Building Interactive Animations using VRML and Java

Fabiana Saldanha Tamiosso[1], Alberto Barbosa Raposo[1],
Léo Pini Magalhães[1,2], Ivan Luiz Marques Ricarte[1]

[1] State University of Campinas (UNICAMP)
School of Electrical and Computer Engineering (FEEC)
Dept. of Computer Engineering and Industrial Automation (DCA)
CP 6101 – 13083-970 – Campinas, SP, Brazil
Phone: +55-19-239-8385 – Fax: +55-19-239-1395
`fabiana, alberto, leopini, ricarte@dca.fee.unicamp.br`
[2] University of Waterloo – Computer Science Dept. – Computer Graphics Lab.
200 University Ave. W Waterloo, On N2L 3G1, Canada
Phone: +1-519-885-1211 ext. 2041 – Fax: +1-519-885-1208
`lpini@cgl.uwaterloo.ca`

Abstract. This paper exploits the combination of VRML (Virtual Reality Modeling Language) and Java for the construction of highly interactive animations, whose behavior is defined in real-time by user's actions. The animations are modeled in VRML, which allows the definition of a Java program to process and generate events that determine the behavior of scene elements. An application for the generation of Java graphical interfaces was developed, aiming to establish the communication between the user and the VRML environment, sending parameters to the program that controls the animation.

1 Introduction

A scripting environment general enough to include all the animation paradigms (e.g., keyframe, kinematics, dynamics, and behavioral) [1, 10], presenting real-time characteristics, powerful enough to support both user interaction and the construction of graphical interfaces has always been a necessity in the field of Computer Modeled Animation. However, such environment has not been created yet.

Nevertheless, the conjunction of VRML (Virtual Reality Modeling Language) and Java is a step towards all these needs. VRML [11] is a file format to describe rich 3D environments that enable the user to interact with them. A custom protocol to communicate with Java programs is one of the most important features of VRML regarding animation development.

Java [2] is a platform-independent, object-oriented language, which can be used to control the movements in a VRML scene, using any animation technique. As a general-purpose language, Java can also be used to construct user interfaces.

Real-time presentation of animations, although presently at the cost of rendering quality, is guaranteed by efficient VRML browsers, like Cosmo Player [8], Community Place [9], and Liquid Reality [3], among others. Current browsers may execute as built-in tools for Web browsers and as autonomous tools.

In this paper we present strategies we have used to achieve the integration of VRML and the Java user interface, in order to develop highly interactive computer modeled animations.

In the next section we introduce the VRML features. In Section 3 we give details about its integration with Java and present techniques and facilities we have developed to enable the user to have a tight control over the animation. In Section 4 we show some examples of interactive animations using VRML and Java, developed following different animation paradigms. Our conclusions are presented in Section 5.

2 VRML

The main goal of the current version of VRML – 2.0 – is to provide a rich 3D interactive graphical environment, allowing the user to define static and animated worlds, and to interact with them [6, 11].

The improvements of this version of VRML include: development of more realistic scenes, prototyping (capacity to encapsulate nodes to create a new one), direct interaction with the user via sensors, interpolators, and creation of "intelligent" animations using scripts.

The paradigm to produce VRML scenes is based on *nodes* defining a scene graph. Each node defines a name, a type, and default values for its parameters. There are two kinds of parameters: *fields* and *events*. *Fields* can be called simply "fields" (private) or "exposedFields" (public). *Events* can be sent from a node to another by an "eventOut" parameter and received by an "eventIn". This can also be done by exposedFields (exposedField

= eventIn + field + eventOut). Events signalize changes caused by external stimuli and can be propagated by the nodes using *Routes*, which connect an eventOut to an eventIn of the same type (see Figure 1). Events and Routes drive the animation of the worlds.
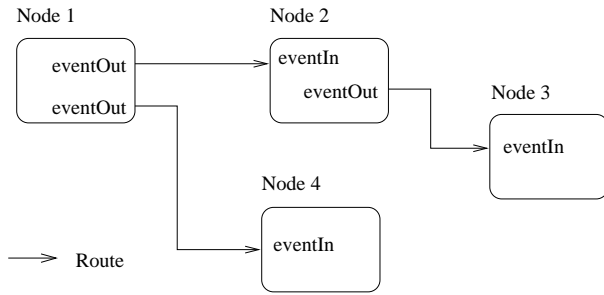

Figure 1: Nodes connected by routed events.

There are many kinds of nodes in VRML. For instance, we may cite *geometric nodes* (that define geometric objects), *illumination nodes*, *grouping nodes* (that group children nodes and other grouping nodes, causing the group to exhibit common behavior and defining the hierarchical structure of a VRML file) and, particularly important for animated worlds, *sensor nodes* and *interpolator nodes*.

Sensor nodes generate events based on user actions; the TouchSensor node, for example, detects when the user clicks the mouse over a specified object (or group of objects), and generates an eventOut. This eventOut may be routed to other eventIn(s), and cause the start of an animation. Sensors are responsible for the user interaction in VRML. Furthermore, they are not restricted to generate events based on user actions; the TimeSensor, for instance, automatically generates an event at each tick of the clock, being normally used as the animation clock.

Interpolator nodes define the keyframes of an animation, interpolated by a linear function. An example is the PositionInterpolator node, where the user defines n key positions and n time instants (each instant associated to a key position). Used in conjunction with a TimeSensor, the PositionInterpolator generates an event at each clock tick, representing the current position, resulted from the interpolation function.

The events generated by sensor and interpolator nodes routed to geometric, illumination, and grouping nodes may define interesting keyframe animations. A typical example is illustrated in Figure 2. In this example, a TouchSensor is linked to a TimeSensor, meaning that the user's click over a certain object starts the clock. The TimeSensor is linked to a PositionInterpolator, sending time values for the interpolation function. The interpolator is finally linked to a geometric node, defining, at

each instant, the new position of the object.

Nevertheless, the routing approach is limited, since it does not allow the handling of a whole class of behaviors that depends on logic operations (e.g., it is not possible to choose between two predefined trajectories) [5]. VRML overcomes this limitation defining a special node, called *Script*, that allows the user to connect the animation to a program, where the events can be processed. This program can be written in any programming language, but we will consider only Java, since it is presently by far the most used one.

3 Controlling Animation Behavior in VRML with Java

Script nodes, via their associated programs, bring decision logic and state management into VRML. Animations modeled with VRML can use a Java program to determine the trajectory to be followed, for example. This program is capable of receiving, processing and generating events that control the animation behavior.

By the use of Script nodes, it is possible to model animations using more complex techniques than keyframing. For example, the time generated by a TimeSensor may be routed to a Script node associated to a program that calculates the position of an object based on a kinematic (or dynamic) function $f(t)$. The new position calculated by the program may then be sent as an eventOut from the Script node and routed to a geometric node, whose associated object will move according to $f(t)$. This kind of VRML and Java integration is illustrated in Figure 3.
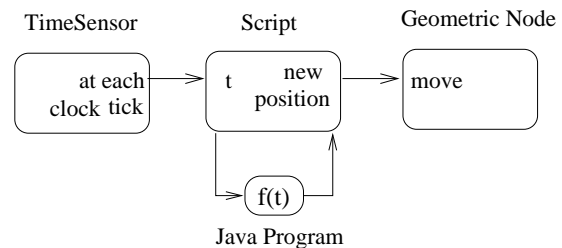

Figure 3: Events routing in an animation with the Script node.

The combination of VRML and Java is a very powerful tool for the control of animations behavior. Interactive animations that have their behavior defined by user actions can be modeled with VRML using a Script node and its associated program, which is capable of receiving parameters from another Java application, responsible for the user interface. We have developed an interaction tool to enable the creation of a "control panel" for the animation, in which the user defines the values of parameters
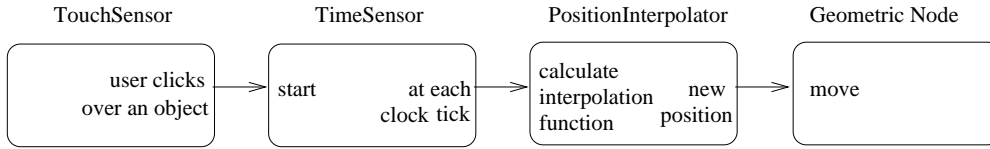
Figure 2: Events routing in a typical keyframe animation.

that control the animation in real-time (e.g., the user may alter the velocity or acceleration of a certain object in a kinematic animation).

3.1 IUlib: an Application for User Interface Support

In order to facilitate the creation of the user interface and enable its connection to the program associated to the Script node (from now on, we will call it "Script program"), we have implemented an application called IUlib. Using this application, it is possible to easily construct the graphical interface, responsible for sending animation control parameters to the VRML file.

The IUlib generates a Java source file (.java) including a class responsible for all the treatment necessary for the graphical components of the user interface and another one, responsible for the data transmission to the VRML file, via the Script program. This Java source file, after compilation, generates an executable class (the graphical interface) defining methods that allow the Script program to obtain the user's data. The class responsible for the data transmission uses methods of the other class (the one responsible for the user interface) to read such data, since this second class determines which graphical components are associated to animation control parameters.

The transmission methods are based on standard Java packages (Java.net and Java.io). The Script program should be able to receive the data, using methods of the same packages.

In our scenario, the Script program receives data from the user interface. The communication between this interface and the Script program is based on the client-server model. The user interface is the server, and the Script program is the client. The VRML file starts the Script program, that connects to the interface, requesting data from the server. The interface then sends the requested data, the Script program processes them and returns the results to the VRML file. This approach is schematized in Figure 4.

Both the server and the client are defined to be executed in the user's host machine, on a specified port. These definitions are included in a shell script, becoming transparent for the user.

The communication between the VRML file and the

Script program follows the standards proposed in the VRML specification [11]. For that reason, we will not detail it in this paper.

The connection between the Script program and the user interface is responsible for the transmission of the animation control parameters defined at real-time by the user.

The construction of the user interface using the IUlib is accomplished by combining several graphical components defined by the Java language, such as Labels, Buttons, Scrollbars, etc. The first window that appears during this construction process asks the name of the Java program that is going to be generated. The second window (see Figure 5) shows the options of graphical components that can be included in the user interface. For each option chosen, a new window appears. Figure 6 shows the window for the definition of a scrollbar. It is possible to define maximum and minimum values for the Scrollbar, its initial value, its orientation (horizontal or vertical), among other parameters defined by the Java standard libraries. Since the Scrollbar represents a numerical value, this value can be associated to an animation control parameter and sent to the VRML process.

Figure 5: IUlib window: choosing graphical components for the user interface.
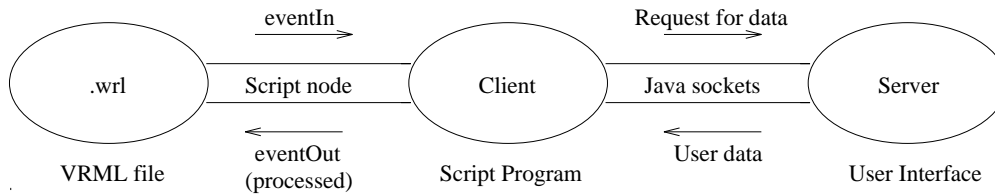
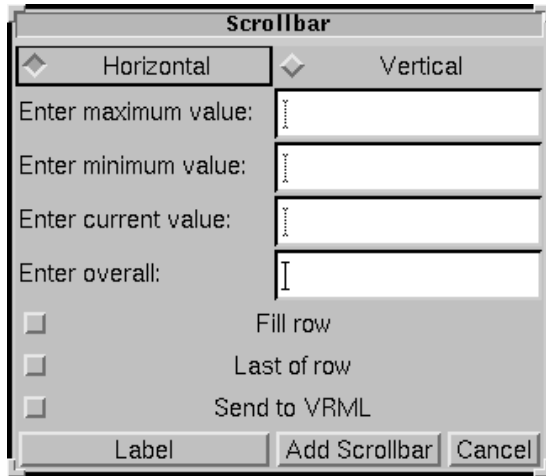Figure 4: Communication schema in our scenario.

Figure 6: IUlib window: including a Scrollbar to an animation control parameter.

The animation control parameters can be associated to Scrollbars, CheckboxGroups, and TextFields, selecting the option *Send to VRML* in the corresponding window. This option creates the communication methods necessary to send the parameter's value to the Script program. These communication methods simply define that the specified value will be sent to the client when it requests an update of its control parameters (this update can happen at each clock tick, at each animation cycle, and so on, depending on the implementation).

The graphical components appear in the user interface and are sent to the client in the same order they have been created.

In the next section we will present some examples of interactive VRML animations using interfaces and communication facilities developed with the IUlib.

4   Examples

In this section we present three VRML animation examples using the Script node to process events. The first one is a keyframe animation, whose behavior is defined by user actions applied directly in the scene (i.e., it does not

use an external user interface). The following examples show similar animations implemented according to more sophisticated paradigms (kinematics and dynamics) using external interfaces to send control parameters.

The examples are platform-independent. They are visualized with a simple browser implemented with methods of Liquid Reality [3], a set of Java class libraries providing the functionality needed to write out, to render, and to manipulate a VRML scene. IUlib was used to implement the user interfaces.

4.1   Keyframe Animation

Using the keyframe paradigm, the animator predefines the initial and final frames of an animation sequence (the key frames) and an interpolation function or a set of interpolated values responsible for the transformations between those frames. Time and number of frames in this case play similar roles.

In the animation shown in Figure 7, a sphere can follows two elliptical trajectories, depending on which "button" is chosen (the "buttons" are represented by the two-colored cube at the right side). If the user clicks on the left button, the sphere will follow an internal trajectory, and if he/she clicks on the right one, the sphere will follow an external trajectory.

In the VRML file, the trajectories are defined by two PositionInterpolator nodes, the buttons are TouchSensor nodes, and a Script node is used to recognize which sensor has been touched by the user and to choose the correct trajectory for the sphere.

4.2   Kinematic Animation

The previous example was adapted to use a kinematic model, in which the animator defines the movement using an equation (or an equation system) with kinematic variables (e.g., objects position, velocity, acceleration) as a function of time. If the model is appropriately defined, movements more realistic than those using interpolation can be obtained.

The elliptical trajectory of the sphere is calculated according to the following equations:
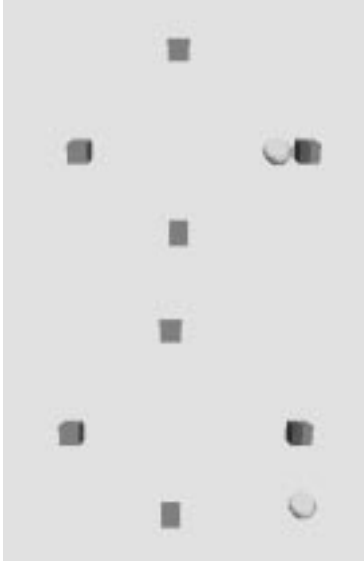
Figure 7: Keyframe animation (in the upper frame the sphere follows an internal trajectory, and in the other frame it follows an external one).

$$
\begin{aligned}
x &= a \cdot cos\theta \\
y &= b \cdot sin\theta \\
z &= 0 \\
\theta &= \theta_o + \omega t + \gamma \frac{t^2}{2}
\end{aligned}
$$

where (x, y, z) represents the spatial position of the sphere; a and b are the size of the ellipsis' horizontal and vertical semi-axis, respectively; ω is the angular velocity; γ is the angular acceleration; and t is the time.

The VRML implementation of this animation follows the strategy presented in Figure 3. A TimeSensor continuously generates t values and sends them to the Script node, that is associated to a program that calculates the positions of the sphere according to the previous equations.

Using an external interface constructed with IUlib, the user can alter the trajectory to be followed, changing a and b values. In addition, it is also possible to control the movement's characteristics altering the values of ω and γ.

The user interface that controls these parameters and the animation developed are shown in Figure 8. The interface is composed of Labels, Scrollbars, and CheckboxGroups. Each Scrollbar is associated to a CheckboxGroup. The first one is related to the ellipsis' semi-axis (a and b) and the other is related to the kinematic parameters (ω and γ). The alterations in the Scrollbar change the value of the parameter associated to the selected Checkbox. These values are transmitted to the Script program, that calculates the trajectory, which is immediately modified.

Figure 8: Kinematic animation controlled via a user interface.

4.3   Dynamic Animation

Dynamic animations define the movements using an equation (or an equation system) with dynamic variables (e.g., mass, force) as a function of time. If the model is well designed, more realistic movements can be achieved than by the last two techniques. On the other hand, modeling is more complex and the movement's control using dynamic variables is not "natural" for the animator [1].

The dynamic simulation of an elliptical trajectory is based on the Kepler's law for the attraction between two celestial bodies (a planet follows an elliptical trajectory around the Sun, located on the ellipsis' focus).

The mathematical development of this model can be found in [4], being out of the scope of this article. The resulting equations are:

$$
\begin{aligned}
t &= \sqrt{\frac{a^3}{G \cdot (m_1 + m_2)}} \cdot (E - \epsilon \cdot sinE) \\
E &= arcsin(\frac{\sqrt{1 - \epsilon^2} \cdot sin\theta}{1 + \epsilon \cdot cos\theta})
\end{aligned}
$$

where t is the time; a is the ellipsis' horizontal semi-axis; ϵ is the ellipsis' eccentricity (relation between the distance from the center to the focus and the semi-axis a — $0 \le \epsilon < 1$); G is the universal gravity constant; m_1

and m_2 are the masses of the bodies; θ is the angle that defines the spatial position (x, y, z) of the moving body; and E is another parameter related to the position of the same body.

In order to determine the position as a function of time, we should isolate E in the first equation, which is an ordinary equation. To compute the complex solution for this equation it would require a long processing time, not adequate for interactive animations. However, we found an alternative solution suitable to our animation purposes, based on the construction of a table that relates time values to regular intervals of θ. θ is varied in intervals of 2 degrees and the corresponding times are calculated; the time received from the TimeSensor is then compared to the values of the table to determine the current position of the body.

The position of the body is determined from θ based on the following equations:

$$
\begin{aligned}
x &= r \cdot cos\theta + a \cdot \epsilon \\
y &= r \cdot sin\theta \\
z &= 0 \\
r &= \frac{a \cdot (1 - \epsilon^2)}{1 + \epsilon \cdot cos\theta}
\end{aligned}
$$

The construction of the table relating θ to t requires a relatively long processing time, when compared to an animation cycle interval. For this reason, a new table is constructed only when the user has changed at least one control value and the body is passing in the initial position. Therefore, the effects of a user alteration in a control parameter will be noticed only in the following cycle.

The user's control parameters for this dynamic animation are a, ϵ, m_1, and m_2. However, we are dealing with celestial bodies, and small alterations in the masses will not have perceptible effect on the movement. For this reason, we added another control parameter, the exponent of the mass, enabling larger variations on the masses of the bodies. The user interface for this animation, built with IUlib as in the previous example, is shown in Figure 9.

5 Conclusions

VRML enables the development of interactive animations by the use of sensor nodes and the routing of events through the nodes. The conjunction of VRML and Java through the Script node enables the definition of more complex animations.

In this paper, we contributed to achieve a better interaction with the user, by the creation of graphical interfaces (Java applications) that send animation control parameters to the Script program.

A large amount of time would be spent by the animator for the creation of this interface. To facilitate the
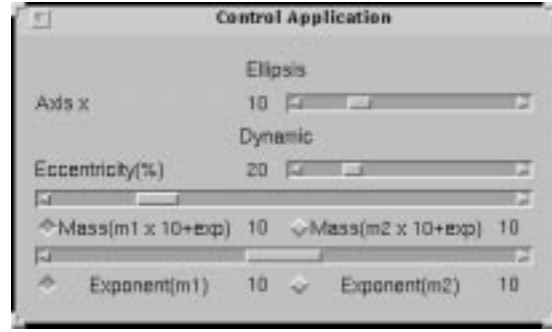

Figure 9: User interface for the dynamic animation.

animator's work, we have developed IUlib, a tool to easily and rapidly construct the graphical user interface for control and communication with the animation, remaining for the user only the development of the Script program.

The IUlib allows the animator to create any type of communication interface to VRML, using several types of graphical components. This communication, in our case, is used for animation control, but it can be reused with other goals (e.g., definition of colors of geometric objects). Our objective is to make the user interface reusable also for other kinds of applications, offering facilities for the user via a VRML extension (prototyping).

In the next steps of this work, we plan to facilitate even more the creation of interactive VRML animations, working on the definition of the Script program, regarding the communication with the user interface and the definition of movements (the development of the Script program can be supported by a script editor, like the one presented in [7]). We plan to develop a library containing different kinds of movements (generated by the various animation paradigms) that can directly receive the parameters from the user interface. The user should then only choose a movement among the library's predefined movements and control its parameters via the interface.

Interactive VRML animations, like those presented in this work, can be used in a variety of applications such as distance learning, scientific simulation, cooperative environment, among others.

Furthermore, the combination of VRML and Java is not restricted to the control of animation behaviors. As a general-purpose language, Java can be used to connect a VRML scene to the external world (outside the VRML file). As stated by Lea [5], VRML applications could query databases and display results, display disk contents, access the Web, and interact with users, via menus, buttons, and so on, opening several other possibilities for the improvement of this work.

References

[1] J. T. F. Camargo, L. P. Magalhães, and A. B. Raposo. *Fundamentos da Animação Modelada por Computador*, 1995. Tutorial presented at VIII SIBGRAPI (`http://www.dca.fee.unicamp.br/projects/prosim/publiPS.html`).

[2] M. Campione and K. Walrath. *The Java Tutorial - Object-Oriented Programming for the Internet*. The Java Series. Addison–Wesley, 1997. Online version: `http://java.sun.com/nav/read/Tutorial/index.html`.

[3] Dimension X. *Liquid Reality*. (`http://www.microsoft.com/java/gallery/lrpro.htm`).

[4] D. T. Greenwood. *Principles of Dynamics*. Prentice-Hall, Inc., 1965.

[5] R. Lea. Java and VRML 2.0 Part 1: Basic Theory. *VRML Site Magazine*, February 1997. (`http://www.vrmlsite.com/feb97/a.cgi/spot2.html`).

[6] L. P. Magalhães, A. B. Raposo, and F. S. Tamiosso. *VRML 2.0 — An Introductory View by Examples*, 1997. Online tutorial: `http://www.cgl.uwaterloo.ca/~lpini/tutorial/vrml-tut.html`.

[7] A. B. Raposo. *Um Sistema Interativo de Animação no Contexto ProSIm*. Master Thesis, DCA – FEEC – UNICAMP, 1996.

[8] Silicon Graphics, Inc. *Cosmo Player*, 1997. (`http://vrml.sgi.com/cosmoplayer/`).

[9] Sony Corporation. *Community Place*, 1997. (`http://vs.spiw.com/vs/`).

[10] N. M. Thalmann and D. Thalmann. *Computer Animation - Theory and Practice*. Springer-Verlag, 1985.

[11] VRML Consortium. *The Virtual Reality Modeling Language Specification ISO/IEC DIS 14772-1*, April 1997. (`http://www.vrml.org/Specifications/VRML97/DIS/`).