A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality

(this article first appeared in the May 1995 issue of IEEE Computer Graphics and Applications)

User Interface Group, Computer Science Department, University of Virginia

Alice Team: Randy Pausch (head), Tommy Burnette, A. C. Capehart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, Jeff White

ABSTRACT

We are developing Alice, a rapid prototyping system for virtual reality software. Alice programs are written in an object-oriented, interpreted language which allows programmers to immediately see the effects of changes. As an Alice program executes, the author can update the current state either by interactively evaluating program code fragments, or by manipulating GUI tools. Although the system is extremely flexible at runtime, we are able to maintain high interactive frame rates (typically, 20-50 fps) by transparently decoupling simulation and rendering. We have been using Alice internally at Virginia for over two years, and we are currently porting a "desktop" version of Alice to Windows 95. We will distribute desktop Alice freely to all universities via the World Wide Web; for more information, see http://www.cs.virginia.edu/~alice/

THE NEED FOR RAPID PROTOTYPING TOOLS

Virtual Reality (VR) has sparked many people's imaginations, but writing VR programs remains difficult. Besides the obvious problems in managing the arcane I/O devices (trackers, gloves, etc), the developer must allow the participant to *operate effectively* in the immersive environment. Virtual environments present a *new medium* for both the participant and the author. We believe that the best way to accelerate a new medium such as VR is to provide tools that allow people without highly technical backgrounds to create

Figure 1: An environment created with Alice

VR programs. These authors must be able to try different nuances of an idea without spending time waiting or re-implementing; the author must be able to easily ask “what if” questions.

To support this goal, we are developing Alice, a rapid prototyping environment which can generate environments such as the one shown in Figure 1. The name “Alice” honors Lewis Carroll’s heroine, who explored a rapidly changing, dynamic environment.

CHANGING A RUNNING ALICE ENVIRONMENT

The state of a running Alice program can be changed in three ways at any time:

1. The interactive, immersed, *participant* can manipulate objects within the VR environment, for example, by grasping them, gesturing, or speaking.
2. The programmer can interpret source code, either as a complete restart of the simulation, or as an incremental change to the current state.

3. The programmer can use GUI control panels.

   Many of our VR programs were developed using a two-person model, where the immersed *participant* uses the first mode of interaction and is assisted by a person at a desktop station using the second two modes. This is useful because the rapid prototyping cycle is so fast that constantly donning the equipment and coming back out to change the code is too time-consuming.

AUTHORING ALICE PROGRAMS

Alice programmers write in Python [PYTHON], a high-level, interpreted, object-oriented language. Python supports high level datatypes such as lists and hash tables as primitives in the language. Python also supports multiple inheritance, polymorphism, and an extensive collection of run-time libraries, including a large portion of the UNIX libraries usually access via the C language.

Because Alice is targeted towards novice programmers, it is important that Python, more than languages such as Tcl or Scheme, can be mastered by new Alice programmers with little effort. For example, the following code snippet sorts a list of words and prints those that start with a vowel:

```
wordList = ['rabbit', 'cake', 'queen', 'eat', 'drink']
wordList.sort()
for word in wordList:
 initialLetter = string.lower(word[0])
 if initialLetter in 'aeiou':
   print word
```

Alice programmers can also write portions of their simulations in C or C++, and statically or dynamically link those modules into their Alice application. More information on Python is available from ftp://ftp.cwi.nl or http://www.cwi.nl/~guido/Python.html.

Alice provides a set of Python classes for creating and manipulating geometric objects in three-dimensional environments. Alice models the world as a hierarchical collection of objects; Figure 2 shows the hierarchy of objects in the scene from Figure 1. By default, an objects moves in its parent's coordinate system: a billiards ball, for example, specifies its

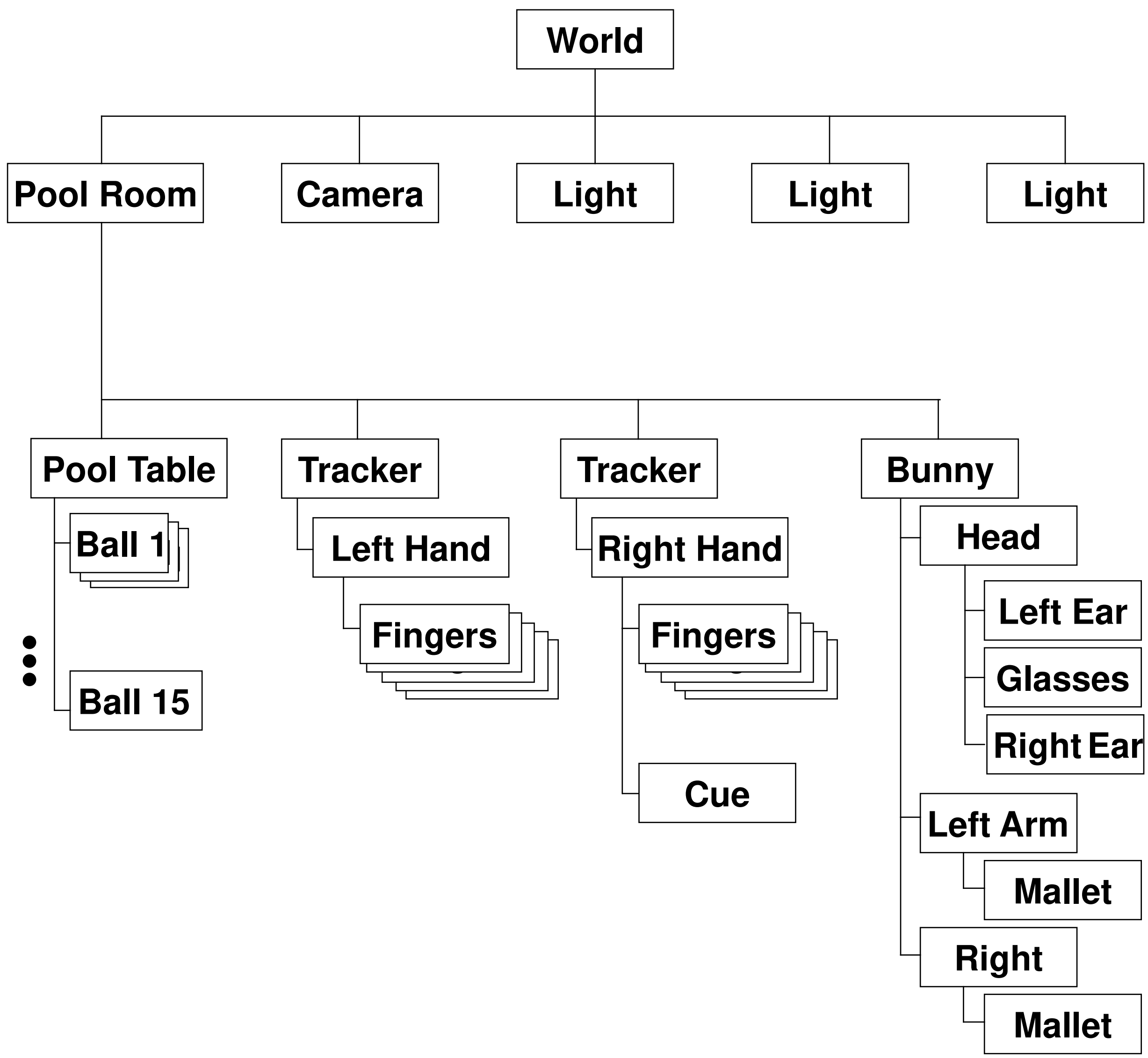

Figure 2: Hierarchy for Figure 1 environment. There are actually more than 50 objects; some have been elided to save space.

position relative to the billiards table. If the billiards table is moved within the room, all the balls will move with it. The parent/child relationships in the hierarchy can be reorganized on-the-fly. To have the cue ball hop off the table and onto the floor, one could type

```
cueBall.makeChildOf(floor)
```

Some popular systems [INVENTOR] provide hierarchically organized rendering lists, which are fundamentally different from the Alice-style hierarchy. In a hierarchical rendering list, the system performs an inorder traversal of the data structure, and performs operations at each node. However, the order of the traversal is important. For example, if the system traverses a node containing a chair's geometry before the system traverses a node containing a light, the chair will not appear lit by the light, even though other objects in the environment will. While hierarchical rendering lists are very flexible, they make it difficult to implement and maintain intuitively simple structures. In contrast, Alice programmers can place headlights in a virtual automobile hierarchy, and then drive that car around the environment, without worrying about internal ordering in the tree. In a hierarchical rendering list where order matters, the programmer would have to ensure that the headlight node be traversed before all the other nodes in the scene graph, which is awkward at best.

In a similar fashion, Alice allows programmers to have multiple cameras in the hierarchy, each of which render to separate windows on the screen. For example, to interactively add a separate "cue ball's eye view" window in the billiards simulation, an Alice programmer can type:

```
cueBallCamera = Camera()
cueBallCamera.makeChildOf(cueBall)
```

By default, transformations on Alice objects are performed in the coordinate system defined by that object's parent. However, by adding an extra parameter to any operation, the programmer can have that operation performed in *any* object's frame of reference, regardless of where that reference object lives in the hierarchy. We have found this mechanism extraordinarily useful in a variety of surprising ways. For example, when creating a new object in the environment, it is often difficult to place it in a location where the immersed participant can see it. However, with the ability to move in any object's coordinate system, we can place the object one meter *in front of* the participant's head. Note that this does not make the object a child of the participant's head, it merely places the object in a location relative to the head's current position when the move is performed.

Each Alice object has a list of action routines, or *callbacks*, which are executed each frame of the simulation. These action routines are responsible for changing the internal state of the object (position, orientation, application-specific state, etc.) for the next animation frame. For example, in Figure 1, the action routines for each ball could compute the physics of their collisions, the bunny might beat his drum, and the cue stick might sample input devices worn by the immersed participant.

In addition to the “once per frame” action callbacks, Alice objects can also employ a time-based animation facility. Any of the standard operations on Alice objects (e.g. move, rotate, setColor) can be made to animate over a given time period, merely by passing in an extra parameter to the method call. So, instead of teleporting from the old location to the new, the object smoothly animates over time. By default, these animations employ slow-in/slow-out (acceleration/deceleration), but the programmer can choose other interpolation functions or supply a custom function of his own design.

DECOUPLING SIMULATION AND RENDERING

Many existing VR systems use a main loop such as:

```
while (true) do
 BEGIN
 get information from input devices, such as head trackers
 compute a "tick" of the simulation
 update graphics to the participant
 END
```

The problem with this model is that it couples the graphics rendering rate (based on the movement of the participant’s head) to the underlying simulation’s computation speed. If an immersed participant is moving his or her head to change the viewpoint, he or she must wait for the simulation work to complete for each frame. In the billiards example, if the system could only calculate the physics for the billiard ball collisions at 2 frames/second, then the display to the participant would update at 2 frames/second. Since the user wants visual updates immediately, this situation is unacceptable.

In Alice, we separate the simulation frames from the rendering frames by using a multiple process architecture, typically with each process running on a separate CPU. This allows the participant to interact in real-time, even when the application-level simulation computations become complex. This is useful because even though the changes to the environment (e.g. the position of the billiard balls) might only update twice a second, the *participant’s viewpoint* in the head-mounted display, being fed by the tracking devices, will update more rapidly, and that is the frame rate that must be kept high at all costs, in order to maintain the illusion of presence in an environment.

To accomplish this separation, we have two main processes: the first computes the simulation and maintains its state, and the second maintains a geometric database and renders it from the participant’s current point of view. Together, this allows Alice to support the billiards example where the balls update their positions at, say, 5 simulation frames/second, while the participant can walk around the table and have his viewpoint updated at, say, 30 rendering frames/second. These processes typically run on separate machines connected by a local area network.

Several other VR systems, including the MR Toolkit [MR], also use multiple processes, but they require the programmer to control both: one process is typically viewed as the “interaction process” and the other is the “computation process,” and the programmer

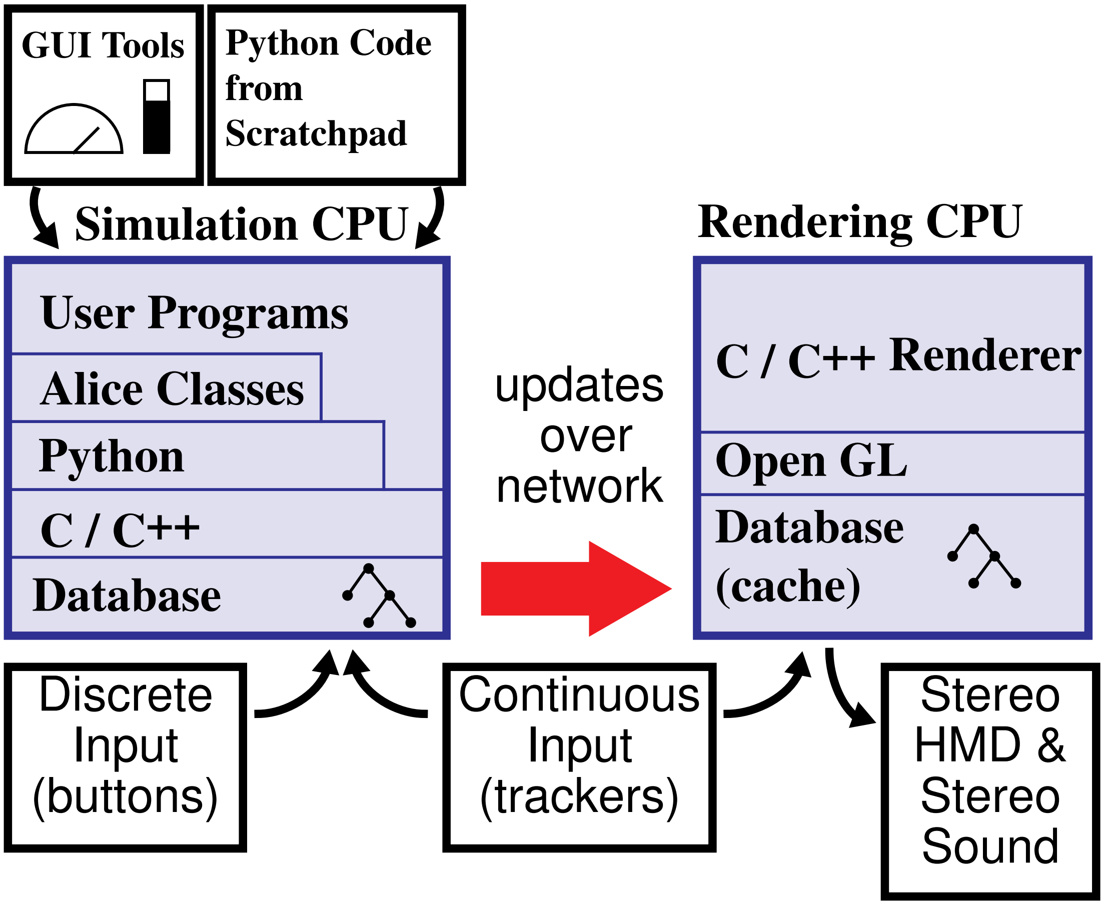

Figure 3: Alice architecture. The programmer need not be aware of the existence of the rendering process.

must explicitly handle all data communication between the two. In Alice, we make the process separation *transparent* to the programmer, who writes a single-process simulation. Figure 3 shows a block diagram of the system.

When the simulation process begins, it transparently spawns a rendering process on a remote machine. As the simulation process runs, it updates its local database, and queues update commands for the rendering processes, which are sent over the network as an atomic unit at the end of each simulation frame. A device handling process is also spawned, which streams updates of multiple six-degree-of-freedom trackers to both the simulation and rendering processes. The rendering process needs the information in order to draw the environment, and the simulation process needs it because some objects in the hierarchical database are "attached" to trackers for their current position and orientation, and the program may query that information. During each rendering frame, the rendering process walks the tree and renders graphics via the openGL graphics library, and localized sound via Crystal River Engineering's Beachtron card. Because the rendering process always has a consistent set of simulation data, it can render multiple viewpoint (and hand position) updates while the simulation computes the next computational change to the database. Discrete input devices, such as buttons, send events over the network to the simulation process.

The rendering process can read database update commands from the network and update its cache of the database in a small fraction of the time that it takes to render the graphics, so the networked database updates do not substantially affect system performance. The key observation about the multiple process architecture is that this is all transparent to the programmer, who merely writes a program in Python.

The hierarchical database and frame rate separation interact quite nicely. For example, consider what happens if the participant reaches up and grabs an animated toy, such as the mechanical bunny. First, the simulation detects the grab event via gesture recognition or a button press. The simulation responds by re-parenting the bunny from the room to the participant's hand. This command would be sent over, and updated in the renderer's cache. Although the animation of the bunny's beating his drum, as a simulation operation, might still only be updated 5 times per second, the bunny (as a child of the hand, driven by the tracker), would follow with the participant's hand at 30 frames/second. In our experience, participants are not bothered by this dichotomy. The system must respond to the *participant's* motions as rapidly as possible, but the other animations in the environment can more safely degrade, even when the participant is grasping an animated object!

This approach works less well if the animations themselves are meant to be low-latency feedback techniques. For example, in applications which attempt to grid the user's hand to fixed distances, participants are more sensitive to lower simulation frame rates. We have not yet found these cases to be motivating enough to implement a three-level separation into simulation/feedback/rendering.

CURRENT STATUS

We have been using Alice at the University of Virginia for two years. We have a rapidly growing library of geometric objects and Python classes which embody object behavior, and we have used Alice twice in our graduate level computer graphics course. On several occasions, researchers from other organizations have used Alice productively during two-day visits; we see this as the best end-validation of our system's simplicity and utility.

We are currently porting Alice to Microsoft Windows 95, where Alice will be a desktop system for interactive 3D graphic using a mouse and keyboard. We will distribute this version of Alice freely to all universities via the Internet.

REFERENCES

[PYTHON] Guido van Rossum, Interactively Testing Remote Servers Using the Python Programming Language. This paper, and more information on Python is available from ftp://ftp.cwi.nl or http://www.cwi.nl/~guido/Python.html.

[INVENTOR] Paul Strauss and Rikk Carey, An Object-Oriented 3D Graphics Toolkit, Computer Graphics (SIGGRAPH '92 Proceedings), 26:2, July 1992, pages 341-350. More information is available via the World Wide Web at http://www.sgi.com/Technology/Inventor.html.

[MR] Chris Shaw, Jiandog Liang, Mark Green, and Yungi Sun, The Decoupled Simulation Model for Virtual Reality Systems, Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference, May, 1992, Monterey, California, pages 321-328. More information is available via the World Wide Web at http://www.cs.ualberta.ca/~graphics/MRToolkit.html.

ACKNOWLEDGMENTS

Our thanks to the many members of the UVa User Interface Group who have worked on Alice, and to Charles Choi, who built custom input devices for us. We are also grateful to the UVa Computer Science Department, which has been very supportive of our group. This and other work by the UVa User Interface Group is supported in part by ARPA, NSF, NASA, SAIC, and the Commonwealth of Virginia.