx = x - tkget(lx, tkprev(lx, t), 0);
            vy = y - tkget(ly, tkprev(ly, t), 0);
            vz = z - tkget(lz, tkprev(lz, t), 0);
            start bounce("room", "ball", x, y, z, vx, vy, vz);
        }
        if (intr()) stop;
        display("scene");
    }
```

6. Summary

We have proposed the integration of interpolated and algorithmic animation in a system that, based on the simulation paradigm, allows animation development in layers of functional abstractions, and its specification through multiple interfacing processes.

The central coordinating mechanism in the system is an interactive interpreter for a computer animation language - Script, that supports concurrent synchronized events, and track data structures. The Script language is intended to perform a double duty, in both the description of temporal object transformations, and the prototyping of animation interfaces, playing an important role in the realization of our model of interaction.

The design philosophy behind the Script animation system was to create an open mechanism for animation production, that being flexible and extensible would evolve with day to day use.

7. Extensions and New Directions

The Script language would benefit from the addition of several features, among them: a richer set of operators, such as the reminder (%), compound assignments (+=, -=, *=, /=, %=), and auto increment/decrement (++, --); string operations that could be implemented as built-ins, such as copy, compare and concatenation; a three dimensional vector data type; array data structures for the existing data types; and additional control of flow constructs like break and continue. The animation mechanism of the language could be enhanced with the addition of a time structure, and a send/receive communication scheme.

New directions for work on the subject point towards the research of more sophisticated ways and resources for the description of complex animation problems. Some emerging topics that need to be explored include: general collision detection, dynamics simulation tools, object manipulators, and knowledge-based animation.

8. Acknowledgements

Thanks to David Zeltzer, whose work has been a major contribution to the computer animation field, and a invaluable source of inspiration to me.

Thanks to Gloriana Davenport for her interest in this project and help in administrative matters. Thanks to Christopher Sawyer-Laucanno for proof-reading and style suggestions.

Thanks to Project Athena for the use of computing resources, and especially to Jim Gettys for providing the VS-100 display software.

9. References

[Maxwell 83] Maxwell, Delle Rae., Graphical Marionette: A Modern Day Pinocchio., Master's thesis, Massachusetts Institute of Technology, 1983.

[Badler 79] Badler, N. and Smoliar, S., Digital Representations of Human Movement., ACM Computing Surveys 11(1):19-37, March, 1979.

[Stern 83] Stern, Garland., Bbop - A Program for 3-Dimensional Animation. , NICCOGRAPH '83, pages 403-404., Niccograph, Tokyo, Japan, December, 1983.

[Gomez 85] Gomez, Julian., Twixt: A 3D Animation System, Computer and Graphics 9(9): pp. 291-298., Pergamon Press Ltd, (1985),

[Zeltzer 84b] Zeltzer, David., Issues in 3-D Computer Character Animation. , 1984, Course Notes: Tutorial on Computer Animation - ACM Siggraph 84.

[Fortin 83] Fortin, D., Lamy, J. F. and Thalmann, D., A Multiple Track Animation System for Motion Synchronization., Motion: Representation and Perception, pages 180-185., ACM Siggraph/Sigart, April, 1983.

[Zeltzer 84a] Zeltzer, David., Representation and Control of Three dimensional Computer Animated Figures., PhD thesis, Ohio State University, 1984.

[Hanrahan 84] Hanrahan, Pat and Sturman, David., Interactive Control of Parametric Models., 1984, Course Notes: Tutorial on Computer Animation - ACM Siggraph 84.

[Reynolds 78] Reynolds, Craig William., Computer Animation in the World of Actors and Scripts., Master's thesis, Massachusetts Institute of Technology, 1978.

[Thalmann 83] Thalmann, D. and Magnenat-Thalmann N., The Use of High-level 3-D Graphical Types in the Mira Animation System., IEEE Computer Graphics and Applications, 3(9):9-16, December, 1983.

[Foley 82] Foley, James and Van Dan, Andries., Fundamentals of Interactive Computer Graphics., Addison Wesley, 1982.

[Green 82] Green, M., Torwards a User Interface Prototyping System. , Graphics Interface 82, pages 37-46., NCGA Canada, May, 1982.

[Kochanek 84] Kochanek, Doris., Interpolating Splines with Local Tension, Continuity and Bias Control., Computer Graphics, pages 33-42., ACM Siggraph, July, 1984.

[Baecker 69] Baecker, R. M., Picture Driven Animation., In Proceedings AFIPS Spring Joint Computer Conference, pages 273-288., AFIPS, 1969.

[Burtnyk 73] Burtnyk, N. and Wein, M., Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key-Frame Animation., Communications of the ACM 19(10):564-569, October, 1973.

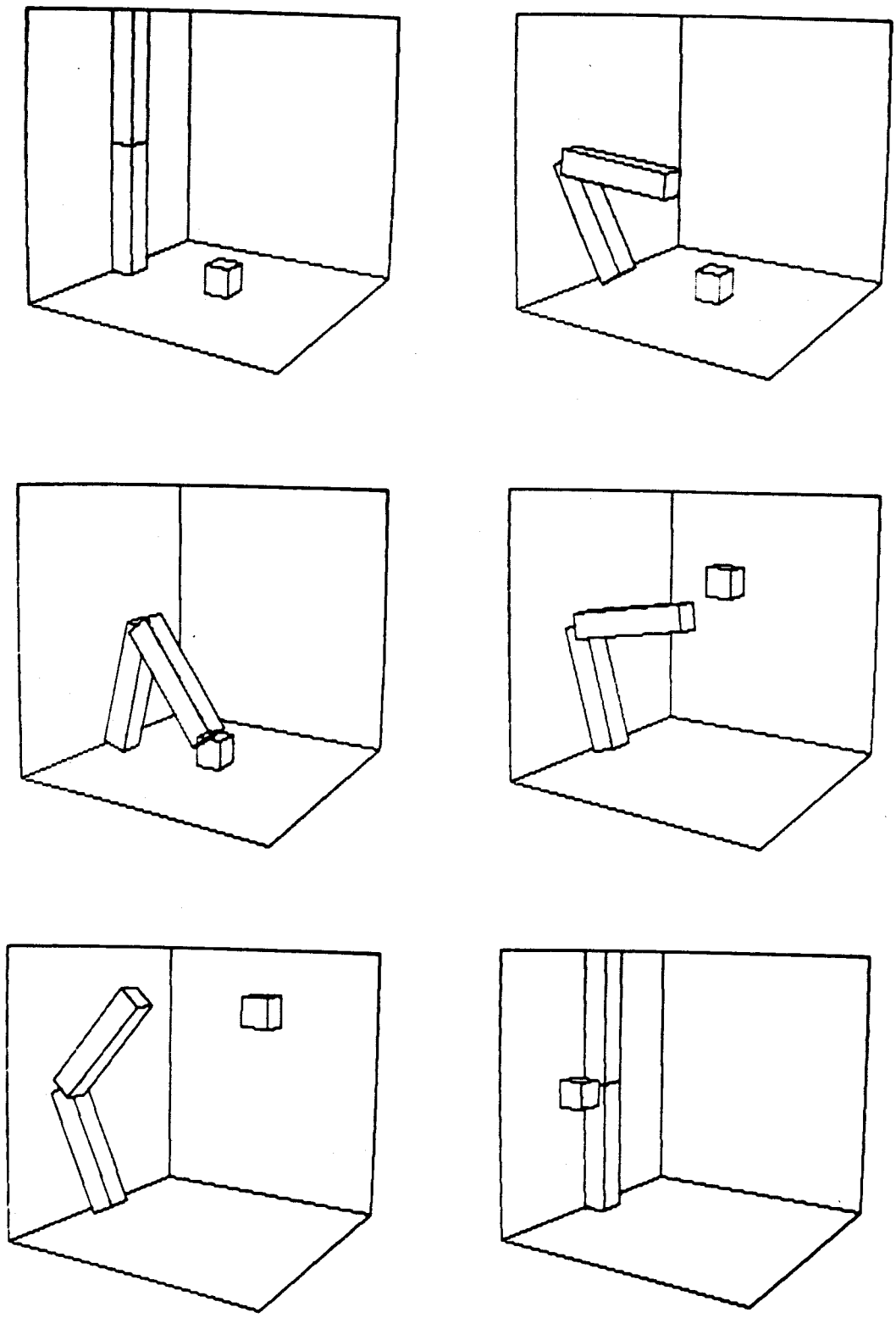

Figure 1: Robot arm and bouncing ball animation sequence