Integrating a Scripting Language into an Interactive Animation System

Michael Gervautz, Dieter Schmalstieg
Institute of Computer Graphics
Technical University of Vienna, Austria
email: gervautz@eibyte.dnet.tuwien.ac.at, dieter@eigsg1.tuwien.ac.at

Abstract

STORYBOARD is a scripting language for an interactive computer animation system. The language was designed to be simple in its use, to support various animation techniques provided by the animation environment, and to support procedural abstraction of animation. It is easily expandable and lends itself to the integration of interactive features of the system.

Some of the important aspects of design and implementation are discussed, like the time-table like structure of programs, the embedding of a message mechanism into the language, animation data types and their application, concurrent execution and local time, and recompilation of an animation specified interactively into a script.

Keywords: *computer animation, scripting languages, actors, controllers, local time, subscripts, linear transformations, time dependent data types*

1. Introduction

When using a programming language for computer animation - a *scripting language* - the animator is required to write a program to define a set of actions. Many animators are rather artists than programmers and therefore prefer using an interactive system with a graphical user interface. However, there are significant advantages of scripting languages over interactive systems, especially flexibility and diversity of algorithmic notation (see also [28]).

Because both interactive and scripted animation has its distinct areas of application, a system aimed at integration of animation techniques should not only provide multiple techniques to work with, such as dynamic simulation [26] or behaviors [13], but also multiple means to specify the animation, either interactively or by a program.

We present the scripting language STORYBOARD [21] that is part of the animation system VAST (Vienna Animation System Technology) [10]. This language was designed to fulfil a number of needs that could directly be derived from its area of application and its system environment:

- It should be **easy to use**. Animators are often unskilled in programming and are easily deterred by a complex scripting language that requires experience in program design rather than in animation design. The syntax of the language should be simple and natural, and the language elements such as built-in functions and data types should be powerful.
- It should **support different animation techniques**. All the animation techniques that are embedded in the system must be accessible via the language. A number of basic and auxiliary data types are necessary to allow for efficient communication with the animation modules. Furthermore, the language must lend itself easily to extensions as new animation techniques are added to its host system VAST.

0-8186-6240-9/94 $3.00

- It should be suitable for the **integration of interaction**. A paradigm of VAST is interactivity, and from this rises the problem of how changes made interactively can be reflected in the script that represents the animation.
- It should offer opportunities for **procedural abstraction**. Animations can quickly get complicated as the number of participants grows, and the only way for the animator to keep track is to introduce subdivisions and hierarchies. Besides, for the sake of productivity, reusability of animation parts is as crucial as reusability of software modules. The notion of time should be flexible so that parts of the animation can easily be altered in the time domain without affecting the rest of the animation [22].

2. Concepts of scripting languages

First we will give an overview of the essential concepts that can be found in scripting languages. As stated in [Breen 1987], the "object-oriented paradigm is an advantageous, useful and natural concept" for computer animation. Therefore most computer animation systems are more or less object-oriented. In such systems, objects **communicate** with each other via **messages**. This message mechanism is also used if entities execute concurrently and synchronisation is needed. Therefore animation systems and languages based on the actors/scripts paradigm must provide a message mechanism. Almost all known animation languages have such a mechanism, cf. [3], [19], [23], [11], [8], [4].

Animations describe scenes composed of visible entities, and their changes over time. Such entities in computer animation are generally called **actors** [24], a term that suggests coherence of computer animation and theater/movie and stresses the active role (lat. agere - to do) such objects play in computer animation. Besides, the notion of "actors" matches the animator's self image as a "director" [13]. The term actor is used in robotics and artificial intelligence in a similar way [1].

[14] develops a theory about entities he called actors. He defined an actor as an object that can send or receive messages. All elements of a system are actors and the only activity possible in the system is the transmission of messages between them. Actors form a natural level of abstraction that is suitable for the object-oriented approach. Besides, animation often involves behaviors or goal driven efforts. Some researchers try to apply methods of artificial intelligence to computer animation to supply actors with methods to solve the problems the actors are confronted with [20].

While actors can only execute simple tasks, a **script** controls the animation on a very high level. It fulfils a task similar to the process manager in a multitasking operating system. The script creates and deletes actors and triggers specific events. Such events are often called *cues*. The script and the actors execute concurrently. Script elements can be found in [19], [25], [4], [20].

The code to evaluate time-dependent differential functions frame by frame can quickly become very unpleasant to write and maintain. To overcome this restriction, new data types, that may best be described as **time dependent functions**, are introduced to help the animator. The value of a such a variable may change over time. Examples for this concept are *newtons* [19], *animated basic types* [23], or *articulated variables* [18], [15].

3. The animation system: VAST

VAST (Vienna Animation System Technology) [12] [10] is an interactive animation system that is able to integrate and combine various animation techniques like keyframe animation, scripting, physically based and behavioral animation. The system is based on objects that communicate with each other through message passing. Basically there are two different types of objects in VAST.

- **Actors** appear in the animation. An actor is able to send and receive messages and to react on such messages. Actors can be simple mass points as well as rigids, kinematic chains, deformable objects, etc.
- **Controllers** influence the animation. The main task of a controller is to send messages to actors to change their motion, shape, or non-geometric properties. Controllers are usually not visible. They contain dynamic data, the description of the motion, or a behavior of one or more actors. Separating the controlling mechanism from the actors provides a flexible environment in which different controlling mechanisms can easily be implemented and integrated.

An example for a simple controller is a spring that controls the distance of two objects. Other examples for controllers are dampers, kinematic and dynamic constraints, or input devices, which also can be controlled by the animator. Scripts are also controllers.

An animation in VAST is calculated incrementally: For every frame *animated objects* (actors and controllers) send messages to each other. Each object has its own message buffer to store the incoming request messages during one time step. When all messages are sent, the objects react to these messages. In case of multiple messages to one object, a **priority mechanism** decides which messages are considered. It has to order the messages, possibly discarding irrelevant ones, or combining others (e.g. summing up forces).

VAST is implemented in C++, defining a broad hierarchy of animation objects. The animated objects are the key to the integration of multiple animation techniques into VAST. [Fig. 1] gives an overview of the object hierarchy in VAST. The hierarchy splits into two sub-trees. the actor sub-tree and the controller sub-tree. A member of the latter is a "script" (= script controller) that is discussed in the section 6.

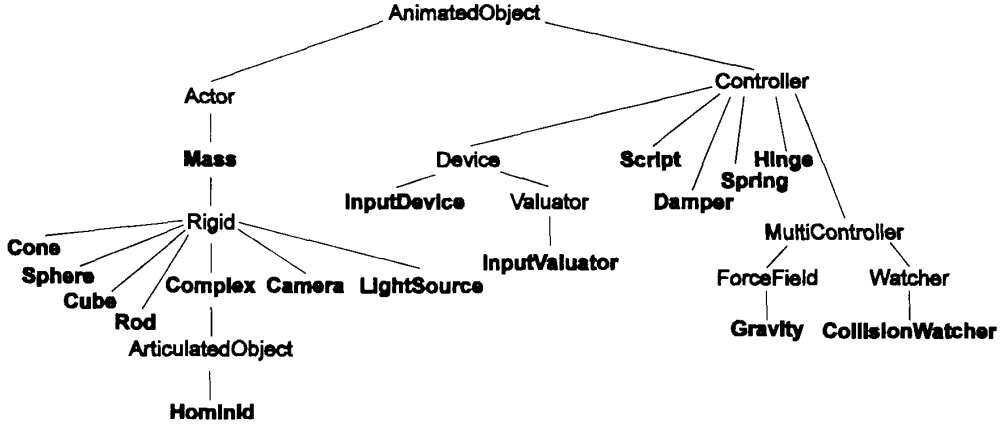


Fig. 1: Excerpt from the object hierarchy currently implemented in VAST

4. The STORYBOARD language

Many of the animation languages that have been proposed so far are extensions to existing general purpose programming languages, such as MIRA-3D [23] that extends PASCAL, ASAS [19] that extends LISP, or ML [18] that is close to C in its syntax. While these approaches are certainly powerful, they lack simplicity. LISP is definitely not the language of choice for artists who work as computer animators.

The problem with these languages is that they are ment as a toolkit upon which the **animation techniques** can be built. The person who implements the module that processes the animation technique (e.g. inverse kinematics or particle systems) is the same person that uses this module to produce animation. The disadvantage of this approach is obvious: The animator has to be skilled in both implementation and artistic animation, which is usually not the case.

For this reason we think that the concept available in most commercial software is superior: The functions of a system can be accessed by a simple language (often coined "macro" language) that can only deal with existing animation modules but neither create new modules nor modify the internal behavior of existing modules. An option is to create "virtual" new modules by combining existing ones, but this can be achieved on a very high language level. In that way, we introduce a "two-class-society" among people working with the system: The technical animators (or programmers) develop new - technically complex - animation modules in the system's native language (C++), and the artistic animators use this modules in the high level animation language (STORYBOARD) to create animations without the need to

know about how a given animation module works (cf. [23]).

Such a language has to be really natural in its use. The general idea of a STORYBOARD script was to resemble the structure of an exposure sheet in movie production or a time-table in every-day life. It is merely a list of points of time associated with one or more statements that are to be triggered at that moment.

```
AT 1   : statement1; statement2;
         statement3;
AT 3   : statement4;
AT 3.5: statement5; statement6;
```

Inside of such an AT-construction the statements are executed in the order given. This is important because the execution of one statements (e.g. an assignment) may affect the next one.

The central type of statement in STORYBOARD is the **message statement**. The message defined by those statements are sent to other objects at the appropriate time. This is the way a script sets up and controls an animation.

The message statements are implemented in the form of simple command sentences that are inspired by natural language. The *subject* of the sentence is addressed by a name, followed by the *command verb* and *objects* completing the command sentence (these are objects in linguistic sense, not in OOP terminology!). Comments embraced in single quotes (e.g. `'comment'`) may be used as *fill words* to bring a message statement even closer to natural language.

In the example given next, the parameters of a mass point are set. `set` is the message name, `position` is the label for position and `velocity` is the label for velocity.

```
VAR MASSPOINT masspoint1;
                'variable declaration'
AT 12: masspoint1 set position 'to'
  [0,0,0] 'and' velocity 'to' [0,0,0];
```

Special messages are reserved to tell objects to become visible or invisible: `appears` and `disappears`.

```
AT 13: masspoint1 appears;
```

The time-table like program structure consisting of message "sentences" is easily comprehensible by both the animator (who writes the script) and by the compiler (who must be capable of processing it). Let us examine a typical message statement like the one given in the example above: Objects participating in the animation are declared as variables in the script, so the *subject* of the message is addressed by a **variable name**. The *verb* (like "set") is a **method** in object-oriented terminology, and the *sentence objects* are the **parameters** for the method. Unlike most programming languages, the order of parameters is free. To avoid ambiguities, parameters are preceded by a parameter **label** that is unique for that method and allows type checking to be performed upon the expression that follows the label.

Data types

The animation objects (actors and controllers) are implemented in STORYBOARD in the form of data types. Creating a new object is done by declaring it as a variable, and communication with these objects is established by sending messages. In addition to those object data types, a number of primitive data types has been added. With these data types basic programming tasks can be carried out, like calculations and assignments. The results can then be used as parameter values for messages. The basic data types are **reals**, **booleans**, **3D vectors**, **quaternions** (useful for rotations), and **strings**.

Some data types are specially designed for computer animation: **linear transformations** (for the definition of simple movements) and **anireals** (a time-dependent numerical type), see below.

Programmed animation

The field of program level animation as defined in [28] is easily covered by the scope of our script language. In addition to the language elements that have already been given, it is possible to use conditional statements of the form

```
IF condition THEN... ELSE... ENDIF;
```

and loop statements of the form

```
LOOP... EXIT IF condition;... ENDIF;
```

to enable efficient programming.

Furthermore, some special data types for keyframe animation have been introduced: anireals and linear transformations.

Anireals

Time dependent data types have been adopted in the form of *anireals* [9] (short for **ani**mated **reals**). Anireals

are a superset of normal real numbers. They have the same syntax as reals and behave exactly like reals do, with the exception of the possible dependency on time. Time is denoted with the keyword TIME.

```
ANIREAL a, b;
a := SIN(5);
b := 23 * COS(TIME) + a;
```

Anireals are compatible with reals. Reals and anireals can be used together in numerical expressions, but the result is always of type anireal. Therefore an expression containing one or more anireals cannot be assigned to a real.

To make use of an anireal, an *anireal valuator* is required. An *anireal valuator* is a controller that evaluates an anireal and modifies a specific property of an animated object according to the result. The anireal valuator executes completely independent from the script that creates it. Once the anireal valuator is told to send messages to a particular object, the controller does so continuously until it is told to stop.

To create key-frame animation, a good way is to define the position as a function dependent on time - an anireal. The anireal is created by the script, then passed to an anireal valuator together with the order to supply an internal state variable of the object to be guided continuously with the time-variant value. This step is called "coupling". From the moment of coupling, the animator does not have to care any more about that particular part of the animation. The system takes care of this part of the animation, that goes on concurrently with the other parts. Of course the object also can be decoupled from the valuator again. In the following example, a sphere's mass is continuously incremented according to the square of time. The important statement is the message couple that links the anireal valuator to the sphere (OBJECT indicates the target object and KEY "mass" indicates what property of the target to manipulate).

```
SCRIPT HeavySphere;
VAR   a: ANIREAL;
      v: ANIREAL_VAL;
      s: SPHERE;
AT 0: a := TIME * TIME;
      v couple ANI a 'with'
        OBJECT s 'by' KEY "mass";
```

Anireals are a very powerful tool that can not only be used to implement simple functions like polynomials, but also complex (for example composed) functions.

Linear transformations

Like anireals, *linear transformations* are another data type designed for describing animation. A LINTRAFO variable (from **linear transformation**) stores a simple movement description composed of elementary operations like translation, rotation or scaling. These movements can be linked to execute either sequentially or concurrently with the keywords SERIAL and PARALLEL, respectively. Every part of that movement may be given a duration (LASTS).

Like the anireal valuator, the lintrafo valuator is also used for guiding level animation, but with a linear transformation as a control structure instead of an anireal. Linear transformations directly affect the position and orientation of a rigid.

```
SCRIPT SquareMove;
'move a sphere along a square'
VAR   l: LINTRAFO; v: LINTRAFO_VAL;
      s: SPHERE;
AT 0: l :=
  TRANS [10,0,0] LASTS 5 SERIAL 'rite'
  TRANS [0,10,0]  LASTS 5 SERIAL  'up'
  TRANS [-10,0,0] LASTS 5 SERIAL left'
  TRANS [0,-10,0] LASTS 5;      'down'
v couple LIN l 'with' OBJ s;
```

Compared with anireals, linear transformation are more limited in applicability, but more easy to use. The description used in linear transformations matches the way the human animator thinks about movement. The area of application is restricted to movement, but this is the majority of action going on in an animation.

Spline-driven keyframe animation

In addition to analytically defined keyframe animation, it is also possible to define motion or change of other parameters over time by splines. A special controller is introduced for that purpose, a spline valuator.

A spline is usually defined by a piecewise cubic polynomial. Animators usually wish not only to have control over the spatial properties of the curve, but also over the velocity and duration properties. All these properties are much easier set interactively than algorithmically (i.e. in a script). Therefore splines are set up in a separate interactive tool. The spline's properties are saved in a curve file and the script has only to refer to that file.

The spline valuator works in the same way the anireal valuator does. It is told the name of the curve from which to obtain the spline data, the object that shall be supplied with values, and the key property of the target object to couple to.

```
VAR SPLINE_VAL sv; SPHERE s;
AT 1: sv couple SPLINE "up_and_down"
'with' OBJECT s 'by' KEY "position";
```

5. Extending the language

Easy extendibility was a key feature in the design of STORYBOARD, because VAST is built to be a testbed for the integration of new techniques. The animation object's properties in STORYBOARD are table driven, so that extending the language with a new animation objects simply means feeding the object's method declarations to a table.

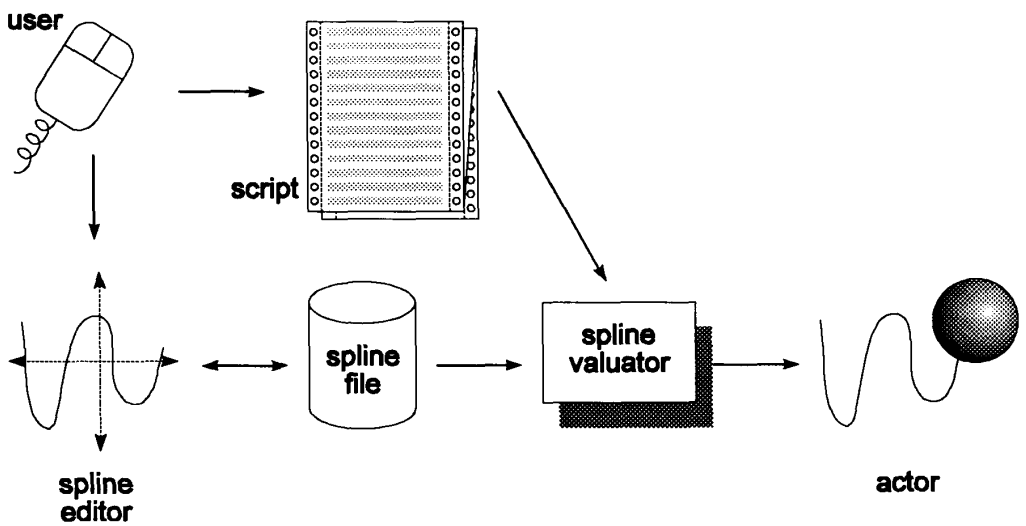


Fig. 2: Data flow for a spline valuator

Physically based animation

Physically based animation is achieved by animation objects with physical properties. Actors like rigids have a mass and a geometric representation, and they react to physical influences like spring forces, gravity, or force fields. The latter are implemented as controllers.

To create a physical simulation with a script, the animator has just to set up the scene in its initial state. The animation itself is derived automatically by the object's skills, exercised by the system's message mechanism.

The example given in the program on the next page shows how this princeple works: A tetrahedron is created by tieing together four mass points with six springs, thus forming a "jelly tetrahedron". Furthermore, a gravity object is introduced that influences the masses by applying a force.

For example, behavioral animation models such as particle systems [17] or other models that are not based on a physical layer are as easily integrated into STORYBOARD as the techniques already mentioned. New animation modules are built in the systems native language of the system (C++), and then added as new object data types into the script language. Since this step means only adding a new data type defined by a set of messages (each of which comes with a set of parameters), the extension of the language is straightforward. Not the syntax is extended, but rather the semantics. A sufficient set of basic data types that can be used as parameters guarantees that a new animation object fits into STORYBOARD's framework.

6. Procedural abstraction

Procedural abstraction means the introduction of language concepts that enable the user to structure his scripts,

make repeated use of code once written (reusability), and be able to alter it, especially the script's timing properties.

Subscripts

In general purpose programming languages, an essential abstraction mechanism is provided by procedures and functions. Support for modules in the form of multiple source files for one program has also been proven to be advantageous. This concept is built into STORYBOARD in the form of subscripts.

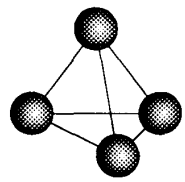
Subscripts are a way of decomposing a huge and complex animation into handy animation parts. These parts are stored in separate script program files. A natural way to decompose an animation is scene by scene. But also statements involving one particular actor can be put into a subscript.

Arguments are a very powerful tool. A parameterized script can be used as a subscript in different animations with different parameters. Very often parts of an animation are structurally identical but different in various values. If a parameterized script has been programmed, only the current values have to be supplied to the script and it can immediately be utilized. By use of program control elements such as if-statements variations not only in value space but also in behavior are possible.

It is possible for (sub)scripts to share animated objects. A superscript may pass a reference to an object as an argument to a subscript. In the following, both superscript and subscript may address this object and manipulate it simultaneously by sending messages.

Subscripts principally execute concurrently and independently. They form a tree-like hierarchy, in which the root script is referred to as the *master script*. Multiple instances of a script program can co-exist, even with overlapping execution time. A subscript can be compared with an asynchronous procedure call. A script can even recursively call itself (although the applicability of that feature may be limited in practice). Multiple instances of scripts are completely independent of each other, maintaining their own variables (except shared objects passed as arguments).

```
SCRIPT Jelly;
VAR
  MASSPOINT m1,m2,m3,m4,m10;
  SPRING    s1,s2,s3,s4,s5,s6;
  GRAVITY   gr;
AT 0: 'set up the masspoint properties'
  m1 set mass 10  position [100,100,100]  velocity [10,90,0];
  m2 set mass 10  position [120,120,120]  velocity [10,90,0];
  m3 set mass 10  position [100,120,140]  velocity [0, 90,0];
  m4 set mass 10  position [110, 90,140]  velocity [0.1,90,0];
'tie them together'
  s1 set first_tie m1  second_tie m2  length 60  strength 5;
  s2 set first_tie m1  second_tie m3  length 60  strength 5;
  s3 set first_tie m1  second_tie m4  length 60  strength 5;
  s4 set first_tie m2  second_tie m3  length 60  strength 5;
  s5 set first_tie m2  second_tie m4  length 60  strength 5;
  s6 set first_tie m3  second_tie m4  length 60  strength 5;
'and now let the simulation run...'
  gr appears; m1 appears; m2 appears; m3 appears; m4 appears;
  s1 appears; s2 appears; s3 appears; s4 appears; s5 appears; s6 appears;
```



Subscripts are invoked with a *run* statement, requiring the specification of the name of the file containing the script program, the start and end time of the subscript (in terms of local time of the calling script) and values for the *arguments* expected by the subscript.

```
AT 90: RUN "planecrash" LASTS 5 ARG
plane := boing_747, big_mess := TRUE;
```

In this example, a run statement is executed that produces an animation of a crashing aeroplane. "planecrash" is the name of the script file that contains the program, the value after LASTS gives the duration of the animation, and after ARG the arguments for the subscript are listed. Every entry consists of an

assignment to a local property of the subscript, with the syntax <variable> : = <expression>.

Local time

Time plays a crucial role in animation. A (sub)script describes a coherent part of an animation. Often it is necessary to scale the duration of such a script or move it along the time axis. If all objects refer to the same global time base, this is a difficult task. Therefore different logical times were introduced:

- *Global time*: This is the overall system time. A time unit in global time is the reference element all over the system. The global time is also identical with the master script's time. The global time begins with the start time of the master script and ends with the master script's end time. The framerate is specified relative to global time.
- *Local time*: Every script maintains its own local time that is independent from the global time. This local time is projected onto the lifetime of the script. The majority of scripts are subscripts (all except the master script), so that usually the local time is different from the global time. The lifetime of a script starts at the moment of the call (the value specified after the AT of the RUN statement that starts the subscript) and lasts for the amount of time specified after LASTS. Note that these times are also local times of the calling script and not necessarily identical with the global time. The script hierarchy may contain several layers of subscripts, therefore the local time has to be transformed multiple times to calculate the global time.

If both local start and end time and global start and end time are given for a script, the conversion from a given local time to the according global time and vice versa can be calculated as follows:

$$g = g_s + (g_e - g_s) \cdot \frac{l - l_s}{l_e - l_s}$$

$$l = l_s + (l_e - l_s) \cdot \frac{g - g_s}{g_e - g_s}$$

with:

g...global time	l...local time
g_S...global start time	l_S...local start time
g_e...global end time	l_e...local end time

The formulae above can also be applied for a pair superscript/subscript. If the calculation ought to be performed for a hierarchy of subscripts, the formulae have to be used recursively.

Because all actions specified in scripts are associated with local time only, a script can be transposed and scaled by the user without creating the need for the system to recalculate the timing of the animation. Only the lifetime values for the manipulated script have to be altered.

7. Integration of interaction

To integrate a scripting language into an interactive system, interfaces in two directions have to be defined: from script to animation and vice versa. These two interfaces are discussed in the following.

Transforming a script into an animation: The STORYBOARD controller object

The compiler/runtime system for a script in VAST is a controller object, the *STORYBOARD controller*. For every script program that is part of an animation project (a scene), a STORYBOARD controller is created that compiles the script program. Note the different terms **script object** (or rather STORYBOARD controller), **script program** (that refers to the program text the animator wrote), and **script file** (that refers to the file on disk containing the script program).

A script program may contain calls to *subscripts*, comparable to procedures in other programming languages. For every subscript, a separate STORYBOARD controller is generated, forming a tree-like hierarchy. Upon compilation, the script program is read from the file and parsed. The result of this procedure is a *command queue*, a data composed of *commands*, each of which holds the contents of a message associated with the time when the message should be sent. The control is then passed on to those subscripts to compile their own command queues.

A subscript is treated in the same way as its superscript: The subscript controller exercises the same compilation mechanism as the calling script, but for its own script program. More than one subscript may execute one script program, because all variables are treated locally in the script object.

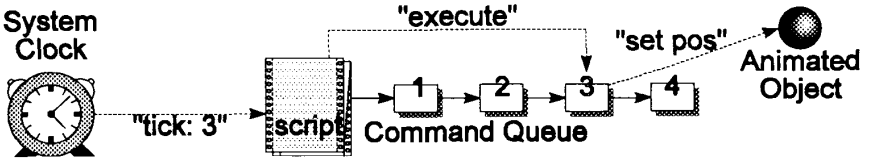


Fig. 3: Executing a command queue

Every controller generates its own command queue that is private to the controller. When the animation is run, all STORYBOARD controllers participate in the communication phase. They consult their command queues to see if any of the commands are scheduled for execution. If this is the case, the command is told to execute, and in turn sends the stored message to an animated object to trigger the desired effect. A command holds all the information that is necessary to post a well-defined message. Because the script program is compiled into a command queue and not interpreted directly upon reading, the animation can be reproduced without having to read the script program again. Changes that are made by the animator via the interactive tools of the system are reflected in the command queue as new or altered commands.

Transforming an animations into a script: Cyberscripts

One of the objectives in designing both the STORYBOARD language and the command queue data structure was to allow script programs to be used as a way of storing animations. This solves the problem of how changes made interactively to an animation that was read from a script program can be reflected in that script program. Such programs, that are automatically produced from the animation worked upon in VAST, are called *cyberscripts*.

The interchangeability of data between the programming module and the interactive module(s) allows the programmer to combine programming and interaction on the same objects. He might specify a sequence by programming then refine it by interaction (which is generally easier because of the visual feedback provided by the graphical user interface as opposed to the unappealing numbers in a program). The programmer might also rough out a design with the interactive module, then view it as a program in a text editor and adjust to an arbitrary precision by the power of control given by real numbers and mathematical expressions. This refinement process cycle can be iterated until the animation matches the animators expectations.

A cyberscript is produced from the command queue. Once a script program is compiled into a linear command queue the program control information (loops, assignments, if-statements) contained in the original script program is discarded. The program control information is immediately interpreted when reading and processing the script file in a way [15] calls "unrolling the model". The "compiled" command queue contains only messages, no program control information. The discarded information cannot be reproduced in a cyberscript built from a command queue. Therefore the original script and the cyberscript are not identical but produce the same animation. The script hierarchy is not lost, because the cyberscript mechanism can reconstruct it from the tree of STORYBOARD controllers.

All information needed for the cyberscript is contained in the STORYBOARD controller and its command queue. The time for a message is stored in the command. The command also holds a pointer to the object it has to send a message to when triggered, and the values of parameters.

The animation stored in a command queue can be manipulated via the user interface by inserting, deleting and altering command nodes. If the animation is stored, the (altered) command queue is processed command by command. Every command is translated into a message statement. If any changes have been made to the animation interactively, these are reflected back in the cyberscript.

Limitations

Further work will examine possibilities of introducing interaction while the animation is executed. Input in the scripting language is not yet available, and there is no way to send messages to a script controller. A possible solution is to extend he command queue by another data structure that can handle asynchronous events, like described by [8].

8. Conclusion

We have presented a script language that fits into an interactive animation system aimed at the integration of animation techniques. The language is easy to learn, so that it can be used by unskilled programmers. This is achieved by leaving the implementation of animation modules at the system level, while utilization of animation modules is achieved by the scripting language on a very high level. Its syntax is simple and has a natural time-table like structure. The message mechanism of the animation system is supported by introducing animation objects as data types and applying message statements on them.

Structured programming is supported and a mechanism for procedural abstraction is introduced in the form of subscripts that can be triggered from within a script and execute concurrently. The animation can be structured in a number of sub-animations that can be edited and manipulated separately, providing high-level control on the timing properties.

Furthermore, special data types (anireals, linear transformations) and splines for simple animation tasks are included that can also execute concurrently with the script by the use of valuator objects. This concept relieves the animator from specifying all details of the animation for every single keyframe.

Powerful animation techniques such as physical simulation are supported by the language's message mechanism, and it is easily extended as new modules are developed.

The animation system's paradigm of interactivity is supported by the possibility to recreate a script from an animation represented in the interactive part of the system. All information that has been specified interactively can be reflected into a cyberscript.

References

[1] Agha Gul: An Overview of Actor Languages. SIGPLAN Notices, Vol. 21, No. 10, p. 58-67 (October 1986)

[2] Barzel Ronen: A Modelling System Based on Dynamic Constraints. Computer Graphics, Vol. 22, No. 4, p. 179-188 (1988)

[3] Bergman S., Kaufman A.: BGRAF2: A Real-Time Graphics Language with Modular Objects and Implicit Dynamics. Computer Graphics, Vol. 10, No. 2, p. 133-138 (1976)

[4] Breen David E., Getto Phillip H., Apocada Anthony A., Schmidt Daniel G., Sarachan Brion D.: The Clockworks: An Object-Oriented Computer Animation System. Proc. of Eurographics, p. 275-282 (1987)

[5] Breen David E.: Message-Based Object-Oriented Interaction Modelling. Proc. of Eurographics, p. 489-503 (1989)

[6] Breen David E., Wozny Michael J.: Message Based Choreography for Computer Animation. In: State-of-the-art Computer Animation, p. 69-83 (1989)

[7] Burtnik M., Wein M.: Computer-Generated Key-Frame Animation. Journal of Society for Motion Picture and Television Engeneers, No. 80, p. 149-153 (1971)

[8] Chua T.-S., Wong W.-H., Chu K.-C.: Design and Implementation of the Animation Language SOLAR. In: New Trends in Computer Graphics (Proc. of CG International), p. 15-26 (1988)

[9] Devide, Robert: ANIREALS. Technical Report, Institute of Computer Graphics, Technical University of Vienna (1990)

[10] Devide, Robert: VAST. Diploma Thesis, Institute of Computer Graphics, Technical University of Vienna (1992)

[11] Fiume E., Tsichritzis D., Dami L.: A Temporal Scripting Language for Object-Oriented Animation. Proc. of Eurographics, p. 283-294 (1987)

[12] Gervautz Michael, Devide Robert: VAST - An Integrated Animation System Based on an Actor-Controller Structure. Proc. of Eurographics Workshop on Animation (1993)

[13] Haumann David R., Parent Richard E.: The Behavioral Test-Bed: Obtaining Complex Behavior from Simple Rules. The Visual Computer, Vol. 4, p. 332-347 (1988)

[14] Hewitt: Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models for a robot. Ph.D. Dissertation, MIT (1971)

[15] Ostby Eben F.: Simplified Control of Complex Animation. In: State of the Art in Computer Animation, p. 59-67 (1989)

[16] Plentinckx, D.: The Use of Quaternions for Animation, Modelling and Rendering. Proceedings of CG International '88, p. 44-63 (1988)

[17] Reeves William T.: Particle Systems - A Technique for Modelling a Class of Fuzzy Objects. Computer Graphics, Vol. 17, No. 3, p. 359-376 (1983)

[18] Reeves William T., Ostby Eben F., Leffler Samuel J.: The Menv Modelling and Animation Environment. Journal of Visualization and Computer Animation, Vol. 1, p. 33-40 (1990)

[19] Reynolds Craig W.: Computer Animation with Scripts and actors. Computer Graphics, Vol. 16, No. 3, p. 289-295 (July 1982)

[20] Ridsdale Gary, Calvert Tom: Animating Microworlds from Scripts and Relational Constraints. In: Computer Animation '90, p. 107-117 (1990)

[21] Schmalstieg Dieter: STORYBOARD - A Programming Language for Computer Animation. Diploma Thesis, Institute of Computer Graphics, Technical University of Vienna (1993)

[22] Smith Jeff, Drewery Karin: Timewarps: A Temporal Reparameterization Paradigm for Parametric Animation. Proc. of Eurographics '91, Vienna, p. 413-424 (1991)

[23] Magnenat-Thalmann Nadia, Thalmann Daniel: The Use of High-Level 3-D Graphical Types in the Mira Animation System. IEEE Computer Graphics and Applications, p. 9-16 (1983)

[24] Magnenat-Thalmann Nadia, Thalmann Daniel: CINEMIRA: A 3D Computer Animation Language Based on Actor and Camera Data Types. Technical Report, University of Montreal (1984).

[25] Magnenat-Thalmann Nadia, Thalmann Daniel: Computer Animation - Theory and Practice Springer (1985)

[26] Wilhelms Jane: Dynamics For Computer Graphics: A Tutorial. Computing Systems,Vol. 1,No. 1,p. 63-93 (1988)

[27] Witkin A., Kass M.: Spacetime Constraints. Computer Graphics, Vol. 22, No. 4, p. 159-168 (August 1988)

[28] Zeltzer David: Towards an integrated view of 3-D computer animation. The Visual Computer, Vol. 1, p. 249-259 (1985)