

Integrating Pedagogical Agents into Virtual Environments

W. Lewis Johnson
Jeff Rickel
Randy Stiles
Allen Munro¹

Abstract

In order for a virtual environment to be effective as a training tool, it is not enough to concentrate on the fidelity of the renderings and the accuracy of the simulated behaviors. The environment should help trainees develop an understanding of the task being trained, and should provide guidance and assistance as needed. This paper describes a system for developing virtual environments in which pedagogical capabilities are incorporated into autonomous agents that interact with trainees. These pedagogical agents can monitor trainees' progress and provide guidance and assistance. The agents interact with simulations of objects in the environment, and with trainees. The paper describes the architectural features of the environment and of the agents that permit the agents to meet instructional objectives within the virtual environment. It also discusses how agent-based instruction is combined with other methods of delivering instruction.

1. Introduction

Training is an important application area for virtual reality technologies. Immersive displays, three-dimensional sound, and other types of virtual reality interfaces can be used to create realistic virtual environments that closely simulate the real-world environments for which students are being trained. Students can therefore "learn by doing" (Dewey 1939), i.e., improve their skills through practice on realistic tasks. It is expected that the richer perceptual cues and multimodal feedback (e.g., visual, auditory, and haptic) afforded to students in virtual environments will enable VR-based training to transfer easily to real-world skill (Durlach & Mavor 1995). Preliminary evidence supports this conjecture (Regian et al 1992).

In order for a virtual environment to be effective as a training tool, it is not sufficient to focus just on maximizing the fidelity of the object renderings and behaviors. One must also consider how the environment supports the delivery of instruction, and facilitates effective learning experiences. There is considerable experience to date on the problems that arise from unguided interactions with simulation-based learning environments; since interactive virtual environments typically incorporate simulations such experience is relevant to the design of virtual environments for training. When students do not know

¹ Drs. Johnson and Rickel are at the Information Sciences Institute, University of Southern California. Mr. Stiles is at the Advanced Technology Center, Lockheed Martin Missiles and Space. Dr. Munro is at Behavioral Technology Laboratories, University of Southern California. Email: johnson@isi.edu, rickel@isi.edu, stiles@aic.lockheed.com, munro@usc.edu.

how to complete a given task, the learning process can get bogged down while the students are searching for some method of completing the task (Hill & Johnson 1995). If they do not have a clear understanding of the tasks being trained, they may perform the tasks incorrectly without realizing it, and thus learn the task incorrectly. Furthermore, they may acquire incorrect models of the systems with which they are interacting (Self 1995).

There are a number of ways to alleviate the deficiencies of simulation as a training tool. One way to ensure that learning experiences are effective is to have human instructors continually monitor student activities and give feedback. However, this places heavy demands upon instructors' time, and may limit the amount of access that students have to the learning environment. Another approach is to support collaborative learning, where groups of students work together within the virtual environment, as in a "multi-user dungeon" (Soloway 1995). Collaboration is definitely appropriate and even necessary for some learning situations, such as team training. However, collaboration by itself is only a partial solution. A group of students who are equally unfamiliar with the tasks being trained may duplicate and reinforce each other's mistakes. Students who rely too much on the assistance of other students may excessively interrupt and divert their attention, and slow the progress of the group. Requiring multiple participants can force limitations in the amount of practice students get in the virtual environment, either because not all individuals are present and available, or because a limited amount of virtual reality hardware is available.

This paper describes an alternative approach that uses synthetic, autonomous agents to support learning in virtual environments. Students are immersed in a virtual environment in which they can practice tasks that they are learning to perform. An agent called Steve (Soar Training Expert for Virtual Environments) shares the virtual space with the students and can demonstrate tasks, offer advice, and answer questions. Multiple Steve agents and students can inhabit the same space. When fully developed the Steve architecture will support team training, enabling students and agents to work together on tasks.

This work is part of the Virtual Environments for Training (VET) project, which consists of three technical thrusts. The virtual environment interface system, called Vista (Stiles et al 1995), was developed by Lockheed Martin to create and interact with immersive virtual environments. The VRIDES system, developed by USC Behavioral Technologies Laboratory, simulates the behavior of the objects in the virtual environment, and provides a framework for creating structured lessons. VRIDES is a 3D extension of the widely used RIDES authoring system for 2D simulations. The Steve agent itself was developed by USC / Information Sciences Institute. It is built on top of the Soar agent architecture (Laird et al 1987), augmented with a set of general-purpose instructional capabilities, such as explanation and demonstration.

2. Objectives of the Approach

Steve's architecture is intended to provide both control for autonomous agents and intelligent tutoring capabilities. These technologies have traditionally been developed

independently. Steve is unusual in combining both functions, in order to create agents that can both act and teach.

There is already a significant body of work applying artificial intelligence techniques to human learning, by means of intelligent tutoring systems (Wenger 1987). AI techniques have been shown to be effective in such systems both to improve student performance and to reduce learning time. They rely upon a model of the subject matter being taught in order to track the student's progress and provide assistance where needed. Such systems frequently seek to assume the role of a tutor in an apprenticeship learning situation (Collins 1989). In the apprenticeship learning model, the tutor interacts with the student in different ways during the learning process: first modeling the skill (i.e., demonstrating and explaining how it is done), then coaching the student in performing the task, and gradually reducing the amount of assistance as the student becomes increasingly proficient. In a similar vein computer tutors adapt to the needs of individual students when presenting material, coaching, critiquing, etc. Intelligent tutoring technology has not been applied extensively in virtual environments, although a few examples exist, such as Loftin and Kenney's Hubble Telescope maintenance trainer (Loftin & Kenney 1995), Kotani and Maes's guide agent in the ALIVE system (Kotani and Maes 1994), and Billinghamurst and Savage's system for training sinus surgery (Billinghurst & Savage 1996).

A distinct thread of research has focused on the development of autonomous agent technologies for synthetic environments. Although in many instances autonomous agents are controlled using simple mechanisms such as state transition networks or potential fields, there has been increasing interest in the use of AI architectures for controlling agents in synthetic worlds. AI techniques are appropriate when the agents must perform complex behaviors in a wide variety of situations. AI planners have been employed to control the behavior of human figures such as Jack (Webber & Badler 1993). Agent architectures such as Soar and BBK have been used to create agents that can pursue goals yet respond dynamically to changing environments, as is necessary for agents that are expected to operate effectively in interactive virtual environments (Tambe, Johnson, Jones, et al. 1995, Hayes-Roth & van Gent, 1997).

In virtual environments, there are a number of potential advantages to integrating intelligent tutoring and autonomous agent capabilities into the same system (Durlach & Mavor 1995). Such a combined system can not only offer students advice, but actually show the students how the task is done: the agent can move its body in the virtual world and manipulate the same virtual objects that the students are expected to manipulate. Thus acting in the virtual world can be in service of teaching in the virtual world. Because the student and the agent are in the same environment, they can interact with each other in a more natural way, akin to the way human tutors and students interact: through gaze, gestures, and the like. Studies have shown that users sometimes ascribe personality traits to computer systems, and user acceptance is determined in part by the compatibility of the user's personality traits and the system's perceived personality traits (Fogg & Moon 1994). The agent-centered approach affords the possibility of experimenting with the use of different types of interpersonal interactions in the service of pedagogical strategies.

In order to successfully integrate intelligent tutoring technology and autonomous agent technology in the context of virtual environments, a number of difficult technical issues must be addressed. The architecture must permit the agent to use its knowledge to explain tasks and monitor students performing them, as well as allowing the agent to perform the task itself. In contrast, most agent architectures focus just on task execution, and are unable to explain to users what they are doing, or recognize when others are attempting to perform the same task. The architecture should be able to act both reactively and in a goal-directed manner, so that its actions are relevant both to the task being trained and the current state of the environment, and so that the agent is able to explain the relevance to the students. At the same time the system must meet all of the requirements of embodied synthetic agents: the agent must be able to perceive the virtual world, as well as control its realization (i.e., its body) in the virtual world (e.g., gaze at objects, point at and manipulate objects, etc.).

In addition to the above characteristics, we require further properties of the Steve agent that are useful for immersive training systems. First, the architecture must be able to control multiple alternative realizations or bodies. Agents may appear either as full human figures, or as abstracted figures consisting just of hands and faces, or possibly in other forms.² There are two reasons for this. When viewing a virtual environment through a head-mounted display, the student's field of view is quite limited. When operating close enough to an object to be able to reach it, a full human figure would either be mostly invisible, because it is outside of the agent's field of view, or might block the student's view of the object. Also, it is possible that if naturalistic human figures are chosen, students might critique the fidelity of the human figure model instead of focusing on the instruction and guidance provided by the agent. Steve therefore has been designed so that it can support multiple agent realizations, and so that the realizations can be readily interchanged.

Another important requirement is that it should be easy to provide the knowledge Steve needs in order to perform in a new domain. Other projects implementing agents in Soar, such as Soar-IFOR (Tambe, Johnson, Jones, et al. 1995), specify agent behavior by means of production rules. Encoding task behavior as production rules requires a detailed understanding of the Soar execution engine, as well as expertise in artificial intelligence. Other agent architectures such as BBK require specialized expertise as well. In contrast, Steve has been designed so that instructional designers and others without such expertise can teach Steve new tasks. Two techniques have been investigated to this end: high-level languages for specifying tasks, and techniques for acquiring task knowledge from demonstration examples. These automatically generate the Soar production rules needed to encode the acquired task knowledge.

Finally, Steve is intended to be used in concert with non-agent-based instructional methods. The RIDES project has developed an authoring interface that simplifies the construction of structured exercises of various types, such as drills that test the student's ability to name and identify objects (Munro et al 1993). VRIDES, the descendant of RIDES designed for virtual environment training, inherits this interface. Although such

² Another example of how an agent might be realized is the "angel" in the virtual environment depicted in the film *Disclosure*. There the agent appears as a diminutive human figure that hovers over the user's shoulder.

structured exercises could in principle be administered by an agent such as Steve, there is relatively little advantage in doing so. Instead, Steve was designed so that it can be combined with instruction delivered by other means, in particular VRIDES. A given lesson plan can make use of Steve for some lessons, VRIDES for other lessons, a combination of the two, or even other instructional tools that might be added in the future.

The remainder of this article describes in more detail how the above issues are addressed in Steve. The next section provides examples of Steve's capabilities, and discusses how they meet the above requirements. Then an overview of the VET system architecture is presented. Section 5 presents the Steve architecture, describing Steve's main components and how they interact. Section 6 describes how knowledge about tasks is represented in Steve. Section 7 shows how task knowledge is used in Steve to support student learning. Section 8 describes a method that enables Steve to acquire knowledge automatically from demonstration examples. Section 9 explains how Steve interacts with the instruction management and authoring support capabilities of VRIDES.

3. Examples

Our initial application for Steve is teaching students how to operate a High Pressure Air Compressor (HPAC). The HPAC is part of the gas turbine propulsion systems aboard US Navy ships. Using a head-mounted display, students enter the room containing the HPAC. The 3D model of the HPAC and its surroundings was created by Lockheed Martin, based upon CAD models acquired from the US Navy. Lockheed Martin also created the Vista Viewer software by which students view and interact with this virtual world. The Vista Viewer can be used in immersed mode, using a head-mounted display, or in desktop mode, displaying the scene monoscopically. The image in Figure 1 was created by the Vista Viewer in desktop mode.

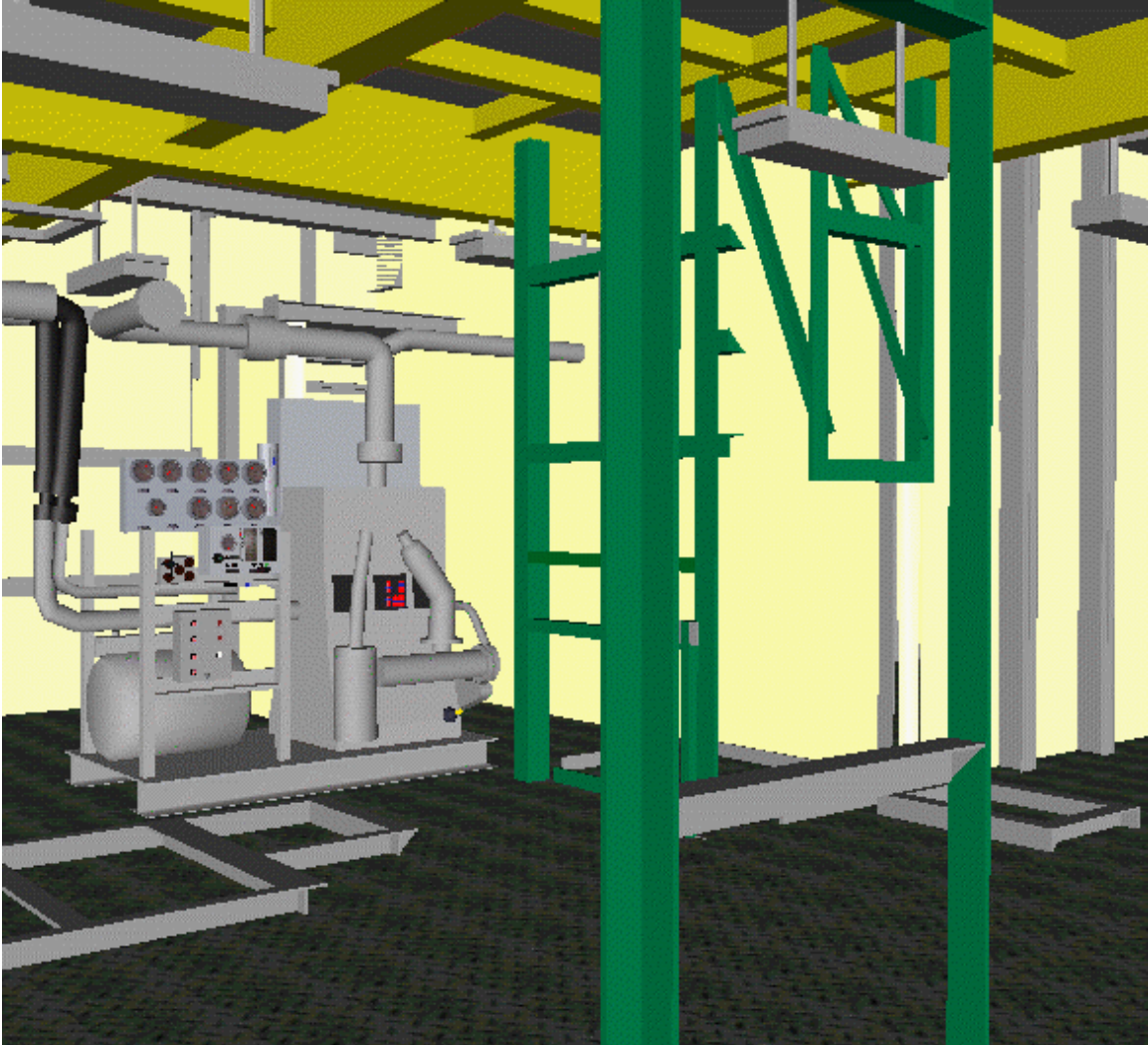


Figure 1: The HPAC shipboard environment

Steve can appear in the virtual world in several different ways. Typically, Steve appears as a head and a hand. Figures 2, 3, and 4 show snapshots of this version of Steve demonstrating a routine inspection of the HPAC. The image in Figure 2, for example, shows Steve pointing to one of the lights on the HPAC console, and commenting about it using synthesized speech. Steve continually adjusts his gaze to follow his focus of attention; in this case he is directing attention to the student receiving the demonstration. In Figures 3 and 4 Steve is looking toward a console button and the oil level dipstick, respectively, as he is manipulating them. Gaze provides a subtle way of attracting and maintaining the students' attention; it also helps give Steve the impression of being responsive to his environment, and hence more lifelike.

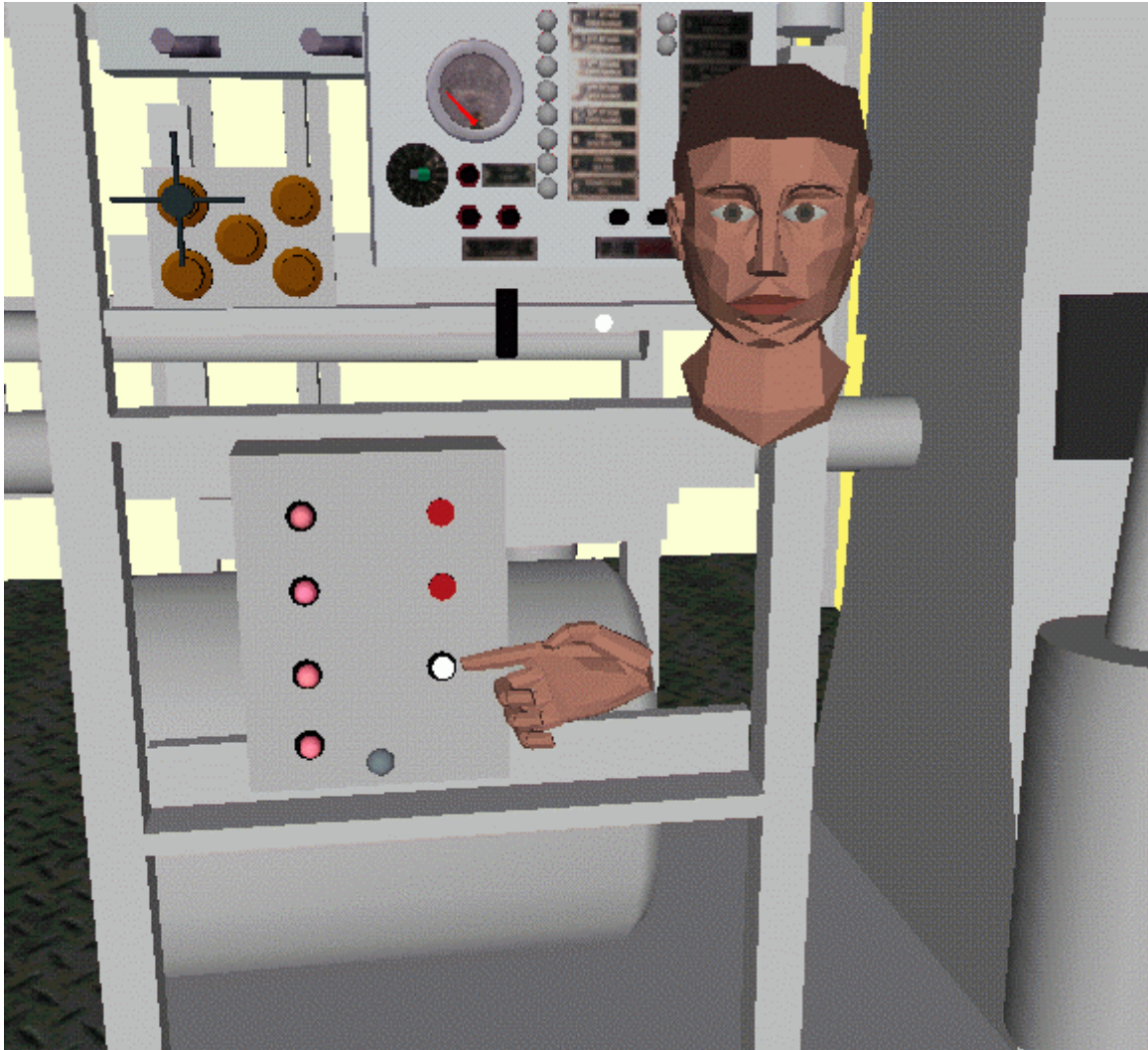


Figure 2: Steve pointing at an indicator light

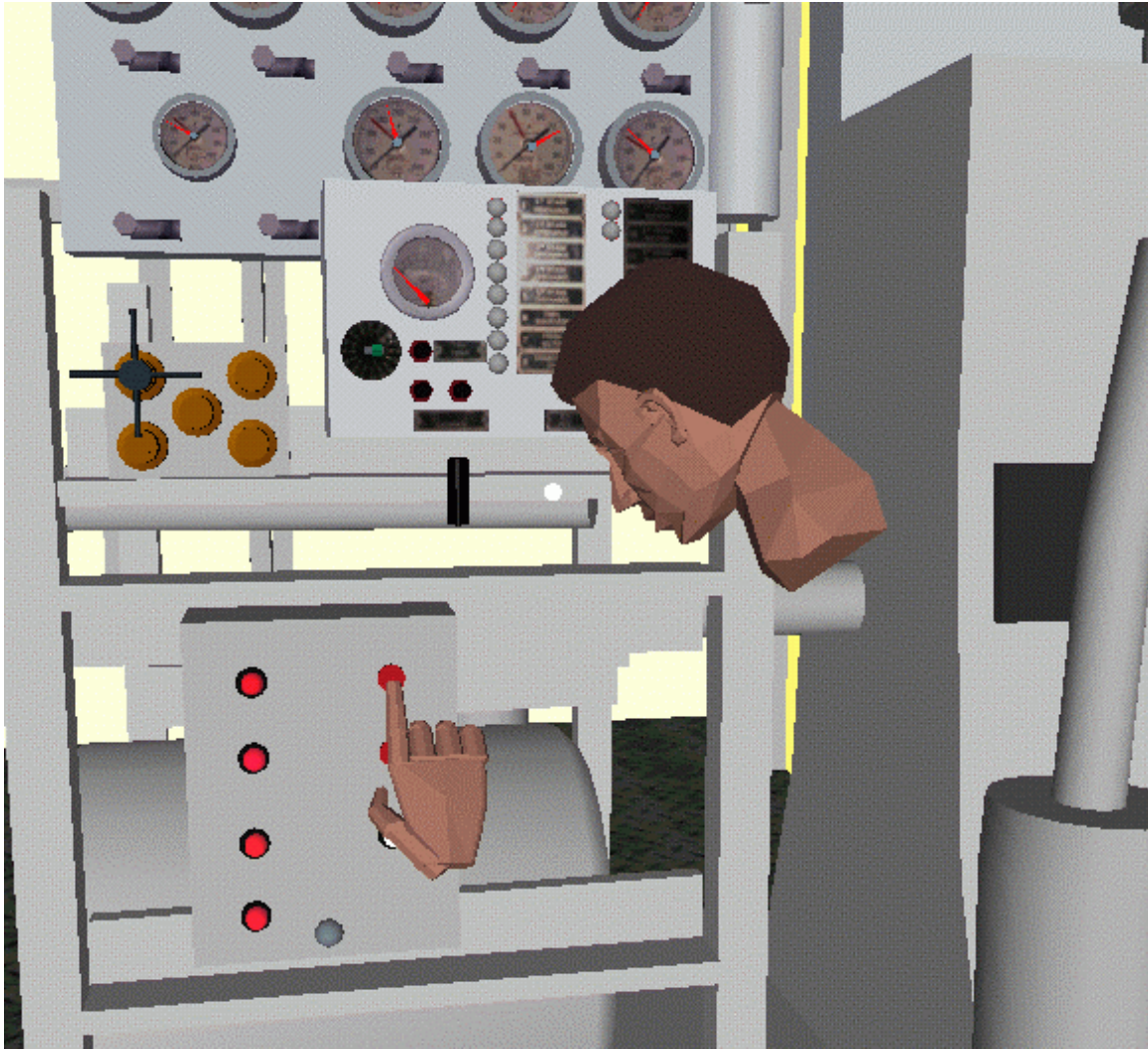


Figure 3: Steve pressing a button

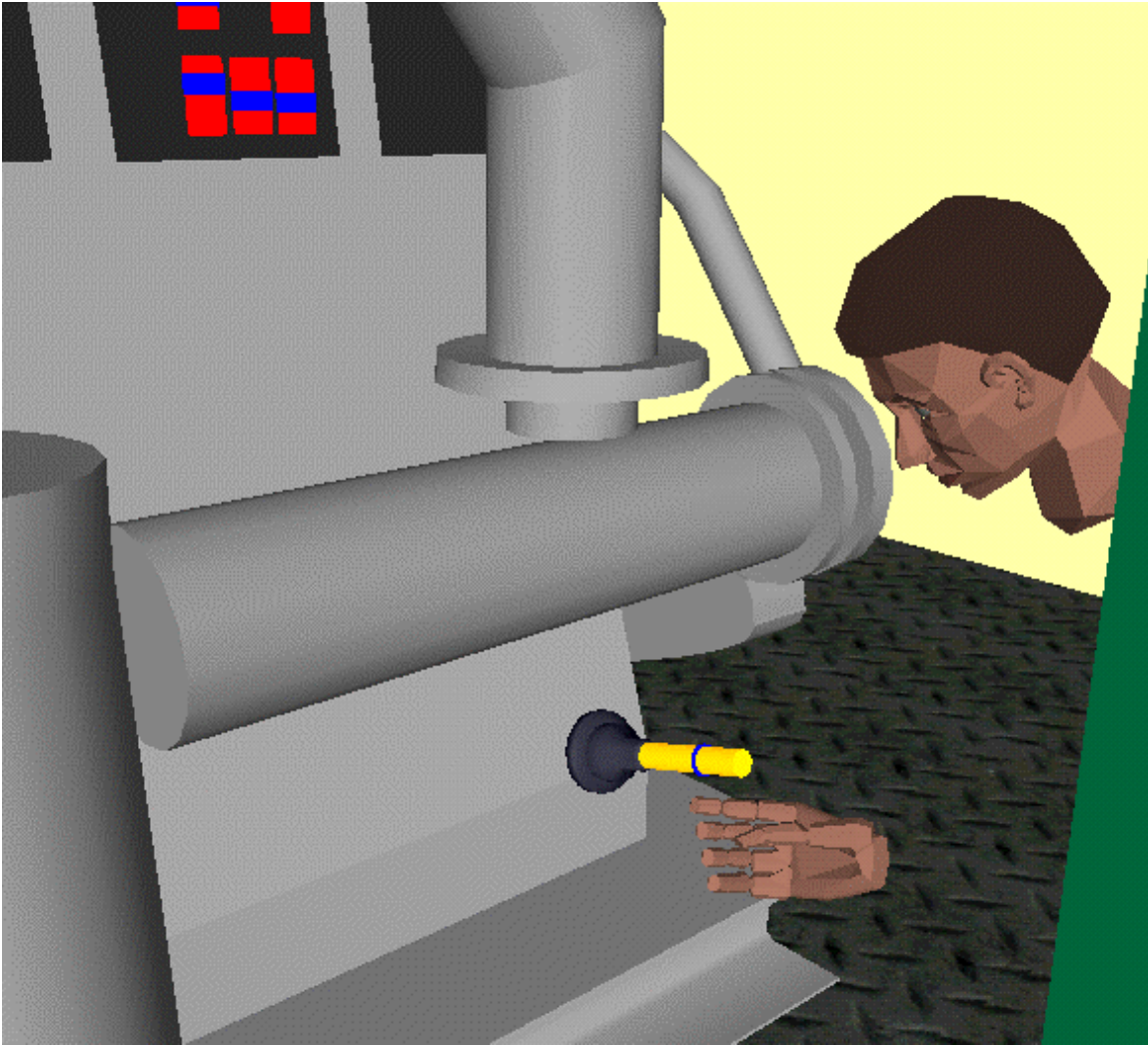


Figure 4: Steve grasping a dipstick

We are also experimenting with two alternative representations for Steve: as a full human figure (using the Jack software developed at the University of Pennsylvania [Badler et al 1993]), and as a hand alone. The image in Figure 5 shows one Steve agent, represented by a human figure, watching another Steve agent, represented by a hand, demonstrate an inspection of the HPAC. This is not a typical use of Steve – Steve agents usually monitor students, not other agents. However, the picture does illustrate that Steve agents can interact with other participants whether human or synthetic. This capability will prove crucial as Steve is extended to support the training of team tasks, where some team members may be human and others synthetic.

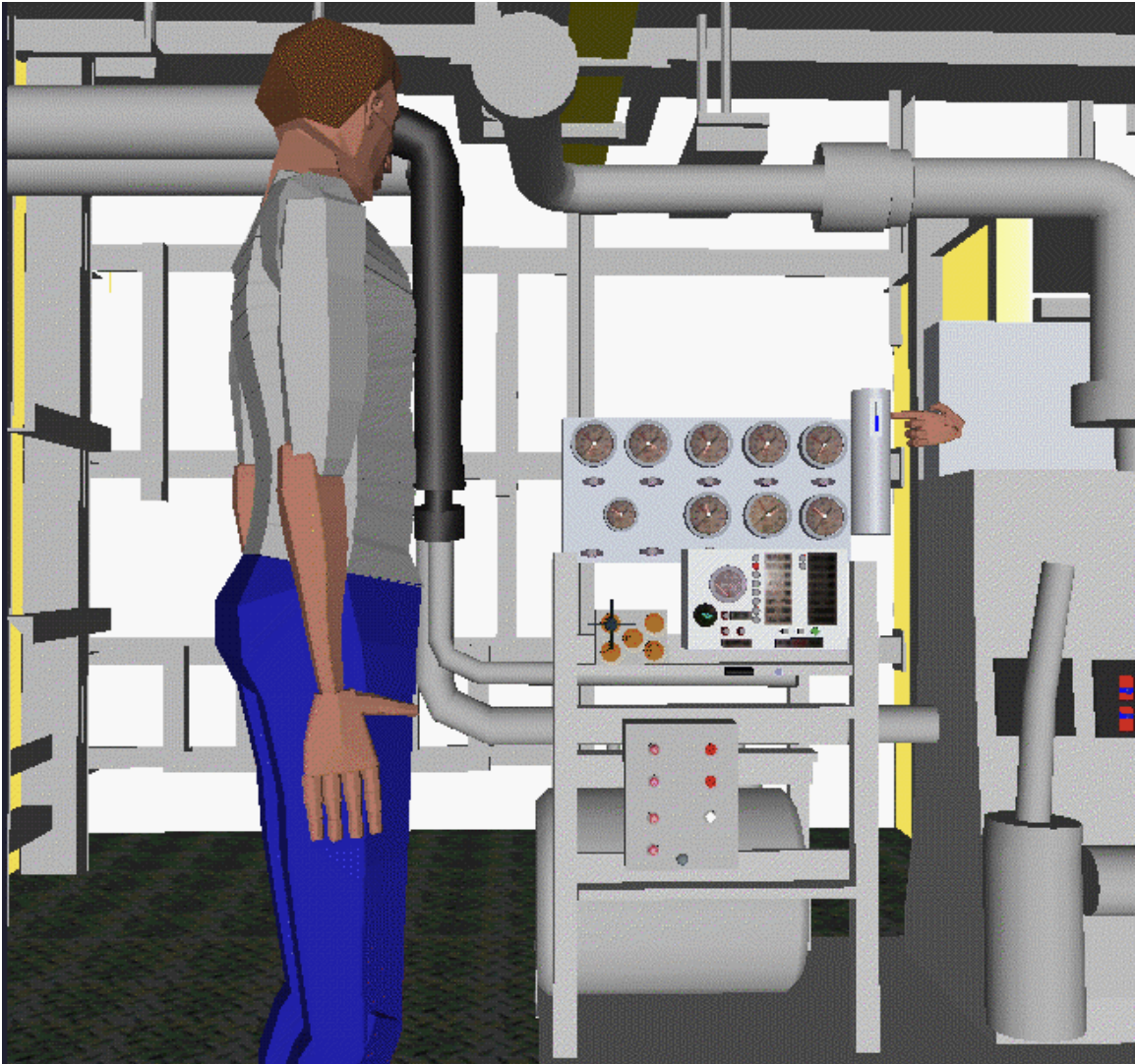


Figure 5: One Steve agent watching another

In addition to demonstrating tasks, Steve can monitor students performing tasks, providing help when it is requested. Figure 6 shows the icon palette used to communicate questions to Steve. At any time while performing the task, the student can touch the “What Next?” button, requesting help regarding what action to perform next. Steve responds with suggestions such as “I suggest that you push the test button,” realized as spoken utterances using the TrueTalk speech generator. If the student is unclear about the reason for Steve’s suggestion, they can select the “Why?” button, whereupon Steve responds with an explanation of rationale, such as “That action is relevant because it places the condensate drain monitor in test mode.” If the student asks “Why?” again, Steve will generate follow-on explanations, e.g., why it is necessary to place the condensate drain monitor in test mode. Steve’s task model represents the interrelationships between steps in the task, such as the other steps that a step enables and the goals it achieves. This permits Steve to generate a series of explanations of both

actions and goals. If the student selects the “Show Me!” button, Steve will carry out the action, explaining what it is doing at the same time.

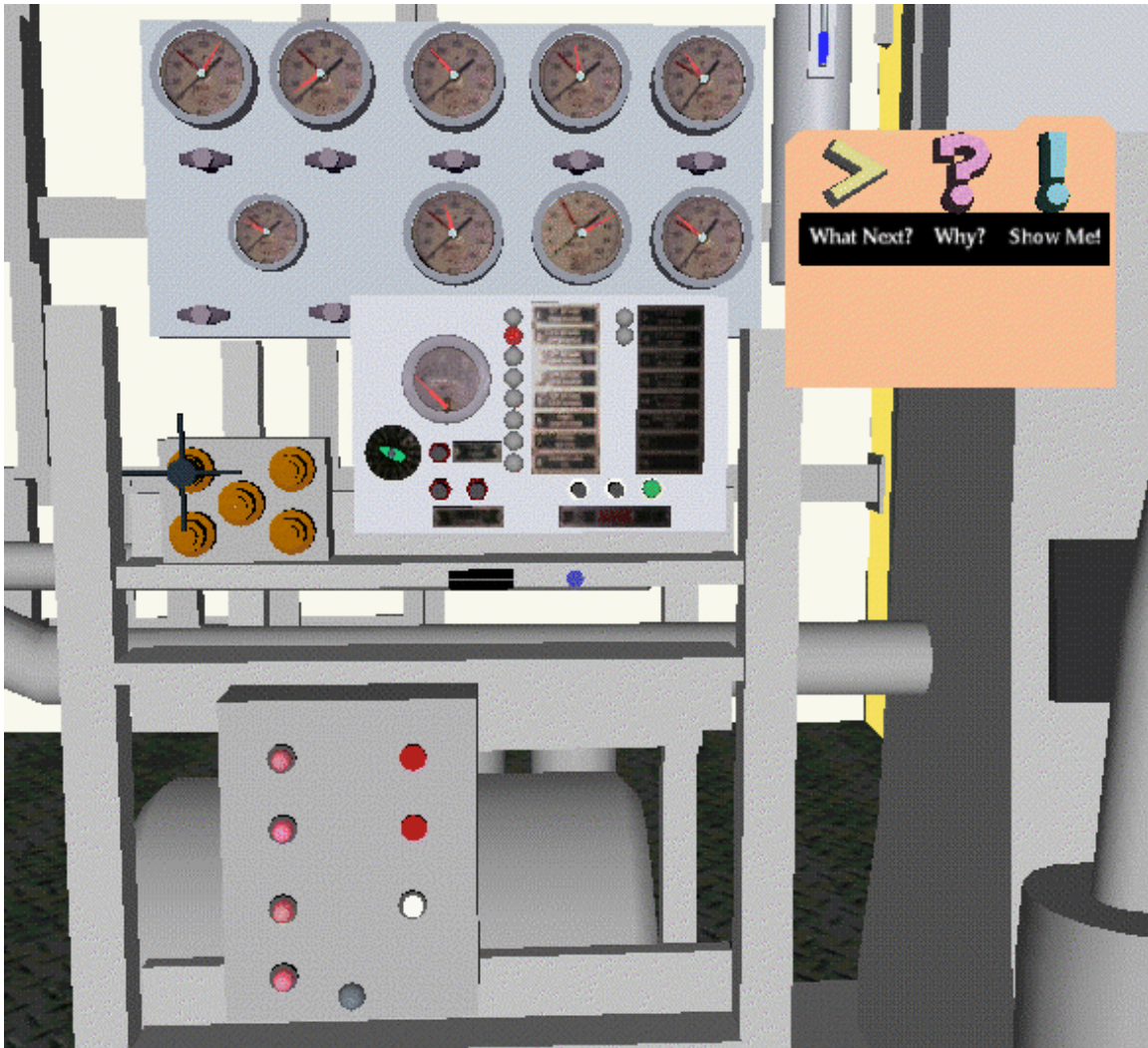


Figure 6: Interaction interface for Steve

Steve maintains an episodic memory of the actions that it performs during a demonstration, so it can explain the rationales for its previous actions as well as its current actions. This is useful when Steve is demonstrating a task for the first time, as well as in situations where it is not possible to interrupt the agent to ask questions without disrupting the activity. Time-constrained tasks and team tasks fall into this category.

Instructors can also interact with the environment in order to teach Steve new tasks. To train Steve, the instructor first places the objects in the environment in some neutral initial state, and then carries out a sequence of actions in the virtual world. Steve observes the instructor’s actions, and whenever the instructor performs an action that is new to Steve (e.g., pressing an unfamiliar button on the console), it asks the instructor to enter a name that Steve can use to refer to the action (e.g., turn-handle), along with text describing the purpose of the action. At the same time, Steve keeps track of what state

changes have occurred in the environment as a result of the action. When the instructor indicates that the task is complete, Steve puts up a menu on the instructor's console listing the differences between the initial state and the final state; the instructor marks those changes that were the intended effects of the task, as opposed to incidental or side effects. Steve then proceeds on his own to determine how the task achieves the intended effects, by performing variants of the demonstration example and checking whether or not the desired effects are achieved. It then asks the instructor to verify which of the remaining dependencies were intended. Once the task description is verified, Steve can use it to train students to perform the same task. Because Steve understands the task dependencies, it can explain them to students. The current implementation is able to learn a subset of the constructs in Steve's task representation, and is being extended so that hopefully the full range of tasks can be learned via this method.

4. The VET System Architecture

Steve agents interact with the other components of the VET virtual environment system in order to produce the capabilities that are illustrated above. The VRIDES component runs the simulation that controls the virtual world. Steve monitors the state of the virtual world through messages it receives from VRIDES. The Vista Viewer components (one for each participant) provide an interface between the virtual world and the human participants; they produce a 3D graphical rendering of the virtual world, and they detect interactions between participants and virtual objects. Steve monitors the actions and field of view of the human participants through messages it receives from Vista Viewers, and Steve controls its own visual appearance by sending messages to the Vista Viewers. Steve generates speech by sending messages to Trishtalk components. Each participant has their own Trishtalk, and each Trishtalk is an extended version of Entropic's TrueTalk text-to-speech program. All of the message passing among these components is accomplished via a communications bus using a message protocol called TScript. The overall system architecture, illustrated in Figure 7, is similar in some ways to other virtual environment architectures such as SIMNET (Calvin et al 1993) and Spline (Barrus et al 1996), in that it provides a common interface so that multiple applications can share access to a virtual environment over a network. One major difference is that multiple threads of interaction can take place at the same time, each using a different set of messages. For example, one set of messages is used to control the Trishtalk speech synthesis system, another set of messages is used to communicate changes to object locations in the virtual world, and yet another set of messages is used to communicate changes in the simulation state.

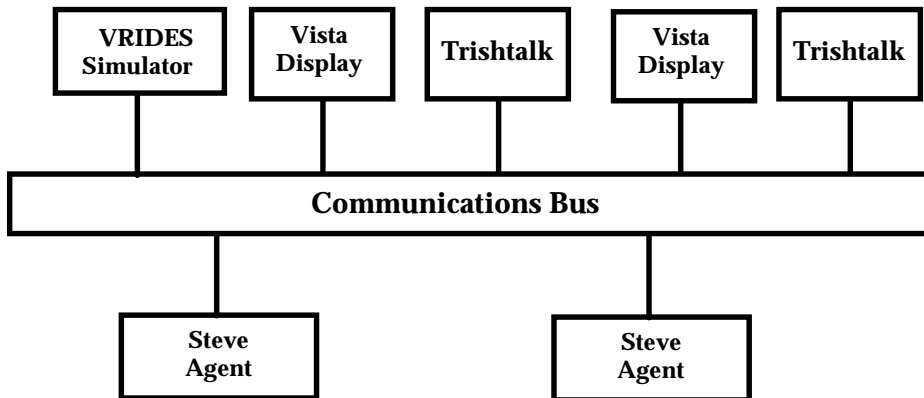


Figure 7: VET Architecture

The communications bus allows an arbitrary collection of components to communicate. It is currently implemented on top of Sun's ToolTalk software. Components connect to the communications bus by sending a message, and they subsequently send messages announcing the types of messages they wish to receive. Components do not send messages directly to other components. Rather, all messages are sent to the communications bus, and the communications bus broadcasts each message only to those components that registered interest in it. This provides an important filtering mechanism; components only receive message traffic they can use. The approach also provides a more extensible architecture than direct component-to-component communication.

The VRIDES component, which controls the behavior of objects in the virtual world, is a 3D extension of the RIDES 2D simulation authoring system (Munro et al 1993). In VRIDES, as in RIDES, each object in the virtual world is assigned a set of attributes. Some attributes control the visual appearance of the object, while others control its behavior. The behavior of the objects is programmed by rules and constraints that propagate changes in one object to other objects. This object-oriented representation makes it easy for other programs, such as Steve, to monitor the state of the simulation.

A Vista Viewer component provides the interface between a human participant and the virtual world. Vista displays the objects in the virtual world in real time, and students can view the display either immersed, using position sensors and a head-mounted display, or in a flat-screen window. Multiple students, each with their own Vista Viewer, can connect to the same communications bus, and each will experience the same changes in the environment (albeit from their own viewpoint).

Vista acts much like an X server in X Windows. As each application starts in X Windows, it asks the server to create a window, as well as objects inside that window, and it expects the server to notify it of user events for those objects and windows. The server provides services for the display screen, and implicitly for the user. Vista acts more explicitly as a display server for the participant, but the idea is the same. Components such as VRIDES and Steve ask that objects be created in the scene. Vista can build objects from graphical primitives, and it can also load them from various file formats, most prominently VRML 2.0. When the participant interacts with these objects, Vista sends event messages to the communications bus, and these messages are broadcast

to other components that are interested. Participants can interact with objects using a variety of devices (e.g., Flock of Birds™ position sensors and the Virtex Cyberglove™), but the details of these devices are abstracted out by Vista in order to provide a generic set of interaction messages (e.g., selection of an object). Steve can send these same messages, and hence can interact with objects in all the ways that human participants can. Components such as Steve can also register interest in particular objects, in which case Vista will send messages when these objects come into or go out of a participant's field of view. Thus, Vista serves a dual role: it makes the virtual world real for students, and it informs the other components of the students' actions and field of view. The information provided by Vista is crucial for allowing agents such as Steve to monitor the behavior of students.

TScript, which stands for Training Script, is the collection of messages that the components use to communicate. Object creation, modification, and deletion are controlled by TScript messages. Participant control for the purpose of instruction, such as changing a particular student's view or moving them along a path, are controlled by Tscript messages. Vista Viewers send notification of participant actions, their selection of objects, their movements in the world, and events they cause to happen indirectly all as Tscript messages. Each TScript message consists of a message name and arguments. The TScript protocol is extensible; the communications bus is not restricted to a fixed set of message types. Each component on the communications bus is free to define the set of TScript messages it can provide, and other components are free to register interest in the messages they need.

During a training session, much of the message traffic comes from VRIDES. Since VRIDES controls the behavior of objects in the virtual world, it must inform the other components when attributes of these objects change. Vista Viewers register interest in visual attributes, such as the location and color of objects. Steve agents register interest in the attributes they wish to monitor, primarily those that describe the state, rather than the appearance, of objects. For example, in the case of the High Pressure Air Compressor, this includes information such as the pressure in cylinders, and whether lights are on or off. Each component receives only those messages that are relevant to it. Moreover, VRIDES only broadcasts changes in those attributes in which some component has registered interest. Such efficiency in message traffic is crucial for handling complex, dynamic worlds.

To illustrate how the components have been integrated using the communications bus, we close this section with a brief description of starting the system for one student, in the High Pressure Air Compressor domain, and give examples of how the components interact with TScript messages. Initially, a student starts up a Vista Viewer, which shows an empty 3D scene to the student. Then VRIDES is started, and the HPAC course is selected. The 2D control panel for VRIDES appears, as does a small window with an "Initialize" button. The student presses that button, and VRIDES creates all the simulation objects and sends TScript messages to create corresponding 3D representations in Vista, either from graphic primitives or from pre-existing 3D model files. Then Steve is started, and its interface appears. Steve uses the communications bus to register interest in particular attributes of relevant objects, and it sends messages

requesting the initial state of these attributes. VRIDES responds with Tscript messages describing the state of those attributes.

Now the student is through with setup and starts the course using VRIDES. They put on the virtual environment gear and select Start from the palette in their (virtual) left hand, immersed using Vista. VRIDES progresses through the course, sending voice commands over the communications bus to the student's Trishtalk, which generates speech for the student. The student carries out requested actions, such as pressing buttons and opening valves. At each student action, Vista informs VRIDES and Steve, and VRIDES then can use the action to determine if the simulation state has changed. If it has changed, and Steve registered interest in the changes, they are broadcast to Steve.

At some point in the lesson, VRIDES may send a message to request that Steve monitor the student performing a task. When the student needs assistance, he can touch a button on Steve's interface palette to ask a question, which causes Vista to send a message to Steve. Steve can answer the question by sending text to the student's Trishtalk, causing speech to be generated.

At other points in the lesson, VRIDES may send a message requesting that Steve demonstrate a task, or the student may request a demonstration directly by touching a button on Steve's interface palette. Steve then carries out the task by sending messages to manipulate objects (these messages are handled by VRIDES) and messages to move its own visual representation (i.e., body or hands) in the virtual world (these messages are handled by Vista). VRIDES responds to Steve's actions by changing the state of the world and sending messages describing the changes.

All these interactions between components are carried out using TScript messages on the communications bus. There may be more than one student, each with their own Vista Viewer and Trishtalk, there may be more than one VRIDES, each controlling the behavior of a different set of objects, and there may be more than one Steve agent, each monitoring different students or demonstrating different tasks, but all the components maintain a consistent view of the virtual world via messages on the communications bus.

5. Steve's Architecture

As shown in Figure 8, a Steve agent consists of a cognitive component, a perceptual component, and a motor component. The cognitive component is responsible for high-level control of behavior, e.g., determining whether to demonstrate or explain, and deciding what task steps to perform. It also performs situation assessment, determining whether or not the goals of the task and the conditions necessary for task performance are met. The perceptual component provides the cognitive component with a model of the environment, which it updates and maintains based upon event notifications that are broadcast across the communications bus. The motor component receives commands from the cognitive component to perform motor actions, such as gazing at or grasping an object, computes the graphical operations that must be performed on the agent's graphical realization and the manipulations that must be performed on the objects, and broadcasts

these commands over the communications bus, so that VRIDES and the Vista Viewers on the network can update the scene appropriately. It operates asynchronously, notifying the cognitive component when a given action is complete, and is capable of performing multiple actions simultaneously, e.g., speaking while pointing at an object.

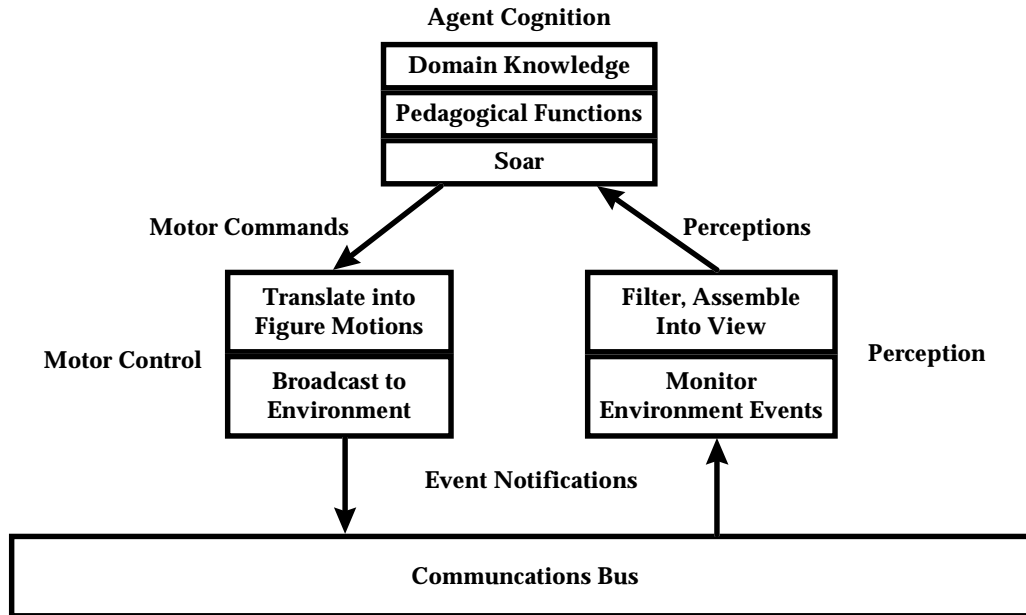


Figure 8: Steve's Architecture

The cognitive component of the agent is built on top of the Soar architecture. Soar has a number of characteristics that make it well suited for constructing agents such as Steve. It explicitly represents both task knowledge, in the form of operators, and control knowledge, in the form of search control rules that select among operators. Soar repeatedly reevaluates which operator to select, based upon the current state of the world, between ten and a hundred times per second. Soar is thereby able to respond quickly to changing situations, switching from one operator to another as necessary. However, unlike some reactive architectures such as Brooks's subsumption architecture (Brooks 1986), Soar maintains an internal symbolic mental model of its own goals, and of the environment. This simplifies the process of modeling complex cognitive capabilities. Among other things, it permits Steve to maintain in its working memory a model of the current status of the task being performed, and to explain the task to students. Soar also has a built-in learning mechanism called chunking that enables Soar agents to improve their performance over time, and acquire new knowledge. Soar is currently being used to build synthetic combat agents for distributed battlefield environments (Tambe, Johnson, Jones, et al. 1995); many of the lessons learned from that project have been put to profitable use in Steve.

Soar is a very general cognitive architecture that can model expert performance as well as novice behavior. It does not provide built-in support for pedagogical capabilities such as explanation. Such capabilities must be added to a given Soar agent. In order to support pedagogical interactions within Steve, a set of modules was developed in Soar that

provides domain-independent implementations of a number of cognitive capabilities useful for pedagogical interaction. These capabilities include episodic memory, explanation, demonstration, student monitoring, and learning new tasks from examples.

The perceptual component receives the following types of messages over the communications bus: changes to the attributes of objects in the world; actions taken by students and other agents, such as selecting or touching an object; the position, orientation, and field of view of the participants; synchronization messages indicating the beginning and end of atomic state transitions in VRIDES; and the current time as measured by a global clock. Based upon this information the perception module updates a symbolic model of the environment, in which each object in the environment is represented symbolically as an identifier with a set of attributes. The perception module also maintains a list of significant events that occurred, in particular actions performed by other participants. At the beginning of each Soar decision cycle, when Soar reevaluates which operator to select, it obtains from the perception module the current world state, together with the list of significant events that occurred since the last decision cycle.

Motor commands issued by the cognitive component fall into generic categories: manipulating objects, such as pushing buttons and grasping objects; gazing at objects, people, or agents; pointing to objects; moving from object to object; and speech. These motor commands are defined independent of the particular body realization being used. How these commands are handled depends upon the type of body realization being employed. For disembodied hands and other body parts, it selects from among a set of predefined geometric models for the body part – a hand pointing, a hand grasping, etc. It computes the appropriate position and orientation of the body part, based upon the position and orientation of the object being manipulated and the position of the user. For example, if a student is viewing the object from an oblique angle, and Steve wishes to point to it, the hand is oriented where possible to make it perpendicular to the student's view direction. The appropriate commands for selecting the body parts and sending their position and orientation are then sent out on the communications bus, so that the Vista Viewers can render them.

The layered agent architecture of Steve is in some respects similar to the ALIVE system (Blumberg & Galyean 1995), which also defines motor commands abstractly and supports multiple agent realizations. It is also related to the RAP architecture (Firby 1994). Like RAP, Steve represents plans and goals in its cognitive component, and sends action commands to a lower execution layer. Both can perform multiple actions concurrently, and permit concurrent execution of the cognitive and motor layers.

Overall, the trickiest practical problem with implementing this agent architecture has been integrating the various software systems: Steve, Vista, VRIDES, and Jack. As new capabilities were developed for Vista, VRIDES, and Steve, supporting capabilities in the other subsystems had to be developed as well, as well as new Tscript messages for requesting services and reporting their results. In order to ensure that the agents operate on consistent states, environment state changes are grouped into atomic transactions, and special Tscript messages were defined to signal the beginning and end of transactions.

6. Modeling Tasks and Domains

Steve's pedagogical capabilities make use of a standard hierarchical plan representation of tasks (Russell and Norvig 1995). Each plan has a set of goals that it achieves, and a set of steps for achieving the goals. Steps can be either primitive or complex; complex steps in turn are implemented via other plans. Primitive steps can perform actions, such as grasping an object, and draw inferences, such as concluding that a subsystem is operational. Ordering constraints define a partial ordering between steps. Causal relations can also exist between steps: a step can cause a condition to be true, which in turn is a precondition for a subsequent step.

Figure 9 shows an example of a plan in Steve's representation: the plan for performing a functional test of one of the subsystems of the HPAC. It consists of three steps: **press-function-test**, which presses the Test button on the control panel and causes the front panel lights to turn on, **check-alarm-lights**, which examines the lights to make sure they are functional (not burned out), and **extinguish-alarms**, which resets the lights. In addition, every task has two dummy steps: a **begin-task** that precedes all other steps, and an **end-task** that follows all other steps. A number of causal links exist among the plan steps; for example, **press-function-test** puts the device in **test mode**, which is required for **check-alarm-lights**. In order for the task to be complete, the operator must know whether the alarm lights are functional, and the alarm lights must be off. Thus, these goal conditions are shown as prerequisites for **end-task**. Similarly, if the task depended on conditions that must be established prior to starting the task, these conditions would be represented as effects of **begin-task**.

The same representation can be used to represent methods for handling contingencies. Contingent steps are simply represented as additional steps in the plan, along with the goals they achieve. Steve's flexible plan execution mechanism, described in the next section, skips contingent steps if the goals that they are intended to accomplish are satisfied. Thus, the task description can represent the full task, and Steve automatically decides which steps are relevant in a given situation.

In addition to specifying the plans needed to complete a task, one must also define the goal conditions and primitive actions that appear in the plan. Figure 10 shows examples of goals and primitive actions. The first goal, **test-mode**, is true when the value of the **hpac-mode** attribute is **test**. **Know-whether-alarms-functional** holds when there is some value for the **check-alarm-lights-result** attribute. The syntax for these structures is based upon the syntax used by Soar. The token <p> indicates that the attribute **hpac-mode** is a perceptual attribute, which will be provided by the perceptual component of Steve, based on information that Steve receives from the VRIDES model of the HPAC. The token <m> indicates that the attribute is part of Steve's mental state, i.e., a property that Steve infers instead of receiving directly from the perceptual component.

Plan: functional-check

Steps:

- press-function-test
- check-alarm-lights
- extinguish-alarms

Causal links:

- press-function-test achieves test-mode for check-alarm-lights
- check-alarm-lights achieves know-whether-alarms-functional for end-task
- extinguish-alarms achieves alarms-off for end-task

Ordering:

- press-function-test before check-alarm-lights
- check-alarm-lights before extinguish-alarms

Figure 9: Example plan representation

Goal: test-mode

condition: (<p> ^hpac-mode test)

Goal: know-whether-alarms-functional

condition: (<m> ^check-alarm-lights-result)

Primitive plan step: press-function-test

type: press-button

button: function-test-button

Figure 10: Goal and plan step definitions

The figure includes a specification of a primitive plan step, in this case pressing the function test button. The step is an instance of a press-button operation, applied to a specific button. VRIDES simulation models are built out of classes of reusable components, such as buttons, and in a similar manner plan steps are constructed as instance of reusable action types. This greatly simplifies the process of extending Steve to new tasks.

A number of aspects of this plan representation are designed to support authoring, although not the syntax, which will be replaced by a graphical interface. Tasks are described as a hierarchy of plans. Plans are built from a simple, uniform representation. Plans are constructed out of reusable elements. The Soar implementation level is largely hidden, allowing instructors to focus on modeling the task. Instead, the necessary Soar productions are generated automatically from the plan descriptions. This should offer advantages over rule-based authoring languages such as the rule language used in Anderson's Lisp Tutor, (Anderson et al 1990), which does not represent the kind of causal information that Steve captures, and thus is less suited to explanation generation.

Finally, the representation supports automated acquisition from examples, as will be described in section 7.4.

7. Exploiting Task Knowledge

This section describes in more detail how each of Steve's pedagogical capabilities is accomplished, and how the task representation described above supports them. The first capability to be discussed is demonstration.

7.1. Demonstration

Demonstrating tasks involves carrying out tasks for the benefit of the students, so that the students can understand how to perform the tasks, and so can learn to perform the same tasks. In order to accomplish this successfully, Steve's demonstration capability must meet several requirements. It must be suited to dynamic virtual environments, in which objects may change state unpredictably, and where multiple students and agents may be active at the same time. It must be efficient enough to be performed quickly, so that the students are not left waiting while Steve plans what to do next. In addition to these performance-oriented requirements, there are additional requirements that arise due to the pedagogical context in which the demonstration takes place. Steve should give preference to the use of standard procedures, even when there are alternative methods that can get the job done. This can help students watching Steve to learn the conventional practice of the domain. At the same time, Steve should be able to articulate its decision making and the rationales for its actions, to provide a running commentary on the demonstration as well as to answer any student questions. It is essential that students learn the rationales for the steps in the procedures, so that they can recognize when changing circumstances dictate adjustments to the standard procedure. These pedagogical requirements are not often addressed in autonomous agents for virtual environments, where emphasis is usually placed on generating behavior that appears lifelike from the standpoint of a passive observer.

To meet these objectives, Steve executes procedures in the following manner. First, Steve constructs an overall plan for performing the tasks, using a top-down decomposition approach (Sacerdoti 1977). That is, Steve repeatedly expands any complex steps in the evolving plan with the subplan (given to Steve as task knowledge) for achieving it, until the plan has been fully decomposed into primitive actions. The resulting plan includes all steps that might be required, even if some are unnecessary given the current state of the virtual world. This plan obeys the ordering constraints in the task model, ensuring that it conforms to standard operating procedures. However, when Steve carries out the task, it does not blindly execute the plan. Instead, Steve continually evaluates the relevance of each plan step to the current situation. It skips steps that are no longer relevant. It also may repeat steps when necessary, if the step did not have the expected effect or if the environment changed in some way so that the desired effect is no longer maintained. The plan thus serves as a resource that Steve can refer to when deciding what to do next, and Steve can make use of the plan in different ways as the situation dictates.

The method that Steve uses for deciding which plan steps are relevant in the current situation is related to partial order planning (Weld 1994). A partial order planner starts with the end goals of the task and adds steps to the plan in order to achieve those goals. Each step that is added may have unsatisfied preconditions, and each such precondition becomes a new subgoal that must similarly be achieved. Steve follows this same procedure. But whereas partial order planners in general might have to perform an extensive search among alternatives in order to determine an appropriate method for achieving the subgoals, we assume that the task model has been authored in enough detail that Steve can find within the plan appropriate methods for achieving the subgoals. Steve therefore simply marks each step in the plan as relevant or irrelevant in the current context. Implementation of this technique makes use of the Soar architecture's ability to respond dynamically to changing situations. Soar productions are used to test the relevance of each plan step. These productions are continually reevaluated based on the changing input from the environment. For each step that is currently relevant, a Soar operator to perform that step is proposed. Soar's existing mechanism for selecting from among proposed operators is used in order to pick which step to perform next and apply it.

Steve's plan construction and execution methods draw on ideas from the IPEM architecture (Ambros-Ingerson & Steel 1988), which seamlessly integrates construction, execution, and revision of plans. The IPEM control strategy can be easily adapted to perform reactive planning, partial-order planning, task-decomposition planning, or a combination of the above. Steve currently adopts one particular approach to combining plan construction and execution, but alternative methods could be easily chosen if this is found to be necessary when applying Steve to other application domains. For example, when using Steve in team tasks, we expect that it will be unnecessary for each Steve agent to construct a plan encompassing the actions of all of the members of the team. However, if one of the student participants in the team requests help from Steve, Steve should then extend its plan to include possible actions of the participant asking for help, so that he can then show the student how to overcome his or her current difficulty.

While Steve demonstrates a task, he continually explains what he is doing. If he is about to perform a complex subtask that requires multiple steps to complete, he first mentions the subtask in spoken language (e.g., "I will now perform a functional check of the condensate drain monitor."). Each primitive step is described as well, both steps that operate on the environment (e.g., "Press the function test button.") and steps that are concerned with assessing the current situation (e.g., "Look at the alarm lights. All lights should be illuminated."). Before operating on an object in the virtual world, Steve points at the object with his hand. If Steve draws some conclusion based on observations of the environment, these are mentioned as well (e.g., "The lights are illuminated, so they are all working properly.").

Another important feature of Steve's demonstration capability from a pedagogical standpoint is his ability to remember what actions he took, so that he can answer student questions afterwards. Steve employs the episodic memory capability that was developed for Soar agents as part of the Debrief system (Johnson 1994). Debrief is a generic capability that enables Soar agents to remember their actions and explain them to users.

It has also been used to enable Soar agents to learn from their mistakes, by comparing episodes in which decisions led to undesirable outcomes to other situations in which a similar decision led to a favorable result (Tambe, Johnson, and Shen 1997). In Steve, Debrief is used to commit situations to memory, via Soar's chunking mechanism. Later, if a student has a question about why a particular action was taken, Steve recalls from its memory the situation in which the action was taken. Steve can then use its explanation capability, described below, to discuss with the student which actions are appropriate to take in that situation.

7.2. Student Monitoring

Once the student has acquired some familiarity with the skill being trained, Steve lets the student take the lead in carrying out the task. Steve must monitor that student's progress, so that if the student encounters a difficulty or has a question, Steve can offer assistance. Steve's task model supports student monitoring as well as demonstration. Steve can switch back and forth between monitoring and demonstration as needed - the student may ask Steve to demonstrate a particular step within the procedure, and then take over from there.

In dynamic virtual environments, it is in general necessary to monitor both the student's actions and the state of the environment in which the actions are being taken. To put it a different way, student monitoring is situated in the virtual environment. Our approach places primary emphasis on monitoring the virtual environment, so that Steve can provide assistance that is appropriate for the current situation. As we have argued elsewhere (Hill & Johnson 1995), focusing on the student's situation, as is possible in simulation-based learning environments, permits the tutoring system to devote less effort to tracking student actions, compared to conventional plan recognition (Johnson 1986) and model tracing (Anderson et al 1990). When students request help, it is likely to be aimed at overcoming whatever difficulty the students have encountered in the current situation; thus knowledge of what actions are appropriate in the situation has primary importance. The simulation environment provides the students with feedback on their actions; if the student takes an incorrect action, the simulation may respond by going into a state that the student did not expect. Therefore the tutoring agent need not devote as much effort to tracking the student's individual actions and informing the student when they are incorrect.

The current version of student monitoring in Steve focuses entirely on evaluating the situation. The method is closely related to Steve's demonstration capability. When monitoring begins, Steve is given the name of the task that the student will be working on. It then constructs a plan for performing the task, and determines the relevance of each task step in the current situation. But unlike in demonstration, Steve does not carry out the plan itself. Instead, as the student carries out the task, Steve updates its plan to indicate which steps are currently relevant to completing the task. Although Steve receives information from Vista about the actions that the student is taking, Steve updates his plan model only when the action's effect on the simulation is observed. The only plan steps that Steve carries out itself during student monitoring are sensing actions (e.g.,

checking whether alarm lights come on); Steve assumes that the student is performing the same sensing action, and is drawing the same conclusions based upon it.

Using this approach, Steve is able to generate advice for the student that is relevant to the current situation. However, it suffers from some limitations that we will seek to correct in future versions of Steve. Students may desire different amounts of assistance from Steve at different times. A natural way of providing such variation is to provide students a way of asking Steve to “watch” what they are doing. When Steve is actively watching, he would be expected to be more pro-active in pointing out inappropriate actions to the student. Even when Steve is not actively monitoring the student, it would still be useful for Steve to note which subtask the student is currently working on. Then if Steve has multiple recommendations that it might make, it could give priority to those which are part of the current subtask. Steve could also focus its intervention on impasse points (Hill and Johnson 1995), i.e., situations where the student is making mistakes and appears to be unable to correct them. These and other enhancements can be made to the student monitoring capability without fundamentally altering the approach.

7.3. Explanation

Steve’s explanation capability is essential for imparting to students a good understanding of the tasks that they are learning to perform. Because Steve can answer questions about why actions are appropriate, Steve can help students to learn the causal relationships between task steps, and between tasks and goals. The explanation capability depends crucially upon the causal relationships that are present in Steve task models.

Figure 11 shows Steve’s answers to a sequence of “why” questions. Here the student has asked for advice regarding what action to perform next in the task of testing the HPAC condensate drain monitor. Steve has responded by suggesting an action, press the function test button. After this is a series of follow-on “why” questions, which result in explanations in terms of the desired state of the device, subsequent steps that are enabled by this action, and conclusions that may be drawn from them. This sequence can continue until the student is satisfied with the explanation, or the student asks for an explanation of one of the top-level goals of the task, which Steve takes as given and therefore cannot explain.

Student requests help

I suggest you press the function test button.

Student asks “Why?”

That action is relevant because we want the drain monitor in test mode.

Student asks “Why?”

That goal is relevant because it will allow us to check the alarm lights.

Student asks “Why?”

That action is relevant because we want to know whether the alarm lights are functional.

Figure 11: Question answering with Steve

Steve generates these explanations directly from the causal structure of the plan that it is demonstrating or monitoring, along with its assessment of which steps and goals are relevant to completing the task. If Steve considers a step in the plan to be relevant, it is because it achieves a goal that is also relevant. A goal is relevant if it is a precondition for a later plan step, or because it is a top-level goal of the task. Each of these relations is determined from the causal links in the plan specification.

A similar mechanism is used by Steve to explain actions that it took previously as part of the demonstration. If the student asks about a previous action, Steve recalls the situation in which the action was taken, using its episodic memory. It then determines the relevance of each action and goal in the recalled situation. The action that Steve actually took will necessarily be one of the relevant steps. Steve can then generate explanations in terms of the relevant actions and goals that are causally related to the recalled action.

The current approach to generating explanations, to work up the causal chain, is not always the ideal approach. From the human-machine dialog perspective, it sometimes states the obvious (for example, the purpose of pulling on the dipstick is to remove the dipstick from the HPAC). From a pedagogical perspective, it might be better to give students some general hints and have the students figure out the causal relationships themselves. These more sophisticated question answering strategies are supported equally well by Steve's causal task representation, so we are in a good position to investigate such alternative strategies.

8. Acquiring Task Knowledge

Agent-based approaches to training will prove to be effective only if it is easy to create instructional materials that employ them. Sophisticated intelligent tutoring systems usually require expertise in artificial intelligence programming techniques. Virtual reality systems frequently require expertise in computer graphics and networking. By combining networked virtual environments with intelligent agents, we run the risk of requiring a skill set that very few individuals have. We have therefore taken a number of steps to simplify the task of authoring agent-based virtual training environments.

First, many of the facilities that RIDES provides for authoring instructional simulations are also available in the VRIDES system. Using VRIDES, one can first create a 2D simulation of a device, and then link the 2D simulation to the 3D environment. This is accomplished by adding rules to the RIDES simulation that send and receive notifications to and from Vista as objects are manipulated. Furthermore, VRIDES supports the creation of simulations from a library of reusable components. Common objects such as buttons, switches, gauges and lights have been defined, and were used in the creation of the HPAC simulation.

In a similar vein, Steve's behaviors are constructed from reusable primitive actions. Each action applies to a class of objects, such as buttons and gauges. Steve's behavior can be generated by applying reusable actions to instances of reusable VRIDES objects.

The next level of authoring support provided for Steve is at the task level. Most other systems that provide authoring languages for procedural skills, such as RIDES and VRIDES, Rickel's TOTS shell (Rickel 1988), and Billinghurst and Savage's sinus surgery trainer (Billinghurst & Savage 1996), do not attempt to represent causal relationships between steps, and instead focus on ordering constraints and task hierarchies. As discussed in previous sections, Steve's knowledge of causal relationships provides considerable power: using this knowledge, Steve can adaptively execute plans, as well as explain the rationale behind actions and goals. However, requiring this knowledge places an extra burden on instructional developers; although we expect that instructional developers can learn to use Steve's task representation, we have little evidence to support this conjecture at the present time. Instead, Richard Angros in our group has been investigating the use of machine learning techniques to automate the construction of task models from examples.

The approach builds heavily upon previous work by Xuemei Wang on learning via observation (Wang 1995), as well as on previous work on learning from problem solving failures (Tambe, Johnson, and Shen 1997). It is also inspired by Gil's work on learning via experimentation (Gil 1993). The system learns by watching a human instructor demonstrate the task, and then experimenting on its own with variants of the demonstration. Based on the demonstration and the experiments, the system is able to construct a generalized task model, including both causal relations and ordering constraints. The technique is conservative, generalizing the task model only when there is evidence in support of the generalization. As is common in programming by demonstration systems (Cypher 1993), the instructor is given the opportunity to check the system's task model, and further generalize the task model if appropriate. Assuming that the author has selected examples that clearly demonstrate the procedure, it is possible to acquire complex causal task models in this manner, with only a moderate amount of author intervention and guidance.

A typical use of the system proceeds as follows. The author first puts the simulation in some neutral initial state, and informs the agent that demonstration is about to begin. The author then proceeds to perform the procedure step by step, typically using a Vista Viewer in desktop mode. The primitive actions that the author performs are expected to be instances of action classes that are already known to Steve; however, the effects of the actions may not be known. For example, although Steve understands the act of pressing a button, it will not know the effect of pressing a particular button. Whenever the author performs an operation on a new object for the first time, Steve prompts the author for the name and textual description of the operation. The author then signals Steve when the demonstration is finished.

While the demonstration is taking place, Steve notes the changes in VRIDES simulation state. Steve assumes that the simulation proceeds through a series of atomic state changes and that the state changes are a result of the actions that the author is performing during the demonstration. After the demonstration, Steve presents to the author a list of the object attributes whose values are different from when the demonstration started. The author then selects from these the subset that are the desired effects. These then become the goals of the procedure.

As an example, consider the following procedure. Before starting up the HPAC, suppose that two of the valves on the separator drain manifold, shown in Figure 12, must be closed. The author demonstrates the procedure for Steve. The author initially puts the device in a state where all of the valves are open, and the valve handle (the dark handle with two cross bars in Figure 12) is sitting on some valve other than the valves to be closed. The demonstration consists of moving the valve handle to the first valve, turning it, then moving the valve handle to the second valve and turning it. When the author indicates that the demonstration is complete, Steve displays the state changes that resulted: the two valves are now shut, the valve handle is sitting on the second valve, and the handle is in the shut position. The author indicates that the state of the valves are important, and are part of the goal state; the location and position of the valve handle is not. Steve must now determine how the author's actions caused the valves to change state, and what causal relations and ordering constraints exist in the procedure.

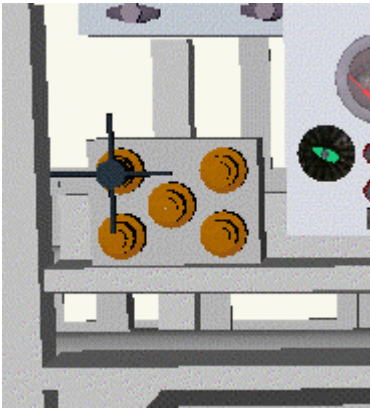


Figure 12: The HPAC separator drain manifold

Following Wang's OBSERVER model, which builds on Mitchell's version space approach (Mitchell 1982), Steve constructs a most general and most specific model for the preconditions and effects of each operator. The most general model contains the conditions that Steve has determined must always hold for the procedure to be performed successfully, the effects that always result from performing the procedure, and the causal relations and constraints that Steve has determined exist. The most specific model includes all conditions and effects that are observed in the demonstration example, and which have not been shown to be irrelevant, together with all causal relations that might exist given the available evidence. In addition to these models, Steve also constructs a heuristic model which represents Steve's best guess concerning the conditions, effects, causal relations, and constraints. The differences between the heuristic model and the general and specific models constitute conjectures that Steve can try to verify experimentally.

The current version of the knowledge acquisition tool focuses on one type of experiment: to delete a step from the procedure and see what effect this has on the outcome of the procedure. This proves to be an effective way of determining whether or not constraints

exist among steps. To perform these experiments, Steve repeatedly resets the simulation back to the initial state indicated by the instructor,³ selects a step to delete, and tries performing the procedure while omitting the step. This process succeeds in eliminating many spurious causal relationships between steps. Further improvement should be obtainable through further experimentation, and through the use of multiple examples. The following example illustrates the kinds of spurious relationships that arise, and the techniques needed to eliminate them. In the separator drain manifold example, the instructor always moves the valve handle to a valve in order to shut it. Steve therefore infers that moving the handle to a valve always results in the valve handle being in the open position. However, if the instructor proceeds to demonstrate moving the valve handle to a closed valve, Steve recognizes that the effect does not always hold, and generalizes its model of the task accordingly.

The current version of demonstration-based authoring has the following limitations. It does not learn hierarchical procedures; without a mechanism for aggregating sequences of steps into subtasks, the task descriptions tend to become very complex as the number of steps increases. The system is only able to learn from one demonstration example at a time. Its repertoire of experiments is very limited, and is unable to plan experiments to resolve specific ambiguities in the current task model. These issues are currently being investigated, and we hope to incorporate solutions soon in preparation for usability studies with potential users.

Once the task model has been constructed, the instruction author needs to make decisions about how Steve should be used in specific lessons: what tasks or subtasks will be covered, what level of proficiency is expected of the student, and what pedagogical roles Steve should play. These issues are being addressed in the overall context of designing instruction in VET, as described in the next section.

9. Instruction and Instruction Authoring

In the VET project, several different types of instruction can be developed and presented to students. Three modes of simulation-centered student interaction are supported: immersed (e.g., with a head-mounted display), 3D-in-a-window, and simple 2D graphics. Several different types of lessons can be presented, some of which provide strict, step-by-step authored control over the student, while others offer less constrained exploration supported by expert advice and guidance.

Instruction is the responsibility of both VRIDES and Steve. VRIDES handles the sequence of lessons and the presentation of pre-authored structured lessons. Steve handles the presentation of lessons centered on a deep representation of procedural expertise for the domain.

Three layers of instruction development tools and techniques are available for VET development:

³ At the present time, Steve has no way of automatically resetting the simulation to a previous state, and has to ask the instructor for help resetting the state. An automatic reset feature is currently being incorporated into VRIDES for this purpose.

- Instructional planning and objectives specification
- Defining structured lessons
- Defining procedural expertise for agent-guided lessons

Techniques for defining procedural expertise were described in the previous section; this section focuses on the other two layers.

Instructional objectives. The author of a VET course can begin by specifying the objectives that the course is designed to address. A *Courses and Objectives* authoring interface supports this level of specification. One aspect of objectives authoring is to specify which objectives must be attained as prerequisites to the attainment of other objectives. This information will later be used, in conjunction with the individual student's performance data, to decide, every time a student finishes a lesson, what lesson should be presented next.

Structured lessons. A wide variety of structured lessons can be authored using VRIDES. Seventeen types of common lessons can be authored very easily in a very high-level authoring interface. These lessons include some that teach students about the components and their positions, some that teach about how to carry out procedures, and some that teach how to troubleshoot the equipment. (Naturally, not every type of lesson is generated for every training application. Authors must decide which lesson types to make use of when specifying their objectives for a given domain.) In addition to these common types of tutorials, authors can use a more low-level authoring interface to build interactive structured lessons. Such lessons are composed of combinations of twenty-four primitive interaction element types, such as "require that the student point to a particular item", "pose the following question and menu of possible answers", and "highlight the following object".

VRIDES lessons can be authored for delivery in either 3D or 2D graphical environments. Although the purely 2D versions of VET lessons cannot be used to provide realistic orientation training, in many domains they can be used to deliver useful part-task training, as a supplement to 3D training. Ordinarily, a lesson authored for delivery in a 3D environment can be used in either an immersed or a 3D-in-a-window presentation.

9.1. Related Work

VRIDES fits into a long tradition of learning environments based on interactive graphical simulation. For example, STEAMER (Williams, Hollan, and Stevens, 1981; Hollan, Hutchins, and Weitzman, 1984) provided a direct manipulation simulation for students learning about a steam propulsion plant. It offered a discovery world for students and a demonstration platform for instructors, but it did not provide authoring tools for the development and delivery of instruction to the learner. IMTS (Towne and Munro, 1988) provided tools for authoring interactive graphical simulations of electrical and hydraulic systems, but the model authoring approach was limited. IMTS supported troubleshooting assistance by a generic expert called Profile (Towne, 1984), but it could not be used to develop or deliver other kinds of instruction.

An early approach to direct manipulation instructional authoring was Dominie (Spensley & Elsom-Cook, 1989). That system, however, did not support the independent specification of object behaviors; the specification of simulation effects was confounded with the specification of instruction.

RAPIDS (Towne, Munro, Pizzini, Surmon, Coller, and Wogulis, 1990; Towne and Munro, 1991) and RAPIDS II (Coller, Pizzini, Wogulis, Munro, and Towne, 1991) were descendants of IMTS that supported direct manipulation authoring of instructional content in the context of graphical simulations. These systems provided a much more constrained simulation authoring system than is found in VRIDES, and they did not provide access to low level of control instructional presentations.

RIDES (Munro, 1994; Munro, Johnson, Pizzini, Surmon, & Wogulis, 1996; Pizzini, Munro, Wogulis, & Towne, 1996) provided much more robust simulation authoring and instructional editing facilities than were to be found in RAPIDS and RAPIDS II. The RIDES system has some features in common with the SMILSE system (de Jong, van Joolingen, Scott, deHoog, Lapied, & Valent, 1994) developed by a consortium of European academic and industrial research groups, but is less restrictive on how simulations can be structured. SMILSE authors must separately specify an inner, 'real' level of behavior and one or more surface depictions of the behaving system. Similar effects can be achieved using RIDES, but they are not required. The SMILSE system also contains additional facilities for supporting student hypothesis formation, but lacks some of the open-ended authoring character of RIDES.

VRIDES, the simulation authoring and structured instruction authoring and delivery system of VET is an extension of RIDES. VRIDES supports the communication of simulation-relevant information both with Vista Viewers and Steve agents. It also provides facilities for sequencing lessons in a course.

9.2. Authoring Objectives for Course Control in VET

A course is defined as a set of objectives, together with a specification for how well a student must perform on each objective in order to progress to the next objective in the course. Objectives themselves are defined in terms of a set of attributes and a criterion lesson, together with a specification of how well the student must perform on the criterion lesson in order for the objective to count as having been achieved. In addition, objectives can be specified to have enabling or prerequisite objectives.

Figure 13 displays portions of three editors that play a role in defining course behavior using VRIDES. The deepest window is the list of courses and objectives that are associated with a particular simulation. A course consists of a list of references to objectives, together with the performance that will be required of students for each objective in that course. The same objective may be appear in several different courses, but with different required performance levels. The next deepest window, shown partially obscured at the bottom of Figure 13, is the Course editor, which lists a course's objectives with the performance level required. The top window shown is the Objective editor, here displaying an objective for carrying out the Pre-Start Check procedure with Steve's

guidance. This objective has two prerequisite objectives: one specifies that the Locations objective be attained at the 50% performance level; the other requires that the PracticePreStartCheck objective be attained at the 65% performance level.

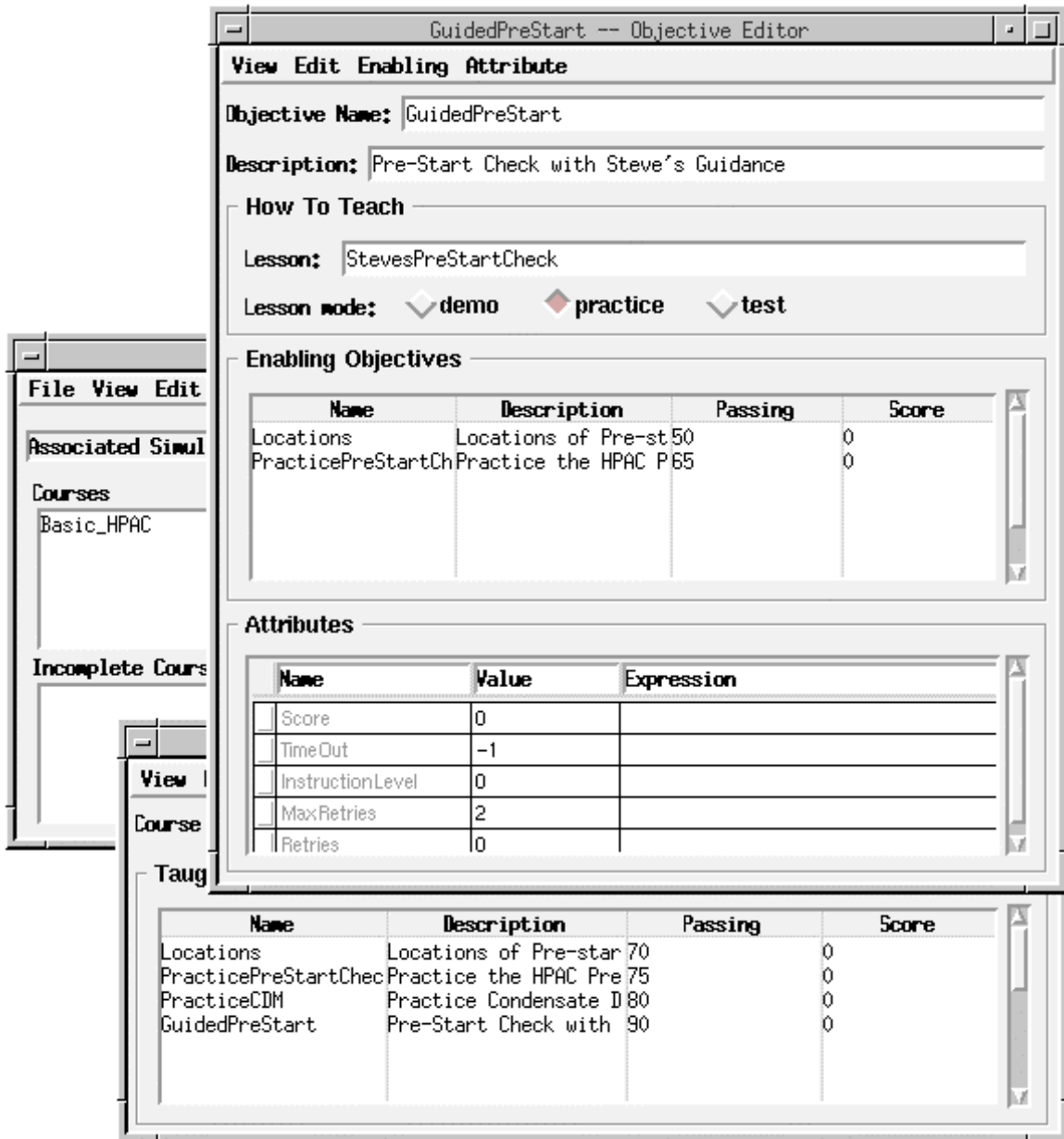


Figure 13: Course objectives authoring interfaces

Objectives, like instructional items in VRIDES, have attributes, which specify and record information about how many times a student may attempt the objective, how many times the student has attempted it, how well the student did, and so on. A normative student model is defined based on the attributes of the objectives of a course. As a student completes a course objective, the student model attributes that reflect performance on the objective are updated. Then the next lesson is selected based on the performance data

captured in his or her student model and on the prerequisite relationships that hold among the objectives.

9.3. Defining Structured Lessons

Structured lessons are easily authored, but they do not provide the same degree of flexibility that agent-based lessons do. Nonetheless, they provide a number of types of lessons that are very useful in many simulation-based training contexts, and they are likely to often be used in conjunction with agent-based lessons.

Two approaches to the authoring of structured lessons are supported. In the *patterned exercise* approach, authors choose one of seventeen types of standard lessons. After the author makes a few specifications, a lesson of the selected type is automatically generated. For example, there are several types of patterned exercises that are designed to teach simple facts about the objects in a simulated world, such as where the objects can be found and what their names are. To specify the contents of such a lesson, the author simply specifies the simulation state in which the lesson should begin and indicates whether any of the named objects in the simulation should *not* be included in the lesson.

Procedural types of patterned exercises can be authored easily. An author specifies the system state in which the lesson should begin and then carries out the steps of the procedure that is to be learned by the student. Figure 14 shows one of the patterned exercise editors in use. The author is defining a "Goal" type lesson. The author has specified that a configuration (a snapshot of the simulation state) named "vrconfig" should be installed at the start of the lesson. Then the author used the mouse to set a number of controls (first the drain and monitor mode selector switch, then the condensate drain operating mode switch, etc.) into their required states, in just the same way that a student would. As each simulated control was manipulated, its name appeared in the Controls field of the editor. Once the required state was achieved, the author hit the "Stop Controls" button, and then clicked the mouse pointer on an indicator that shows that the goal has been achieved. Its name, "manual-monitor override indicator light" appears in the Indicators field of the editor. If other indicators participate in revealing that the goal has been met, the author would click them, too. Finally, hitting the Save Exercise button creates a detailed, editable lesson structure based on this interactively authored lesson specification. Since this authoring method does not identify causal relations between steps, it applies to a more limited range of lessons than Steve's demonstration-based authoring method.

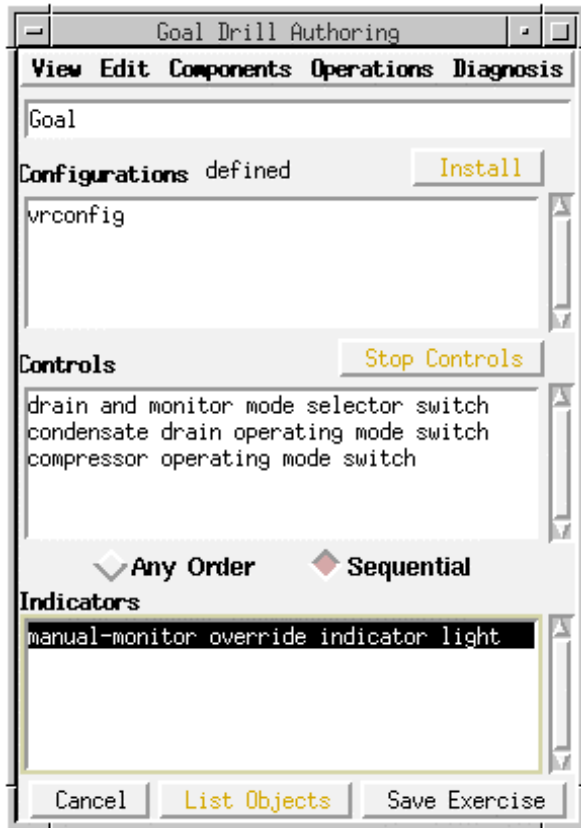


Figure 14: A Patterned Exercise Editor during the authoring of a 'Procedure' structured lesson

A second approach to developing structured lessons allows authors more detailed control over the content and sequencing of instructional items. Twenty-two elementary instructional interaction types are supported in this type of authoring. Examples of these include "Make a textual or voice presentation to the student", "Highlight an object", "Require that the student click a particular object", and "Require that the student manipulate a simulated control object to a particular value". The lessons generated by the patterned exercise editors create lessons that are composed of groups of these instructional interaction primitives. It is possible to generate a structured lesson quickly using a patterned exercise tool, and then to customize it in an instructional editor.

9.4. Integrating Agents into Lessons

At the present time, the instructional author has limited ability to control how Steve agents are used in the lesson plan. Agent-supported lessons are all classed as practice lessons: the student is given an opportunity to perform the task, and Steve monitors the student performing the task. However, it should be clear that Steve can actually support a variety of different pedagogical interaction styles, and in fact can switch styles as appropriate, e.g., switch between demonstrating a task and monitoring the students as they perform the task. To accommodate and control this flexibility, we plan to provide authors with the following types of controls on Steve's interaction. First, the author will

be able to indicate the type of lesson, including but not limited to the following types: introductory lessons and coached practice lessons. The lesson type serves as an indicator of the expected level of mastery of the student, and therefore the type of interaction that Steve should assume by default. Second, the author may specify constraints on Steve's interaction with the student. For example, the instructor might specify that the lesson should focus on a specific task, and the student should not be given the option of digressing to a different task. Without such constraints, Steve will interact with the student and guide the lesson in whatever manner the student and Steve consider appropriate.

10. Assessments and Future Work

The primary focus of effort on VET to date has been on creating an integrated system that incorporates the various technologies for virtual environments, simulations, agents, and course authoring that have been described in this paper. Now that these capabilities have been integrated, it is possible to evaluate the accomplishments so far, assess limitations, and plan future work.

Overall, the integration of virtual environments, simulations, and pedagogical agents has proved to be quite effective and successful. The capabilities of the various technologies complement each other well. Initial informal evaluations of the system confirm our expectation that the technologies are mutually supporting, and the combined system is more effective as a teaching tool than are the component technologies individually. In the case of pedagogical agents, we do indeed find that interacting with agents in the virtual environment is more natural than using conventional text-based tutoring interfaces. The ability to demonstrate tasks also gives Steve a significant advantage over conventional tutoring systems.

The support for authoring is seen as extremely important by prospective users of our system. Interestingly, this view is not equally shared among instructional system developers, who sometimes regard authoring as a relatively minor issue. Further evaluation will determine whether or not our authoring approach is really usable. The authoring capabilities in RIDES have been evaluated by a number of different organizations. However, the other authoring capabilities have been tested only to a limited degree. Armstrong Laboratory is currently undertaking evaluations of the various technologies in VET, and is attempting to apply it to other domains.

Finally, a key focus of our current work in VET is support for team training. VET can already support multiple virtual environment displays, multiple students, and multiple agents; the challenge is how to combine these effectively to support team training. Team training poses challenges both from the perspective of agent control and from the perspective of pedagogy. Each Steve in a team must be able to perform his own role within the team, monitor the activities of other team members, and to track the overall progress of the team. These issues have been investigated in the context of the Soar-IFOR battlefield simulation project, which is modeling teams of combatants (Tambe 1997); we are applying Steve to similar battlefield tasks in order to assess how to transition Steve to such team tasks. The pedagogical issues that come up have to do with

how to manage training interactions with multiple students, each with different levels of mastery of the material. Although it remains to be seen how best to accomplish this, it would appear that flexible behavior control and interaction are likely to be even more important than they have been to date.

Acknowledgments

We wish to acknowledge the efforts of the other VET project participants in this work: Richard Angros, Mark Johnson, Laurie McCarthy, Quentin Pizzini, Erin Shaw, Sandeep Tewari, and Jim Wogulis. This work was supported by the Office of Naval Research under contract N00014-95-C-0179.

References

- Ambros-Ingerson, J.A., and Steel, S., 1988. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pp. 83-88, San Mateo, CA, Morgan Kaufmann.
- Anderson, J.R., Boyle, C.F., Corbett, A.T., and Lewis, M.W., 1990. Cognitive modeling and intelligent tutoring. *Artificial Intelligence* (42), pp. 7-49.
- Badler, N., Phillips, C., and Webber, B., 1993. *Simulating Humans*, Oxford University Press.
- Barrus, J.W., Waters, R.C., and Anderson, D.B., 1996. Locales and beacons: Efficient and precise support for large multi-user virtual environments. *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pp. 204-213. IEEE Computer Society Press.
- Billinghamurst, M., and Savage, J., 1996. Adding intelligence to the interface. *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pp. 168-175. IEEE Computer Society Press.
- Blumberg, B.M. and Galyean, T.A., 1995. Multi-level direction of autonomous creatures for real-time virtual environments. *SIGGRAPH 95 Conference Proceedings*, pp. 47-54.
- Brooks, R.A., 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 2(1): 14-23.
- Calvin, J. et al, 1993. The SIMNET virtual world architecture. *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pp. 450-455. . IEEE Computer Society Press.
- Coller, L. D., Pizzini, Q. A., Wogulis, J., Munro, A. & Towne, D. M. , 1991. Direct manipulation authoring of instruction in a model-based graphical environment. In L. Birnbaum (Ed.), *The international conference on the learning sciences: Proceedings of the 1991 conference*, Evanston, Illinois: Association for the Advancement of Computing in Education.

- Collins, A., Brown, J.S., and Newman, S.E., 1989. Cognitive apprenticeship: teaching the crafts of reading, writing, and mathematics. In Resnick, L.B., ed., *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, pp. 453-494, Lawrence Erlbaum Associates.
- Cypher, A., ed., 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.
- Dewey, J., 1939. Fundamentals of educational process. *Intelligence in the Modern World: John Dewey's Philosophy*. Edited by Joseph Ratner. New York: Random House, Inc.
- de Jong, T., van Joolingen, W., Scott, D., deHoog, R., Lapied, L., Valent, R., SMILSLE: system for multimedia integrated simulation learning environments. In T. de Jong and L. Sarti (Eds.) *Design and production of multimedia and simulation based learning material*, Dordrecht: Kluwer Academic Publishers, 1994.
- Durlach, N.I., and Mavor, A.S., eds., 1995. *Virtual Reality: Scientific and Technological Challenges*. National Academy Press, Washington D.C.
- Firby, R.J., 1994. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*.
- Fogg, B.J., and Moon, Y., 1994. Computer as teammate: effects on user attitude and behavior. *Proceedings of Lifelike Computer Characters '94*, p. 54, Microsoft Research, Snowbird, UT.
- Gil, Y., 1993. Efficient Domain-Independent Experimentation. USC / ISI technical report ISI/RR-93-337. Also appears in the *Proceedings of the Tenth International Conference on Machine Learning*.
- Hayes-Roth, B. and van Gent, R., 1997. Story Making with Improvisational Puppets, *Proceedings of the First International Conference on Autonomous Agents*, ACM Press.
- Hill, R.W. and Johnson, W.L., 1995. Situated Plan Attribution, *Journal of Artificial Intelligence in Education* 6(1), pp. 35-67.
- Hollan, J. D., Hutchins, E. L., and Weitzman, L., 1984. STEAMER: and interactive inspectable simulation-based training system, *AI Magazine* 5(2), pp. 15-27.
- Johnson, W.L., 1986. *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, Menlo Park, CA.
- Johnson, W.L., 1994. Agents that learn to explain themselves. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1257-1263. AAAI Press, Menlo Park, CA.
- Kotani, A. and Maes, P., 1994. Guide agents for virtual environment. *Proceedings of Lifelike Computer Characters '94*, Microsoft Research, Snowbird, UT, p. 59.
- Laird, J.E., Newell, A., and Rosenbloom, P.S., 1987. Soar: An architecture for general intelligence. *Artificial Intelligence* 33(1), pp. 1-64.

- Loftin, R.B., and Kenney, P., 1995. Training the Hubble space telescope flight team. *IEEE Computer Graphics and Applications* 15(5): 31-37.
- Mitchell, T.M., 1982. Generalization as search. *Artificial Intelligence* 18: 203-266.
- Munro, A. Authoring interactive graphical models. In T. de Jong, D. M. Towne, and H. Spada (Eds.), *The Use of Computer Models for Explication, Analysis and Experiential Learning*. Springer Verlag, 1994.
- Munro, A., Johnson, M.C., Surmon, D.S., and Wogulis, J.L., 1993. Attribute-centered simulation authoring for instruction, *Proceedings of the AI-ED 93 World Conference on Artificial Intelligence in Education*, pp. 82-89, AACE.
- Munro, A., Johnson, M.C., Pizzini, Q.A., Surmon, D.S., and Wogulis, J.L., A Tool for Building Simulation-Based Learning Environments, in *Simulation-Based Learning Technology Workshop Proceedings, ITS'96*, Montreal, Quebec, Canada, June 1996.
- Pizzini, Q.A., Munro, A., Wogulis, J.L., and Towne, D.M., The cost-effective authoring of procedural training, in *Architectures and Methods for Designing Cost-Effective and Reusable ITSs Workshop Proceedings, ITS'96*, Montreal, Quebec, Canada, June 1996.
- Regian, J.W, Shebilske, W., and Monk, J., 1992. A preliminary empirical evaluation of virtual reality as an instructional medium for visual-spatial tasks. *Journal of Communication* 42(4): 136-149.
- Rich, C., 1995. Diamond Park demonstration. *IJCAI Workshop on AI and Entertainment*, Montreal, Que.
- Rickel, J., 1988. An intelligent tutoring framework for task-oriented domains. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, Montreal, Canada.
- Russell, S., and Norvig, P., 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ.
- Sacerdoti, E., 1977. *A Structure for Plans and Behavior*. Elsevier North-Holland, New York.
- Self, J. 1995. The ebb and flow of student modeling, *Proceedings of the International Conference on Computers in Education (ICCE '95)*: 40-40h.
- Soloway, E., 1995. Beware, techies bearing gifts. *Communications of the ACM*, 38(1), pp. 17-24.
- Spensley, F. and Elsom-Cook, M. Generating domain representations for ITS. In D. Bierman, J. Breuker, and J. Sandberg (Eds.), *the proceedings of the fourth international conference on artificial intelligence and education*. Amsterdam: IOS, 1989, 276-280.
- Stansfield, S.A., 1994. A distributed virtual reality simulation system for situational training. *Presence* 3(4): 360-366.

- Stansfield, S.A., Miner, N., Shawver, D., and Rogers, D., 1995. An application of shared virtual reality to situational training. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pp. 156-161.
- Sterling, B., 1993. War is virtual hell. *Wired* 1(1), pp. 46-51.
- Stiles, R., McCarthy, L., and Pontecorvo, M., 1995. Training Studio interaction, *Proceedings of the 1995 Workshop on Simulation and Interaction in Virtual Environments (SIVE95)*. Iowa City, IA: ACM Press.
- Tambe, M., Johnson, W.L., and Shen, W.-M., 1997. Adaptive agent tracking - A preliminary report. *International Journal of Human-Computer Systems*, accepted for publication.
- Tambe, M., Johnson, W.L., Jones, R.M., Koss, F., Laird, J.E., Rosenbloom, P.S., and Schwamb, K., 1995. Intelligent agents for interactive simulation environments, *AI Magazine* (6)1, pp. 15-39.
- Tambe, M., 1997. Agent Architectures for Flexible, Practical Teamwork. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA.
- Towne, D. M. A generalized model of fault-isolation performance. In *Proceedings, Artificial Intelligence in Maintenance: Joint Services Workshop*, 1984.
- Towne, D. M. & Munro, A. The intelligent maintenance training system. In J. Psotka, L. D. Massey, and S. A. Mutter (Eds.), *Intelligent tutoring systems: Lessons learned* (479-530). Hillsdale, NJ: Erlbaum, 1988.
- Towne, D. M., Munro, A., Pizzini, Q. A., Surmon, D. S., Coller, L. D., & Wogulis, J. L. Model-building tools for simulation-based training. *Interactive Learning Environments*, 1991, **1**, 33-50.
- Towne, D. M. & Munro, A. Simulation-based instruction of technical skills. *Human Factors*, 1991, **33**, 325-341.
- Wang, X., 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. *Proceedings of the 12th International Conference on Machine Learning*.
- Webber, B. and Badler, N., 1993. Virtual interactive collaborators for simulation and training. *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, pp. 199-205, Institute for Simulation and Training, Orlando, FL.
- Weld, D.S., 1994. An introduction to least commitment planning. *AI Magazine*, 15(4):27-61.
- Wenger, E., 1987. *Intelligent Tutoring Systems*. Morgan Kaufmann, Menlo Park, CA.

Williams, M. D., Hollan, J. D., and Stevens, A. L. An overview of STEAMER: an advanced computer-assisted instruction system for propulsion engineering. *Behavior Research methods and Instrumentation*, 1981, 13, 85-90.