Abstract
As virtual reality systems become more commonplace the need for VR applications will increase. Agents are typically used to populate VR environments with autonomous creatures. Although systems exist incorporating agent and virtual environments, few support programming tools for specifying agent behavior. The paper presents the design of a system called HAVEN which uses a visual programming language to allow programmers to specify agent behavior from within a virtual environment. The system allows users to specify by example agent actions from low-level movement to higher level reactive rules and plans. Other details about the overall design of the system are also presented.

Introduction
As virtual reality systems become more commonplace, the demand for VR content and applications will rise. One of the more difficult tasks to design and implement in a virtual environment is dynamic behavior. Since the presence of humans in a virtual environment introduces asynchronous unpredictable behavior, inhabitants of a virtual world should react in an intelligent way to these events. A common solution to this is through the use of intelligent agents, since they can dynamically respond to changes in the environment. Their use in virtual environments has become increasingly common, not only for simple animal-like inhabitants of a world, but also as tutors and guides in learning and collaborative environments.

While agents have tremendous potential in VR, their incorporation into these systems can be made easier through the use of authoring tools. These tools can enable the builders of virtual worlds to more quickly incorporate agents into their environment. Ideally these agents should be generic enough to handle a wide variety of tasks in a virtual environment and should be simple to program. This paper details such a system. Called HAVEN (Hyper-programmed Agents for Virtual ENvironments) it combines a generic agent architecture and a visual programming language, allowing visual specification of behavior.

Previous Work
The earliest work in agents for virtual reality systems stems from the behavior animation work of the late 80's and early 90's. Behavioral animation arose from an interest in providing algorithmically driven behavior. The earliest work was that of Reynolds (Reynolds 1987) for modeling flocking behavior. Later work involves more large-scale systems such as the work of Blumberg (Blumberg and Galyean 1995) in the ALIVE system, Improv (Perlin and Goldberg 1995), and Oz (Bates 1992). These systems generally combine reactive agent architectures with computer graphics to produce autonomous virtual creatures. One of the more complex examples of an agent incorporated into VR is Steve (Rickel and Johnson 1998). Herman-the-Bug (Stone, Stelling, and Lester 1999) is a believable agent that acts as a tutor in an interactive learning environment.

While these systems provide a means of incorporating agents in a virtual environment, they induce even more complexity for virtual world designers, as programming solutions for these behavior systems are complex. Most systems provide an API bound to a high level programming language, or a scripting language for specifying behaviors, such as VRML 2 and Performer. These systems provide little in support for behavior specification. As a result, in order to construct behaviors, everything from low level graphical transformations to high level actions must be explicitly programmed. Steve provides alternatives to writing code, by providing a program-by-example system for the creating of plans. Steve provides alternatives to writing code, by providing a program-by-example system for the creating of plans. Most of the authoring support for Steve in the form of interfaces which are used to set parameters. The PBE system, while intriguing is limited to specifying one subset of behavior.

Tools for Programming Agents
HAVEN arose out of an interest to develop a generic agent with two criteria: 1) make it simple to incorporate into a
virtual environment, and 2) make it easy to design behaviors for the agent. Here at the Electronic Visualization Lab we have been spending most of this decade developing an immersive environment called the CAVE (Cruz-Neira et al. 1992). The CAVE provides tools to create virtual environments in the form of an API bound to C++. Many large-scale applications have been developed for the CAVE including a collaborative learning environment called NICE (Roussos et al. 1998).

A programming interface to an autonomous agent architecture which would allow for a range of behaviors from simple to complex to be easily designed and programmed is desirable. These tools could also be constructed to provide support for a range of programming from visual programming to a language bound API. Ideally, these programming tools should work from within a virtual environment.

3D visual languages provide a foundation for such tools. Visual programming languages have been applied to agents for specifying their behavior. This approach has been successfully developed in Agentsheets (Repenning 1995), KidSim (Cypher and Smith 1994), and Toontalk (Khan 1996).

A system, which incorporated a generic agent design with a tier of programming tools, would provide an ideal programming environment for creating intelligent agents. By providing for a simple means to specify agent behaviors, a programmer could create complex behaviors without having to build everything from scratch.

HAVEN’s Design

Since the focus of HAVEN is to develop tools for programming agents and not to develop a new agent architecture, it is advantageous to use an existing agent architecture. InterRap (Müller 96) is such an architecture. It is a vertically layered agent with each higher level handling increasingly complex tasks. This design allows for programming tools to be tailored for each level.

The agent architecture was implemented with additional changes including: converting the basic algorithms to a multi-threaded design and incorporating a distributed scene graph (a database of geometry and transformations stored as nodes in a tree), to handle agent appearance, and adopting it for use in virtual reality environments. Additionally a better motor control system was developed based on the work of Blumberg. The motor control system is vertically layered with the lowest layer being a Degree of Freedom or DOF, and the highest level being a controller for sequencing sets of DOFs.

The agent’s input system is composed of sensors giving the agent perception. These sensors are bound to nodes in the agent’s representational scene graph. This is required because some sensors, such as synthetic vision sensors need to consider the orientation of the sensor, when providing information.

Overall Design

If agents are to be used in a virtual environment, they need a support framework from which to operate. HAVEN is designed to be modular. This allows for the modules to interface with VR applications easily. These modules include: world, display, user, and agent modules.

The world module is the central management system for a virtual environment. This module is responsible for maintaining the world appearance, global state information, and registering agents and users as they enter and leave the environment. The display module is a base class which acts as a client of the world module. The display module has two variants: a user and agent versions. The basic responsibility of the display module is the same regardless of its type. Each display module contains the appearance (local scene graph) of a user (as an avatar) or an agent. When connected to the world module, any action which results in a change of state local scene graph is reflected back to the world module which then updates all of the other connected clients. As a result of this design the world module treats user and agents identically. This allows for a user to take control of an agent in the environment or more interestingly an agent can take over for a human (with limitations on its actions).

Agent Programming

The primary goal of HAVEN is to support the creation of behavior for autonomous agents. As the programmer’s expertise can range from a novice computer user to an expert programmer, there should be support for this range. The novice programmer is the primary user group for this system. While not accustom to textual programming, almost all computer users are experience with iconic based manipulation to justify a visual programming environment. Visual programming has demonstrated but arguably not proven that it is successful for end-user programming. Visual programming languages restricted to a specific domain however have been demonstrated to be very effective. The work demonstrated by Agentsheets and KidSim seem to indicate that graphical-rewrite rules and Programming-By-Example seem to work best for specifying behavior rules. This is the basic visual-programming model that is used for behavior programming.

The programming environment as designed is an immersive virtual environment for visually specifying agent behavior. Most of the programming could be done from within the environment. As an agent is composed of several layers, each handling more complex actions. The
programming support is built around these layers. Since each layer is responsible for controlling actions will be used by the next higher level, an agent’s behavior is developed in a bottom-up manner. This mirrors the agent’s flow of control as it responds to events. As a result, programming support is developed for three layers:

- Motor Skills
- Reactive Rules
- Plans

**Motor Skills**

Motor Skills are the lowest level actions an agent can perform. Motor skills typically involve some type of motion such as bipedal locomotion or grasping an object. Motor skills are composed of DOFs, which are bound to transformation node in an agent’s scene graph.

Motor skills are specified by example. A user can assign a DOF to transform node in the agent’s scene graph. An agent is assigned it’s own local coordinate system. This allows the user to specify what forward/backwards, left/right, up/down is relative to the agent.

Simple motor skills can be programmed by example. For instance, forward motion is specified by example. The user drags a representation of the root node forward and releases it. The distance and time it took to drag forward is then computed and used as the default forward speed of the Motor Skill called forward.

DOFs can either have a limited range of motion (like a head turn). A user can specify the range of motion by selecting the DOF and turning it (it assumes this is a rotational DOF) to its minimum range. Next the user turns the DOF to its maximum range and finally defines a default value if applicable.

It should be noted that motor skills are not limited to transformations of geometry. Motor skills can also be video textures, audio clips, and specialized functions.

**Reactive Rules**

The reactive layer is programmed much in the same manner as KidSim. The user specifies the enabling condition and demonstrates the action to perform. This layer uses the motor skills developed by the user in an earlier session to perform rule-based behaviors. A demonstrated action is a visual specification of a rule, which is of the form:

\[
\text{(START Condition)} \rightarrow \text{(RUNNING Condition)} \rightarrow \text{(END Condition)} \rightarrow \text{ACTION}
\]

For example, in Figure 1, the agent is given the enabling condition, an obstacle in its path. The user informs the agent that this is a new rule and that this is the enabling condition. Next, the agent is moved around the obstacle, (Figure 2), the agent turns until is can no longer detect an obstacle in its path. The agent is then moved so that is past the obstacle. Once clear of the obstacle the agent is informed that this definition has been completed, Figure 3. The resulting action is then translated into a rule.

The programming system is responsible for decomposing the rules action into motor skills. This means that it must have already been trained on how to turn and move forward. If an action defines a motor skill that is not present or unrecognizable then the programming system queries the user to define the appropriate motor skill.

Once a rule has been generated, the user can generalize it. For example, in the rule above the obstacle can be generalized so that any obstacle over a certain size will invoke this rule. The user can also alter the priority of this rule so that it takes precedent over other rules. Additionally objects can be grouped so that a rule applies to the entire class of objects.

Once behavior rules have been specified, sequences of rules and goal directed behavior could be built by using plans.

**Plans**

Plans are compositions of rules for accomplishing goals, and are programmed in a similar manner to the system for rules. Plans, however, are more complex in that they are usually are associated with a goal or a complex situation. There are two modes that can be used for plan development: plan specification and goal specification.

In plan specification, the user specifies the plan by example in a similar manner to reactive rules described above. The user defines the enabling condition and performs a set of actions. These actions however are more general and typically will be composed of reactive rules. The programming system will match actions performed by a user with rules stored in the reactive layer’s database. If it encounters a rule or action it cannot recognize it will ask the user to specify it. These plans have an enabling condition, when this condition is encountered by the agent the plan is enacted. Additionally more complex plans can be composed of other plans.

In goal specification, goals are specified by demonstrating the Goal State. The agent then computes a plan to achieve the goal by using the information and rules from its knowledge base. Again, if the system encounters a state that it cannot resolve it will query the user for the solution. Currently this is in development stage.
Scenario

An example demonstrating how a simple virtual creature can be built is given. A user wants to build a virtual creature. The user has already created its appearance and now is ready to add behavior to it. The first action the user does is to import the geometry into the system. Next, the user assigns DOFs to all transformation nodes that are to be used in motor skills. Finally, any sensors that are to be used by the agent are assigned.

The next step involves teaching the creature its basic motor skills. The creature is taught to move forward, turn, and jump. This is done in the manner described above, by assigned the root DOF and demonstrating the basic motions. After these motor skills have been specified, behaviors can be assigned. For example, the user wants his creature to be able to jump over obstacle that are not higher than 2 feet. To do this the agent is shown two rules. The first rule details how to jump over small objects, the second shows how to go around large objects. The user demonstrates both these rules and them generalize them so as to define the definition of *large* and *small*.

Next, the user wants the creature to place a certain type of object scattered about the world into a bin. The user demonstrates the steps needed to collect objects. The user defines the initial condition as the current state of the environment. Next, the user performs the plan (move to object, grasp object, move to bin, drop object in bin). Once all object are put into the bin, the user defines this as the goal state.

Other Considerations

The use of the visual programming module is not the only means of programming behavior. Plans generated with the visual programming module can be adjusted by modifying their textual representations. Experienced programmers would most likely appreciate a programming level API bound to a high-level language like C++. Both of these systems are available to support a range of programming support. This allows developers to move from a more abstract VPL to an explicit specification.

Future Work

There are many possible expansions on the work discussed here. One improvement would be developing better tools for specifying motor skills. A tool, which uses keyframe animation or inverse kinematics, might be very useful. Another area, which has not been addressed is the ability of agents to affect geometry of other objects in the environment. Since the behaviors are rule based, it should be possible to specify rules that would allow agents to construct, tear down, or alter shape. Systems for altering geometries do exist, including L-Systems and Shape Grammars, and could be incorporated into the environment. Finally user tests on the visual programming language could lead to developing better methods for visual behavior specification.

Conclusion

HAVEN allows for a generic agent to have its behavior programmed visually. These agents are designed to run in an immersive virtual environment. This system allows for complex creatures to be developed with less programming effort than required by other systems. By taking advantage of the layered structure of the agent, it provides a hierarchy of visual programming tools, which allows world designers to create a wide range of creatures. The capabilities of these creatures could range from simple reactive animals to virtual tutors capable of demonstrating complex tasks. It also provides a foundation for future extensions these ideas to provide even more capabilities for agents and their programming.

References


Muller, J. 1996 *The Design of Intelligent Agents*. New York.: Springer-Verlag


