Script Language for Avatar Animation in 3D Virtual Environments

Xiaoli Yang\textsuperscript{1}, Dorina C. Petriu\textsuperscript{2}, Thom E. Whalen\textsuperscript{3}, Emil M. Petriu\textsuperscript{1}

\textsuperscript{1}University of Ottawa, Ottawa, ON, Canada \textsuperscript{2}Carleton University, Ottawa, ON, Canada \textsuperscript{3}Communications Research Centre, Ottawa, ON, Canada

Abstract – This paper continues previous work of the authors on the animation of anthropomorphic avatars in virtual reality environments. An avatar animation control system was built in three hierarchical levels: joint control, basic behaviors and script language. This paper focuses on the grammar and parser for the script language, which was designed to be easy to extend and modify. Also, in order to verify how flexible the hierarchical approach is, the two lower control levels were implemented in two different ways, without affecting the script language level.

1. INTRODUCTION

A three-dimensional (3D) virtual environment provides a new way to describe and explain the objects in the real world. It can be used in a variety of application areas such as engineering, science, education and entertainment. An avatar, which represents a virtual human in the virtual world, plays an important part in any virtual environment.

There is a variety of approaches to avatar animation that attempt to simulate real human beings in 3D virtual environments as closely as possible. The Virtual Reality Modeling Language (VRML) \cite{1},\cite{2},\cite{3}, is frequently used for the definition of the avatars, and more rarely used as the visualization and integration technology. Only some simple and repeatable procedures can be simulated in this way, as VRML itself is not a programming language, and does not give the user many options to interact with the virtual environment.

Avatars can be represented by real-time agents \cite{4}. For the avatar movements, a time series of joint angles are stored so that specific motions can be re-played under real-time constraints \cite{5}. The major advantages of pre-stored motions are fast speed of execution and safer algorithmic security (by minimizing computation). However, the principal disadvantages of pre-stored motion are the lack of generality (since every joint must be controlled explicitly) and anthropometrical extensibility (since changing joint-to-joint distances will change the computed locations of end effectors such as feet, making external constraints and contacts impossible to maintain) \cite{4}.

The Improv System from New York University is mainly controlled by behavioral scripts \cite{6}, \cite{7}, \cite{8}. The behavioral scripts are designed such that they can be translated easily from a given storyboard. In this system, noise techniques are used to change image rendering to character movement. Characters created in this manner can smoothly blend and layer animations. The system is useful for predefined scenarios in the virtual environment. External functions need to be added if it is used in a real-time controlled virtual environment and the characters need to make their decisions often.

ACE (Agent Common Environment) developed by the EPFL computer Graphics Lab is a platform for virtual human agent simulation that is able to coherently manage a shared virtual environment \cite{9}. This system can simulate scenarios in the virtual environment. It provides built-in commands for perception and acting. It also defines an external collection of behavioral plug-ins. The user can use these behaviors to create a story for the virtual environment. However, the user must extend some control functions if he wants to use this system for a specific task.

This paper continues the work presented by the authors in \cite{12}, by describing a three-level hierarchical control system for the animation of avatars that are represented according to the International ISO/IEC Humanoid Standard proposed by the Humanoid Animation Group \cite{11}. The lower level controls the rotation of the avatar’s joints. The second level defines basic behaviors (such as walk, run, jump). The basic behaviors can be combined in sequence or in parallel. The last level of abstraction is represented by a script language, which can be used to describe stories that will be enacted by the avatars. As opposed to \cite{12}, where the emphasis was on the first two control levels, this paper focuses more on the script language grammar and parser. Also, in order to verify how flexible the hierarchical approach is, the two lower control levels were implemented in two different ways without affecting the script language level.
2. AVATAR ANIMATION

VRML is a well-known language for describing interactive 3D objects and worlds. The design of VRML makes it easy to be used on the Internet, intranets, and local client systems [13]. The 3D virtual environment and avatar in this paper are defined in VRML. Fig. 1 shows the avatars controlled by the system described here.

![Avatars Nancy and SphereMan](image)

We chose two example avatars to test our animation control system. They are defined in VRML based on the standard Humanoid – H-Anim body representation [11], and are available on the web. Nancy was created by 3Name3D Company [14], and SphereMan can be found at [15].

According to H-Anim, an avatar is composed of joints and segments. The difference between various avatars defined by the standard consists in the segment shapes and the positions of their joints. We choose to control the joint rotations for animating the avatar, so that the difference among avatars does not influence the control system. The control is not affected by the segment shape or the distance between joints. Therefore, the control system can be reused without any modification for any avatar which is defined according to the H-Anim standard.

As mentioned before, our animation system is composed of three hierarchical levels: joint control, basic behavior control and script language. The design goals were to make the control system easy to develop and extend by keeping a low dependency between levels. The two lower levels, joint control and basic behaviors were implemented in two different ways, as described in the following subsections. Both implementations support the same interface with the script language level, which can run on either one without any change.

2.1. Joint control

The lowest level controls all of the joints of the avatar, like sacroiliac, elbow, wrist or hip and so on. Every joint can rotate in three axes, X, Y and Z. However, every specific joint has its own constraints on each direction due to the human anatomy. For example, the rotation of the hip is restricted to 90 degrees maximum on the Y direction. The following describes two ways to implement this control level.

**VRML/Java Joint Control**

As mentioned above, in the first implementation we chose to develop the control of VRML objects in Java. The communication between the VRML objects and the Java control takes place through the External Authoring Interface (EAI). Java is a programming language able to realize a more flexible and detailed control of the VRML world than VRML itself.

The control code for the rotation of every joint is in a separate Java class, which is running in its own thread of control. This makes it easier to add more joint control classes without affecting the existing code at a higher level of control. The second level of control can use the new added classes for new behaviors, without affecting the behaviors that were defined before. Another reason for using a thread for each joint is to give the impression that the joints are rotating simultaneously.

**Java3D Joint Control**

The second implementation uses Java3D for the control of the avatar. The virtual environment and avatar first defined in VRML, are "loaded" in Java3D by using the so-called file loader CyberVRML97 [16], which reads and writes the VRML files, sets and gets the scene graph information, draws the geometries, and runs the behaviors. Java 3D is a high level scene graph based API [17]. It uses either DirectX or OpenGL low level API to take advantage of the 3D hardware acceleration. This gives the platform-independence one expects from a Java API, along with a high level of performance needed by today's sophisticated 3D applications. The VRML objects were first loaded with CyberVRML97, and then converted into Java3D objects. The Java 3D Transform3D function is used to implement the rotation of the joints.

2.2. Basic behaviors

The role of the second control level is to combine different joint rotations according to a pre-defined schedule in order to create basic behaviors, such as walking or running. We have implemented two kinds of basic behaviors, independent and interactive. Independent behaviors are performed by the avatar alone, without interacting directly with the environment objects. Examples of
independent behavior are walking or waving a hand. The schedules for the joint movements corresponding to three behaviors (“walk”, “run” and “jump”) are taken from the VRML Nancy example given in [14]. Interactive behaviors are those where the avatar interacts with objects from the virtual environment. For example, “turn on/off the computer” is an interactive behavior. In this case, the avatar needs to locate some objects in the virtual environment and interact with them. Since the relative position of the avatar with respect to different objects changes dynamically, we have to obtain the control data at run time. For example, when receiving the command “open the door”, the avatar could be in different places, standing at different angles. In order for the avatar to touch the doorknob, the “inverse kinematics” algorithm is used to get the run time data for the movements of the corresponding joints. The basic behaviors are divided in different groups according to the common joints they are using. Behaviors in the same group are mutually exclusive (such as walk, run and jump), whereas behaviors from different groups can be executed in parallel (such as walk and wave the hand).

We separated the control of different behaviors into different classes, in order to make them easier to use from the higher control level. New classes can be built if we want to add new behaviors, without affecting the existing ones.

By combining the basic behaviors together in sequence or in parallel, the user can get a composed behavior. For example, by combining “go to the door” and “turn the Doorknob”, we can fulfill the behavior of “open the door”.

VRML/Java Basic Behavior Control

Different joint rotations are controlled by different classes, which run in their own thread of control. This produces a smoother avatar animation by giving the appearance that the joints are moving simultaneously. The main disadvantage of this approach is the large communication overhead between VRML and Java. Another disadvantage is the large number of threads necessary for the numerous joints of the avatar. On one hand, the number of threads is limited by the underlying Java virtual machine, on the other hand, multi-threading context switching comes with its own overhead in terms of execution time.

In order to separate the responsibilities related to a behavior, we decided to use three classes working together for each independent behavior. For example, the walking behavior is realized by:

a. WalkData class, which stores the data (i.e., the schedule) for different joint rotations.
b. Walk class, which defines different joint rotation groups at different times.
c. WalkThread class, which controls the timing and coordinates the different rotation groups.

Java3D Basic Behavior Control

After loading the VRML objects into Java3D, we can use special Java3D functions for the avatar control. The main advantage of using Java and Java3D for animation control is platform independence - the code can run anywhere provided that the Java Virtual Machine is supported. Another advantage is that the execution time is much faster than in the VRML/Java case, leaving available CPU time for more complex animations.

In Java3D, there are two special classes, which enhance the perceived quality of the animation, the Alpha and Interpolator. Interpolators are customized behavior objects that use an Alpha object to provide smooth animations of the visual objects [18]. The interpolator defines the end points of the animation. The Alpha controls the animation with respect to the timing, and how the Interpolator will move from one defined point to the other at the times given by the Alpha object. It is obvious just by looking at the animations obtained with the two approaches described here that the one using Java 3D has a higher quality.

To separate the responsibilities related to a behavior, we used two classes working together for each independent behavior. For example, the walking behavior is realized by:

a. WalkData class which stores the data (i.e., the schedule) for different joint rotations.
b. Walk class, which uses the Java3D Alpha and Interpolator objects to control the joint rotations at different times.

2.3. Script language

The third level of control is the script language which is a restricted English-like language described by a context-free grammar. The language has its own interpreter to recognize and execute the commands. The interpreter translates the stories entered by users into a set of commands that represents basic behaviors, and triggers their execution by the avatars. The script language level is dependent only on the interface provided by the basic behavior level, but it is independent of the specific implementation of the two lower levels.

Grammar

Our intention was to design an English-like script language that would allow the user to express easily a short story or a set of commands to be enacted by the avatar. However, it is well known that an unconstrained natural language is difficult to parse, so we decided to settle for a script language defined by a context-free grammar. This type of grammar is
largely used for the definition of today’s programming languages. Efficient parsers can be built for context-free grammars.

The script language defined here is constrained by:

a. The size of its vocabulary (terminal symbols) that depends on the set of basic behaviors implemented in the second level of control.
b. The sentence syntax defined by the grammar.

According to [19], a context-free grammar G is a 4-tuple \((V, \Sigma, R, S)\), where:

- \(V\) is a finite set of nonterminal symbols.
- \(\Sigma\) is a finite set of terminal symbols.
- \(R\) is a finite set of rules that define each nonterminal as a string of symbols (either terminal or nonterminal)
- \(S \in V\) is the set of start symbols.

For the script language grammar defined here, the definitions of \(V\), \(\Sigma\) and \(S\) are the following. The set of rules \(R\) is given in Table I.

\[ V = \{ \text{<story>}, \text{<sentence>}, \text{<composed sentence>}, \text{<simple sentence>}, \text{<subject>}, \text{<predicate>}, \text{<direct object>}, \text{<noun>}, \text{<indirect object>} \} \]

\[ \Sigma = \{ \text{<Nancy>}, \text{<she>}, \text{<stand>}, \text{<wait>}, \text{<run>}, \text{<jump>}, \text{<open>}, \text{<wave>}, \text{<kick>}, \text{<walk>}, \text{<go>}, \text{<turn>}, \text{<left>}, \text{<right>}, \text{<door>}, \text{<arm>}, \text{<leg>}, \text{<head>}, \text{<to>}, \text{<on>}, \text{<yardDoor>}, \text{<table>}, \text{<window>}, \text{<tree>}, \text{<chair>}, \text{<light>}, \text{<computer>}, \text{<fast>}, \text{<faster>}, \text{<slow>}, \text{<slower>} \} \]

\[ S = \text{<story>} \]

How can a string of terminal symbols be generated from the rules? We can begin with the start symbol and choose a rule to apply. If the resulting string contains nonterminals, we choose rules to apply to each of those nonterminals. Each time we apply a rule to a nonterminal, we replace that nonterminal in the current string by the right-hand side of the rule applied. We continue making replacements according to the rules until, eventually, the current string contains only terminals [19].

The following convention was adopted for the composition of basic behaviors, each one represented by a simple sentence:

a. Sequential composition: behaviors separated by "," (i.e. separate sentences)
b. Parallel composition: behaviors separated by "." or "and" (i.e. simple sentences in a composed sentence)

This approach aims for maximum parallelism between the behaviors expressed in the same composed sentence.

For example, the parse tree for the following sentence is given in Fig. 2:

"Nancy turns to the yardDoor. She walks fast, waves the left hand."

### Table I. Grammar for the script language: Set of rules

<table>
<thead>
<tr>
<th>Left-hand side</th>
<th>Right-hand side</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;story&gt; ::= &lt;sentence&gt;</td>
<td>&lt;&lt;sentence&gt;&lt;story&gt;</td>
</tr>
<tr>
<td>&lt;sentence&gt; ::= &lt;simple sentence&gt;</td>
<td>:=&quot;&lt;simple sentence&gt;&lt;subject&gt; &lt;predicat&quot; &quot;&quot;</td>
</tr>
<tr>
<td>&lt;composed sentence&gt; ::= &lt;simple sentence&gt;&lt;separator&gt; &lt;composed sentence&gt;</td>
<td>&lt;&lt;simple sentence&gt;</td>
</tr>
<tr>
<td>&lt;separator&gt; ::=&quot;,&quot;&lt;conjunction&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;conjunction&gt; ::= &quot;and&quot;</td>
<td></td>
</tr>
<tr>
<td>&lt;simple sentence&gt; ::= [&lt;subject&gt;] &lt;predicate&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;predicate&gt; ::= &lt;verb&gt; [&lt;adverb&gt;] [&lt;direct object&gt;] [&lt;indirect object&gt;]</td>
<td></td>
</tr>
<tr>
<td>&lt;direct object&gt; ::= [&lt;article&gt;] [&lt;adjective&gt;] &lt;noun&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;noun&gt; ::= [&lt;article&gt;] [&lt;noun&gt;]</td>
<td></td>
</tr>
<tr>
<td>&lt;indirect object&gt; ::= [&lt;article&gt;] [&lt;adjective&gt;] [&lt;noun&gt;]</td>
<td></td>
</tr>
<tr>
<td>&lt;subject&gt; ::= &quot;Nancy&quot;</td>
<td>['she']</td>
</tr>
<tr>
<td>&lt;verb&gt; ::= &quot;stand&quot;</td>
<td>['wait']</td>
</tr>
<tr>
<td>&lt;adjective&gt; ::= &quot;left&quot;</td>
<td>['right']</td>
</tr>
<tr>
<td>&lt;noun&gt; ::= &quot;door&quot;</td>
<td>['arm']</td>
</tr>
<tr>
<td>&lt;adverb&gt; ::= &quot;fast&quot;</td>
<td>['faster']</td>
</tr>
<tr>
<td>&lt;article&gt; ::= &quot;a&quot;</td>
<td>['the']</td>
</tr>
<tr>
<td>&lt;preposition&gt; ::= &quot;to&quot;</td>
<td>['on']</td>
</tr>
</tbody>
</table>

### Interpreter

The role of the interpreter is to parse the input string in order to identify the sentences corresponding to different basic behaviors, then to invoke the appropriate methods and threads implemented in the second control level for executing them.

A typical top-down parser without backtracking was built for the script language grammar by using recursive methods for all the rules containing nonterminals. If such a rule has multiple alternatives, we look ahead in the input string until we can decide which alternative is the right one for that case. Thus, backtracking is avoided.

The reasons for this design choice are multiple:

1. Accelerate the search among multiple alternatives.
2. Make the parser easier to extend when new terminals are added to the grammar.
3. Simplify the selections of the right sentence structure (some verbs can be followed by a direct object, others by an indirect object, etc).
4. Point to the method or thread that executes the respective actions.
When designing the script language, we decided to make it less like a programming language and more like English. Therefore, no quantitative parameters for different behaviors are expressed in the script language (such as the exact duration of a step, the number of steps or angles for turning, etc). In order to allow the user to control to some degree these parameters, we took the following approach:

a. If no adverb is given in a sentence, use pre-defined default values. For example, "Nancy walks" will be translated into a walk command that corresponds to a "normal" walk (the actual parameter values are: number of steps = 3; step duration = 1s).

b. If the user wants to change some of the movement characteristics, he/she can use adverbs such as "fast" or "slow" to increase or decrease the speed of the movement to some pre-set values. For example, "Nancy walks fast" will be translated into a rapid walk (the step duration parameter is now 0.9s).

c. Another possibility is to use comparative adverbs, such as "faster", to change the speed by a pre-defined increment. By repeating the command "walk faster" several times, the avatar walking speed can be raised up to the level desired by the user.

The approach of using adverbs for characterizing the movements instead of giving quantitative values has the advantage of being simpler for a human user. However, it has the disadvantage of being less accurate than in the case of using numerical parameters. More experiments are necessary to judge the effectiveness of this approach, and maybe to refine it by introducing more pre-defined levels (such as "very fast", "fast", "slow", "very slow") for a better accuracy.

One of the design goals was to make the interpreter easy to modify when new avatar behaviors will be defined in the second level of control, or new objects will be introduced in the virtual environment. This means that the set \( \Sigma \) of terminals will be extended with new verbs, nouns, etc. If the new behaviors can be expressed with the same sentence syntax as defined by the grammar \( G \), then the code of the interpreter does not need to be changed, only the linked table. If, however, the syntax of the script language needs to be extended with new sentence structures, then the code of the interpreter will have to be modified, as well.

An example of using the script language for avatar animation is the following "story" entered in the appropriate field at the bottom of the user interface: 

"Nancy walks. She turns to the tree. She runs. Nancy goes faster to the yardDoor, turns the head to the door and waves the right arm. She turns right. Walks longer. She goes to the door, turns the head to the window and waves the left arm. Nancy opens the door. Turns the head to the light and turns on the light."

In Fig. 3 Nancy is executing the first sentence "Nancy walks", and in Fig. 4 the command "turns on the light".
3. Conclusions and future work

We described the design and implementations of a three-level control system for avatar animation in a virtual environment. The lower level controls the joint rotations; the middle level controls the basic behaviors and their composition, and the higher level implements and executes a script language. We used two different approaches to implement the first and second level of control.

This hierarchical approach, inspired from the field of robotics, has the following advantages:

a. decoupling between different control levels
b. flexibility and extensibility
c. user friendliness, as the complexity of the lower levels is hidden from the user.

In the future, more functions and behaviors can be added to the avatar. More artificial intelligence can be used to enable the avatar to make decisions in impromptu situations without the interference of human beings. Another big extension is to introduce more avatars into the virtual environment and to develop collaborative interactions among avatars. Speech and text can be also added to enrich the whole activities in the virtual environment. Different behaviors can be integrated into models or plug-ins that can be used in other software applications, such as game engines, educational applications, weather forecast or news report. All these extensions will have an impact on the script language for avatar animation. The language will become more complex, and maybe closer to English than it is now.

ACKNOWLEDGMENTS

This work was funded in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the National Capital Institute of Telecommunications (NCIT), and the Communications Research Centre (CRC) Canada.

REFERENCES