/ 995 // 33 47

N95- 19765

/ - /

34101

p- 9

Using CLIPS to Represent Knowledge in a VR Simulation

Mark Engelberg
LinCom Corporation
mle@gothamcity.jsc.nasa.gov

September 13, 1994

Abstract

Virtual reality (VR) is an exciting use of advanced hardware and software techonologies to achieve an immersive simulation. Until recently, the majority of virtual environments were merely "fly-throughs" in which a user could freely explore a 3-dimensional world or a visualized dataset. Now that the underlying technologies are reaching a level of maturity, programmers are seeking ways to increase the complexity and interactivity of immersive simulations. In most cases, interactivity in a virtual environment can be specified in the form "whenever such-and-such happens to object X, it reacts in the following manner." CLIPS and COOL provide a simple and elegant framework for representing this knowledge-base in an efficient manner that can be extended incrementally. The complexity of a detailed simulation becomes more manageable when the control flow is governed by CLIPS' rule-based inference engine as opposed to by traditional procedural mechanisms. Examples in this paper will illustrate an effective way to represent VR information in CLIPS, and to tie this knowledge base to the input and output C routines of a typical virtual environment.

1 Background Information

1.1 Virtual Reality

A virtual experience, or more precisely, a sense of immersion in a computer simulation, can be achieved with the use of specialized input/output devices. The head-mounted display (HMD) is perhaps the interface that most characterizes virtual reality. Two small screens, mounted close to the user's eyes, block out the real world, and provide the user with a three-dimensional view of the computer model. Many HMDs are mounted in helmets which also contain stereo headphones, so as to create the illusion of aural, as well as visual immersion in the virtual environment. Tracking technologies permit the computer to read the position and angle of the user's head, and the scene is recalculated accordingly (ideally at a rate of thirty times a second or faster). [1]

There are many types of hardware devices which allow a user to interact with a virtual environment. At a minimum, the user must be able to navigate through the envrionment. The ability to perform actions or select objects in the environment is also critical to making a virtual environment truly interactive. One popular input device is the DataGlove which enables the user to specify actions and objects through gestures. Joysticks and several variants are also popular navigational devices.

1.1.1 Training

Virtual reality promises to have a tremendous impact on the way that training is done, particularly in areas where hands-on training is costly or dangerous. Training for surgery, space missions, and combat all fall into this category; these fields have already benefitted from existing simulation technologies [2]. As Joseph Psotka explains, "Virtual reality offers training as experience" [3, p. 96].

1.1.2 Current Obstacles

VR hardware is progressing at an astonishing rate. The price of HMDs and graphics workstations continues to fall as the capabilities of the equipment increase. Recent surveys of the literature have concluded that the biggest obstacle right now in creating complex virtual environments is the software tools. One study said that creating virtual environments was "much too hard, and it took too much handcrafting and special-casing due to low-level tools" [5, p. 6].

VR developers have suffered from a lack of focus on providing interactivity and intelligence in virtual environments. Researchers have been "most concerned with hardware, device drivers and low-level support libraries, and human factors and perception" [5, p. 6]. As a result,

> Additional research is needed to blend multimodal display, multisensory output, multimodal data input, the ability to abstract and expound (intelligent agent), and the ability to incorporate human intelligence to improve simulations of artifical environments. Also lacking are the theoretical and engineering methodologies generally related to software engineering and software engineering environments for computer-aided virtual world design. [4, p. 10]

1.2 CLIPS

VR programmers need a high-level interaction language that is object-oriented because "virtual environments have potentially many independent but interacting objects with complex behavior that must be simulated" [5, p. 6]. But unlike typical object-oriented systems, there must be "objects that have time-varying behavior, such as being able to execute Newtonian physics or *systems of rules*" [5, p. 7].

CLIPS 6.0 can fill this need for a high-level tool to program the interactions of a virtual environment. COOL provides the object-oriented approach to managing the large number of independent objects in complex virtual environments, and CLIPS 6.0 provides the ability to construct rules which rely on pattern-matching on these objects.

CLIPS is a particularly attractive option for VR training applications because many knowledge-bases for training are already implemented using CLIPS. If the knowledge-base about how different objects act and interact in a virtual environment is implemented in the same language as the knowledge-base containing expert knowledge about how to solve tasks within the environment, then needless programming effort can be saved.

2 Linking CLIPS with a Low-level VR Library

2.1 Data Structures

Every VR library must use some sort of data structure in order to store all relevant positional and rotational information of each object in a virtual environment. These structures are often called *nodes* and nodes are often linked hierarchically in a tree structure known as a *scene*. In order to write CLIPS rules about virtual objects, it is necessary to load all the scene information into COOL.

The following NODE class has slots for a parent node, children nodes, x-, y-, and z-coordinates, and rotational angles about the x-, y-, and z-axis.

```
(defclass NODE
  (is-a USER)
  (role concrete)

  (slot node-index (visibility public)
        (create-accessor read-write) (type INTEGER))

  (slot parent (visibility public)
        (create-accessor read-write) (type INSTANCE-NAME))
  (multislot children (visibility public)
             (create-accessor read-write) (type INSTANCE-NAME))

  (slot x (visibility public) (create-accessor read) (type FLOAT))
  (slot y (visibility public) (create-accessor read) (type FLOAT))
  (slot z (visibility public) (create-accessor read) (type FLOAT))
  (slot xrot (visibility public) (create-accessor read) (type FLOAT))
  (slot yrot (visibility public) (create-accessor read) (type FLOAT))
  (slot zrot (visibility public) (create-accessor read) (type FLOAT)))
```

It is a trivial matter to write a converter which generates NODE instances from a scene file. For example, a typical scene might convert to:

```
(definstances tree
(BODY of NODE)
(HEAD of NODE)
; and so on
)

(modify-instance [BODY]
        (x 800.000000)
        (y -50.000000)
        (z 280.000000)
        (xrot 180.000000)
        (yrot 0.000000)
        (zrot 180.000000)
        (parent [REF])
        (children [HEAD] [Rpalm] [Chair]))
; and so on
```

2.2 Linking the Databases

Note that the `NODE` class has no default write accessors associated with slots `x`, `y`, `z`, `xrot`, `yrot`, or `zrot`. Throughout the VR simulation, the `NODE` instances must always reflect the values of the node structures stored in the VR library, and vice versa. To do this, the default write accessors are replaced by accessors which call a user-defined function to write the value to the corresponding VR data structure immediately after writing the value to the slot. Similarly, all of the VR functions must be slightly modified so that any change to a scene node is automatically transmitted to the slots in its corresponding `NODE` instance.

The `node-index` slot of the `NODE` class is used to give each instance a unique identifying integer which serves as an index into the C array of VR node structures. This helps establish the one-to-one correspondence between the two databases.

2.3 Motion Interaction

A one-to-one correspondence between the database used internally by the VR library and COOL objects permits many types of interactions to be easily programmed from within CLIPS, particularly those dealing with motion. The following example illustrates how CLIPS can pattern-match and change slots, thus affecting the VR simulation.

```
(defrule rabbit-scared-of-blue-carrot
  (object (name [RABBIT]) (x ?r))
  (object (name [CARROT]) (x ?c&:(< ?c (+ ?r 5))&:(> ?c (- ?r 5))))
  =>
  (send [RABBIT] put-x (- ?r 30)))
```

Assuming the rabbit and blue carrot can only move along the x-axis, this rule can be paraphrased as "whenever the blue carrot gets within 5 inches of the rabbit, the rabbit runs 30 inches away."

2.4 The Simulation Loop

Most interactivity in virtual environments can be almost entirely specified by rules analogous the above example. A simulation then consists of the following cycle repeated over and over:

1. The low-level VR function which reads the input devices is invoked.
2. The new data is automatically passed to the corresponding nodes in COOL, as described in Section 2.1.
3. CLIPS is run, and any rules which were activated by the new data are executed.
4. The execution of these rules in turn triggers other rules in a domino-like effect until all relevant rules for this cycle have been fired.
5. The VR library renders the new scene from its internal database (which reflects all the new changes caused by CLIPS rules), and outputs this image to the user's HMD.

While the low-level routines still provide the means for reading the input and generating the output of VR, the heart of the simulation loop is the execution of the CLIPS rule-base. This provides an elegant means for incrementally increasing the complexity of the simulation; best of all, CLIPS' use of the Rete algorithm means that only relevant rules will be considered on each cycle of execution. This can be a tremendous advantage in complex simulations.

2.5 More VR functions

There are some VR interactions that cannot be accomplished solely through motion. Fortunately, CLIPS provides the ability to create user-defined functions. This allows the programmer to invoke external functions from within CLIPS. User-defined CLIPS functions can be provided for most of the useful functions in a VR function library so that the programmer can use these functions on the right-hand side of interaction rules. Some useful VR functions include:

`make-invisible` Takes the `node-index` of an instance as an argument.

`change-color` Requires the `node-index` of an instance and three floats specifying the red, green, and blue characteristics of the desired color.

`test-collision` Takes two `node-index` integers and determines whether their corresponding objects are in contact in the simulation.

`play-sound` Takes an integer which corresponds to a soundfile (this correspondence is established in another index file of soundfiles).

The `play-sound` function takes an integer as an argument, instead of a string or symbol, because there is slightly more overhead in processing and looking up the structures associated with strings, and speed is crucial in a virtual reality application. Similarly, all user-defined functions should receive the integer value stored in an instance's `node-index` slot, instead of the instance-name, for speed purposes.

It is also possible to create a library of useful deffunctions which do many common calculations completely within CLIPS. For example, a distance function, which takes two instance-names and returns the distance between their corresponding objects, can be written as follows:

```
(deffunction distance (?n1 ?n2)
  (sqrt (+ (** (- (send ?n2 get-x) (send ?n1 get-x)) 2)
           (** (- (send ?n2 get-y) (send ?n1 get-y)) 2)
           (** (- (send ?n2 get-z) (send ?n1 get-z)) 2))))
```

3 Layers upon Layers

Once basic VR functionality is added to CLIPS, virtual environments can be organized in CLIPS according to familiar object-oriented and rule-based design principles. For example, a RABBIT class could be defined, and the example rules could be modified to pattern match on the is-a field instead of the name field. If this were done, any RABBIT instance would automatically derive the appropriate behavior. Once a library of classes is developed and an appropriate knowledge-base to go with it, creating a sophisticated virtual environment is merely a matter of instantiating these classes accordingly.

Consider this final example, illustrating points from Sections 2.5 and 3.

```
(defrule shy-rabbit-behavior
  ?rabbit <- (object (is-a RABBIT) (personality shy)
                     (x ?) (y ?) (z ?))
  ?hand <- (object (is-a HAND) (x ?) (y ?) (z ?))
  =>
  (if (< (distance ?rabbit ?hand) 100) then
    (bind ?rabbit-index (send ?rabbit get-node-index))
    (if (evenp (random)) then
      (change-color ?rabbit-index 1 0 0)   ; rabbit blushes
      else
      (make-invisible ?rabbit-index))))
```

This rule becomes relevant only if there is a shy rabbit and a hand in the simulation. If so, shy-rabbit-behavior is activated whenever the hand or the rabbit moves. If the hand gets close to the rabbit, there is a 50% chance that the rabbit will blush, and a 50% chance that the rabbit will completely vanish.

4 Conclusion

CLIPS has been successfully enabled with virtual reality programming capabilities, using the methodologies described in this paper. The two test users of this approach have found CLIPS to be a simpler, more natural paradigm for programming virtual reality interactions than the standard approach of managing and invoking VR functions directly in the C language. Hopefully, by using high-level languages like CLIPS in the future, more VR programmers will be freed from their current constraints of worrying about low-level details, and get on with what really matters — creating complex, intelligent, interactive environments.

References

[1] Ben Delaney. State of the art VR technology circa 1994. *New Media*, 4(8):44–45, August 1994.

[2] Ben Delaney. Virtual reality goes to work. *New Media*, 4(8):40–48, August 1994.

[3] Joseph Psotka. Synthetic environments for training. In *Second Annual Synthetic Environments Conference*, pages 79–107. Technical Marketing Society of America, March 1994.

[4] U.S. Army Training and Doctrine Command and U.S. Army Research Office. *Executive Summary — Virtual Reality/Synthetic Environments in Army Training*, October 1992.

[5] Andries van Dam. VR as a forcing function: Software implications of a new paradigm. In *IEEE 1993 Symposium on Research Frontiers in Virtual Reality*, pages 5–8, Los Alamitos, California, October 1993. IEEE Computer Society Technical Committee on Computer Graphics, IEEE Computer Society Press.