Hyper-Programmed Agents for Virtual Environments

A Preliminary Proposal

Craig Barnes
University of Illinois at Chicago
June 1999
Table Of Contents

CHAPTER 1 - MOTIVATION.................................................................................................................. 5

CHAPTER 2 - BACKGROUND.............................................................................................................. 7

INTRODUCTION ................................................................................................................................. 7
AUTONOMOUS AGENTS .................................................................................................................... 8
  Definition ....................................................................................................................................... 9
VISUAL PROGRAMMING .................................................................................................................. 10
  History ......................................................................................................................................... 13
VIRTUAL REALITY AND VISUAL PROGRAMMING ............................................................................ 14
  History ......................................................................................................................................... 14
  Body Electric ................................................................................................................................. 15
  Lingua Graphica ........................................................................................................................... 15
  3D-Visualan .................................................................................................................................... 16
  Cube ............................................................................................................................................ 16
  Toontalk ....................................................................................................................................... 17
  Del Bimbo ...................................................................................................................................... 18
  SAM ............................................................................................................................................ 19
AGENTS AND VR .................................................................................................................................. 19
  Historical use of agents in virtual reality systems ........................................................................ 20
  Early Behavioral Animation ......................................................................................................... 20
  Reynolds’s work ............................................................................................................................ 21
  Tu and Terzopoulos ......................................................................................................................... 21
  ALIVE ......................................................................................................................................... 22
  Improv ......................................................................................................................................... 23
  OZ ............................................................................................................................................... 26
    Believable agents ........................................................................................................................ 26
    OZ architecture .......................................................................................................................... 26
  Cremer’s Work ............................................................................................................................... 27
  Wavish ......................................................................................................................................... 28
  Herman-the-Bug ............................................................................................................................ 28
    Agent Design .............................................................................................................................. 29
  Cosmo .......................................................................................................................................... 29
  AHA .............................................................................................................................................. 30
  Virtual Jack .................................................................................................................................... 32
  Steve ............................................................................................................................................ 33
VPL AND AGENTS .................................................................................................................................. 34
  Agentsheets ................................................................................................................................. 34
  Cocococa ....................................................................................................................................... 36
GAMES .................................................................................................................................................. 36
  Rule-based/FSM ............................................................................................................................. 37
  Artificial Life ................................................................................................................................. 37
  Genetic Algorithms ....................................................................................................................... 38
  Quake .......................................................................................................................................... 38
  Unreal .......................................................................................................................................... 38
  Motivate ....................................................................................................................................... 38
CONCLUSION ....................................................................................................................................... 39
List of Figures

Chapter 2

FIGURE 1: TIMELINE OF VR SYSTEMS ................................................................. 7
FIGURE 2: DATA-FLOW VPL EXAMPLE .............................................................. 12
FIGURE 3: FUNCTION REPRESENTED IN LINGUA GRAPHICA ....................... 15
FIGURE 4: 3D-VISUALAN .................................................................................. 16
FIGURE 5: A CUBE PROGRAM TO CONVERT TEMPERATURE ......................... 17
FIGURE 6: A TOONTALK PROGRAM ................................................................. 18
FIGURE 7: A SAM AGENT’S APPEARANCE ...................................................... 19
FIGURE 8: HERMAN-THE-BUG AND DESIGN-A-PLANT .................................. 28
FIGURE 9: COSMO ............................................................................................ 30
FIGURE 10: AHA ARCHITECTURE ..................................................................... 31
FIGURE 11: VIRTUAL JACK ............................................................................... 32
FIGURE 12: PROGRAMMABLE AGENTS FOR VIRTUAL REALITY A ROAD MAP .. 39

Chapter 3

FIGURE 13: ORIGINAL INTERRAP DESIGN ...................................................... 43
FIGURE 14 CONTROL LAYER ALGORITHM ..................................................... 45
FIGURE 15 HAVEN’S INTERRAP AGENT ......................................................... 47
FIGURE 16 WORLD MODULE DESIGN .......................................................... 49
FIGURE 17 BASIC DISPLAY MODULE ............................................................. 50
FIGURE 18 AGENT MODULE ......................................................................... 51
FIGURE 19 USER MODULE ............................................................................ 52
FIGURE 20 ENABLING CONDITION .................................................................. 55
FIGURE 21 RUNNING CONDITION .................................................................... 55
FIGURE 22 ENDING CONDITION .................................................................... 55
FIGURE 23: COMPLETE TIMELINE FOR DEVELOPMENT OF HAVEN PROTOTYPE 57
FIGURE 24 PHASE I - 21 WEEKS .................................................................... 59
FIGURE 25 PHASE II - 22 WEEKS ................................................................... 61
Chapter 1 - Motivation

Virtual reality is the science of illusion - a computer fabrication of a world. These worlds can range from familiar ones, such as the interior of an automobile, to places no one has ever actually been to like the surface of Mars, or it could be a world of fantasy. The key to virtual reality is presenting the illusion convincingly enough so that a user is pulled in as a participant of the world. This illusion must be believable in-order to be effective.

Virtual reality systems push the limits of real-time computer graphics in both hardware and software design. The hardware provides the system with the possibility of creating a believable world or environment. The software is split into two categories: system and user. Similar to an operating system the system level software is responsible for operating the underlying hardware, while the user level allows for the development of content.

Virtual reality, like it’s parent field computer graphics, anchors its foundations from an array of widely different areas of computer science: computer graphics, real-time systems, distributed computing, databases, and human-computer interaction. This amalgamation is required because of the unique requirements of virtual reality.

A virtual reality system is a medium, just as paint and canvas is for an artist. It can only be as good as its content. Because virtual reality system are complex and cumbersome, it can be hard for even experienced programmers to use them to create environments. This becomes problematic: if virtual reality systems can only be programmed by expert programmer how can these new tools achieve widespread use? Will these systems become like supercomputers? Costly and very expensive, existing only in national centers where user must either spend days learning how to program them or spend time working with an expert programmer trying to explain what is to be done

The centralized super-computer model started to break down in the 90’s. With the wide spread availability of high performance CPU’s like the Intel Pentium III and the Motorola G3, computer power which was only available as supercomputers in the 80’s is available today to the average consumer. High-speed graphics chips such as the 3Dfx Voodoo III provide high performance real-time rendered and textured computer graphics for a few hundred dollars. With all this power, virtual reality potentially can be as available as web-access is becoming.

With all these potential users available how can they go about creating virtual world without having to learn a programming language? Why bother to program at all. First no one system will provide everything for every user. Each user has specific requirements that are affected by the “domain” of the problem to solve. Programming provides the tools to allow users to solve their own specific tasks.

What is needed to allow user to create complex virtual environments? Tools which will allow users to create complex world with a minimal amount of programming, but not so general that they are not able to build what they want. One solution is to create a generic engine, which can be given a set of tasks to perform. These generic engines can perform task based on internal states and by perceiving its environment. Enabling these engines to perform tasks on behalf of users means that difficult details are of allocated to them freeing the user to developing what they set out to. These engines are typically referred to as agents.
One of the more difficult tasks to build into virtual environments is behavior. Behavior can be as simple as simple motions of objects to as complex as believable virtual humans with distinct personalities and emotions. A virtual environment in which the sole interaction is a user walking through an non-dyanamic environment quickly becomes boring. It is dynamic worlds, rich in interaction which capture the attention of users.

Such environment cannot only be entertaining, but also be used as a means of instruction. As we shall see the use of interactive 3D graphics mixed with autonomous agents have started to appear as tutoring systems. While these systems are mostly experimental, they do show the potential for virtual reality systems as a means for education, collaboration, and entertainment.

**Programming of Intelligent Agents**

In order to build complex virtual environments, tools, which would facilitate their construction, have and continue to be developed. However, these tools are typically in the form of computer languages or programming toolkits. Ideally a system should provide enough of a framework that would remove most of the technical details away from the user, the user would be mostly concerned with building the world. One such area of difficulty in programming virtual environment is behavior. A system that would give tools for constructing behavior would greatly aid users. Such a system would encompass the following areas:

- A generic programmable engine for behavior
- Tools for programming these engines
- An archive of extendable engine actions

Such a system could allow for creation of worlds that would not require low level programming. These tools could be used from both within and outside of the virtual world being constructed. These tools could allow even non-programmers to build behavior into virtual environments.

**Overview**

This thesis describes a tool which will allow a user to model complex behavior without having to learn a complex programming language. Named HAVEN (Hyper-programmed Agents for Virtual ENvironments) it will demonstrate the advantage of such a system for not only end-user programmers, but also experienced programmers.

This thesis is structured as follows:

Chapter 2 provides a survey of the parent disciplines: Virtual Reality, Behavioral Animation and Autonomous Agents, and Visual Programming. Systems which combine these disciplines as well as efforts which parallels this thesis are presented.

Chapter 3 provides a proposal for HAVEN. It defines the problem to be address by HAVEN, its various components, and a timeline for completion of a prototype.
Chapter 2 - Background

Introduction

As virtual reality systems become more commonplace, the demand for content will rise. Since virtual reality systems are dynamic and interactive, providing content becomes an increasingly harder task. One of the most difficult tasks to design and implement in a virtual environment is dynamic behavior. Because the presence of humans in a virtual environment introduces asynchronous unpredictable behavior, the virtual world should be able to react in an intelligent way to these events.

This chapter will describe the areas involved in providing complex behaviors in virtual environments. Its focus is not only in describing systems for complex behavior in dynamic virtual environments, but also in the area of providing high level programming (or hyper-programming) as a means to specify programs without a programming language.

The next few sections will provide a brief introduction to three disciplines: Virtual Reality, Agents, and Visual Programming. Following that is a survey of existing systems which combine two or more of the parent disciplines.

Virtual Reality

Virtual reality is computer generated immersive environment. It is the union of computer graphics, human computer interaction, 3D-tracker and display information, and a lot of engineering. Virtual reality systems allow a user to interact with a virtual world.

Figure 1: Timeline of VR systems
Virtual reality was first suggested by the work of Sutherland [91]. A brief history of VR is shown in Figure 1. Following the increased availability of raster displays and faster computers, virtual reality systems started develop. A large number of systems started to appear in the early 90’s. Today with the arise of powerful and inexpensive CPU’s like the Intel Pentium III and graphics board such as 3Dfx’s Voodoo III, systems capable of supporting some form of virtual reality are available to the public at a low cost.

Virtual worlds are strange beast to program. The development of scene graphs toolkits like Inventor and Performer has greatly aided in the development of real-time 3D graphics. However programming tools that are specialized for virtual reality programming are less common. The development of VRML 2[75] has promised a simple for a programming environment for Virtual reality. However, VRML 2 is more a file format than a true language, and programming support is supplied through an interface to Java. Alice [8] is also a 3D programming language. Built using the interpreted language Python, it allows users to build virtual environment by writing Alice code. Since Alice is interpreted, the world can be dynamically changed. Other toolkits such as World Toolkit and Dvise provide for the development of virtual world by supplying libraries of functions bound to a programming language like C++.

While both these systems provide for the programming virtual worlds, they still rely on existing methods of programming. It has been suggested that the programming of virtual worlds could be done within the environment. There have been systems that can be used to construct geometry from within the environment, and even a few systems for programming, as we shall see.

One difficult area to program virtual environment is behavior. Behavior can range from spinning objects to autonomous virtual humans. Most systems for programming behavior are still very much dependent on learning a programming language. However, there could be better methods for creating behavior, which could be both expressive and useable by a wide audience.

**Autonomous Agents**

The development of agents grew out of work in artificial intelligence in the 70’s. Although it was not an active area of research until the mid-80's. It has it basis in the early work in the planning systems of AI, and in distributed AI.

In these systems, an agent is termed a deliberative agent. This type of agent is one in which the world is represented in a symbolic state. The agent is given a desired goal state and a set of actions it can perform. It then performs essentially pattern matching to determine the sequence of actions that result in the goal state. An underlying theory of deliberative agents, which has born out over time, is that of BDI [89]: Beliefs, Desires, and Intentions. This formalism simply states that the agent has a set of belief about its environment, a set of goals, and a set of active goals (intentions).

However there are substantial problems with this approach. First, the size of the symbolic representation of the world state can be quite large resulting in increased time to compute a plan to achieve a goal. In the mid-80’s, Chapman produced evidence which indicates that even refined planning techniques would be unusable in any time-constrained system. The poor response time of these agents has led to the development of alternative designs.
Reactive agents arose out of frustration with deliberative agents. Unlike deliberative agents, reactive agents do not maintain a symbolic representation of the real world. Instead, they are based on perception and actions. The intelligence of reactive agents is defined by a hierarchy of behaviors. Lower behaviors have a higher priority over behaviors higher in the hierarchy. Behaviors compete with each other to control the agent’s actions, resulting in an emergence of a global behavior. These agents typically contain no symbolic information about their environment. Much of this work is based on the subsumption architecture model of Rodney Brooks [22],[14],[23], however other contributions further refined their design.

This reactive design has formed the basis for almost every behavioral animation system developed in the past decade. It also heavily influenced the development of the artificial life field [19].

Purely reactive agents are not without their disadvantages. One problem is that with a purely reactive agent overall behavior cannot be controlled. In addition, due to implementation architectures, such as Finite State Machines (FSMs), behaviors can be repetitive. (It should be noted that the Improv system described below, explicitly adds random noise to behaviors to increase the believability). Emergent behavior is just that, it developed as a result of the competition between behaviors in the agent. Additionally these behaviors can be very difficult to construct as a result of the sheer number of rules required to build complex behavior. In addition, the specification of higher level goals is difficult. As a means to address problems with both architectures, hybrid architectures, which combine both the reactive and deliberative approaches, have been developed.

Hybrid agents typically have both a reactive and deliberative layer. The reactive layer is responsible for handling events, which occur in the environment without having to invoke complex reasoning. The deliberative layer develops plans, goals, and makes decisions. Hybrid agents of this type tend to be layered in one manner or the other. Example of hybrid agents are: Touring Machines [30], Cosy [88], and InterRap [58]. Hybrid agents combine the advantages of both approaches while minimizing their flaws. Hybribs represent the current state of the art in agent design.

**Definition**

One of the biggest areas of debate is in the exact definition of an agent. This debate arises from the fact that there is a wide range of theories, which can be classified as an agent. Therefore, the term agent defies attempts to produce a universally accepted definition. Additionally, the term agent has been used perhaps too frequently to encompass any software entity that exhibits behavior that can be mistaken as intelligence. Wooldridge and Jennings [55] provide two different definitions for agents: weak and strong. The weak definition is a hardware or software system that at a minimum has the properties:

- Reactivity: refers to an agent perceiving and responding to changes in its environment.
- Autonomy: agents have and internal state and are able to perform operations without outside intervention.
- Social ability: agents are able to interact with other agents using some type of communication language.
The stronger definitions provided states that in addition to the above properties that the agent should exhibit some form of mental state. These include attributes as belief, emotion, intention, and desire (the foundation for the BDI agent architecture discussed below).

Other definitions include:

The AIMA Agent [56] "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors."

The Maes Agent [10] "Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed."

The Hayes-Roth Agent [54] Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions.

The KidSim Agent [43] "Let us define an agent as a persistent software entity dedicated to a specific purpose. 'Persistent' distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. 'Special purpose' distinguishes them from entire multifunction applications; agents are typically much smaller."

The multitude of definitions of agent results partially from the fact that agents are applicable to a broad range of problem domains.

Another perspective of agents [74] is that they are machines, which interact with their environment. Based on perceptions and internal states they perform tasks that cannot be fully described by an algorithm. These machines, called Interaction Machines, Wegner argues, are more powerful than Turing Machines since they can perform complex task by dynamically responding to their environment.

It should also be noted that not everyone is convinced that agents are a good idea. An interesting perspective on the evils of agents can be found in [48]

While not trying to infer any sense of intelligence or autonomous behavior by using the term agent, the sole purpose should be to act as a machine that can be programmed to perform a set of tasks. For the purposes of this work, the term agent will refer to any software machine, which is capable of performing tasks on behalf of a user.

**Visual Programming**

What is meant by Visual Programming? Visual programming is use of graphics, icons, and animations, in the process of programming. These graphical elements represent the expressions, statements, variables, and control flow of traditional programming languages. Visual programming languages replace these constructs with iconic representations, which can be manipulated by a user to produce a program. It is argued that the Visual programming languages, it is argued, simplify programming by allowing for programming to be performed at a higher level of abstraction. The advantages of this include: eliminating details of syntax, it’s closer to
mental models of the problem being solved. Visual programming has been used for programming in the area of concurrent systems (Visual Linda), Real-time control, and graphical user interfaces (Amulet, BX).

Visual Programming has several variants, which are commonly (and incorrectly), classified as visual programming. One such variant is program visualization. In program visualization the program is written in a textual programming language and the graphics are used to illustrate some aspects of the program.

Another variant that has become very popular is visual programming environments. These systems are not a true visual language but a very advanced editor and CASE Environment built onto of a textual language. These include Microsoft’s visual series: Visual C++, Visual J++, and Visual Basic. Although these systems are not in the Visual Programming Languages classification they are one of the most wide spread and widely used visual programming systems.

There are several flavors of Visual programming languages. These can be categorized as follows:

- Data-flow
- Control-flow
- Rule-based
- Programming-by-example
- Spreadsheets

The next section will give a more detailed description of these various categories of Visual Programming languages. Example systems that fit the classification well will be described along with the general pros and cons of these systems.
Data-flow

Data-flow visual programming systems work by allowing a programmer to visually manipulate data. Data-flow systems are usually a directed graph. Nodes in the graph are equivalent to functions, in a conventional programming language. These functions typically are of three types: data producers, filters, and consumers. Edges in the graph represent parameter passing between the various functions, which comprise the program. Most data-flow visual languages are stateless, however it is possible to start and stop the data flow that is equivalent to running and suspending a traditional program. Typed data can be enforced by binding a node port type to a particular type. This is a typical feature of most data-flow visual languages.

![Data-flow VPL example](image)

Data-flow visual languages are very successful especially in the area of visualization and image processing, primarily because of the data manipulation capability of these languages. This is because functions and/or data can be dynamically added to an executing program. It also allows for interactive control of the program. Successful data-flow based visual languages include: AVS[40], IRIS Explorer, and apE, which are used for scientific visualization.

While successful, data-flow languages have some problems most, of which result from common problems associated with visual programming. The most problematic are screen space and program readability. This arises when the number of nodes increases the visual representation of the program becomes a maze of nodes and data-flow connections. This rapidly consumes screen space and decreases a program’s readability.

Control-flow

One of the earliest visual program representations was the flowchart. A flowchart depicts the control flow of a program. If a program’s functions are represented visually (as blocks for instance), then order of function invocation is depicted as control flow. Typically, a flowchart was used as a documentation tool; however, they have themselves been used as a visual programming language. William Sutherland’s Graphical Program Editor [11] was not only the first visual language but it was control-flow based. Grail was another of the earliest control flow languages. Other include FPL (First Programming Language), IBGE and OPAL[90] which is used in a medical expert system and allows doctors to build programs which have concurrency, iterations, and conditional clauses.

Control-flow languages are quite good for novice programmer or for domain in which the sequencing of tasks is of primary importance. However, most control flow visual languages are not good at data manipulations. Control flow visual languages have fallen out of use and there are very few system which are currently based on this paradigm.

Rule-Based

Rule-based visual languages use iconic representations of the state of an object. The rules appear in the form of before and after states. A program is created by defining one or more rules. Many
visual programming languages have been rule-based including: ChemTrains [36], Agentsheets [9], and KidSim [43].

Rule-base systems have problems however. They do not scale very well because of the sheer number of rules required for larger programs. In addition, end users can have trouble interpreting the behavior of the rules or understanding the interrelationships between them. This arises because conflicting rules can be difficult to discover in system will large number of rules. Overall behavior can also be difficult to predict.

**Programming-by-example**

While Programming By Example (PBE) is not purely the domain of visual programming, a large percentage of such systems are graphically based. PBE systems are designed under the theory that a computer can be instructed to perform a task by watching what a user does and creating a program to perform those tasks. These systems range from simple macro recording systems to systems that can create complex LISP programs. The first PBE system, Pygmalion, was described by David Canfield Smith[]. It was designed to stimulate creative problem solving in people. Although it was designed for programmers, it was an attempt to make programming easier by allowing an algorithm to be specified at a higher level of abstraction than that of machine code. Its major contribution was that it introduced programming by demonstration and icons. An excellent survey of PBE systems can be found in [35]

PBE systems have one problem, which is hard to overcome, and is handled differently by different PBE systems. This problem is that it can be difficult for the system to determine what exactly the user did to perform a task and how to know when to perform that task later. In other words, it can be difficult for the PBE system to infer the user’s intent. Solutions to this problem have been developed, with one of the most interesting being that of Cocooa described below.

**Spreadsheets**

Perhaps the most commonly used visual languages are a spreadsheet. While not explicitly design for general-purpose programming, they do incorporate features than can be used for programming. Some of the features of spreadsheets [X Amberler87] are that they are modeless, they lack variables, and the basic form maps directly to the model of the data. AgentSheets is based on a spreadsheet to represent spatial relationships between agents

**History**

Visual programming languages have been proposed as an alternative to traditional textual based languages. Born at the same time as computer graphics, they have not and very likely will not replace textual programming languages but rather have been used very successfully for domain specific programming.

Visual programming languages appeared shortly after the first computer graphics system of Ivan Sutherland. William Sutherland [11] created the first visual programming language, which allowed users to select functions from a menu, connect them using a light pen, assign input values, and watch it execute in real time.

The increased availability of bitmapped displays in the early 80’s spurred the development of visual programming languages. These early languages spanned a wide array of designs and
domains. One initial goal of visual languages was to develop a general-purpose language with
would replace textual ones. However, it has since been discovered that general-purpose languages
are not completely successful. However, in specific domains they have become successful, as in
the case of scientific visualization programs.

There has yet to be development of any real formalism for visual languages. Visual languages are
extremely varied in semantics and appearance. While there has been indications that visual
programming can be a better way to program it has yet to be proven [92]. Visual programming
languages continued to develop in the late 70’s and early 80’s Some notable systems include
LabView, Khoros[39], and AVS[40].

One early problem with visual programming languages was the goal to completely replace textual
programming languages with a general-purpose visual programming language. This goal was
considered achievable, however, it eventually proved to be nearly impossible to accomplish. This
resulted in forcing visual language designers to reconsider uses for VPL. Today Visual
Languages are used to provide task-specific programming for end users. Such areas include
virtual reality, agent programming, and programming environment for children.

**Virtual Reality and Visual Programming**

Visual programming systems are starting to appear in virtual reality systems. Although an
extensive use of VPL’s in virtual reality has yet to appear there have nonetheless been efforts to
incorporate them. The next section describes what work has been done in incorporating visual
programming with virtual reality.

**History**

Visual programming was born with the work of Sutherland [11]. At the time raster displays were
not common and 3D interactive graphics was still in its infancy. This led to most visual
programming languages being screen spaced based and therefore two dimensional. This is still
the case today as almost all visual programming systems are still 2D. The use of 3D visual
languages is still very much unexplored territory.

The first inkling of attempting to incorporate a visual programming languages in a 3D
environment was describe in Gilnert’s “Out of Flatland” paper [17]. Gilnert asserts that it is
unnatural to program in anything less that three dimension because this is not how we perceive
the world around us. He proposes that it is more natural to program in three dimensions, and that
by extending visual programming into a third dimension that it will greatly expand the
capabilities of visual programming. Three dimensional programming languages could also solve
some of the persistent problems with 2D VPL including scalability and screen real estate. Gilnert
further goes on to describe a 3D extension of his 2D VPL Blox. He proposes extending the syntax
to include height and color to the basic operatives of Blox.

Although Gilnert proposes that three-dimensional visual programming might be better that 2D,
few three dimensional programming languages exists. The ones that have been created since
Gilnert’s papers are describe below.
**Body Electric**

The earliest visual language for virtual reality was VPL’s Body Electric [26]. The creators of this language recognized the importance of end user programming for creating virtual worlds. They stressed the need for a virtual studio, which would provide a basis for designing both object and behaviors in the environment. This realization had not caught on as one might expect. To this day, tools for creating a virtual environment from within VR are rare. A few such systems are described below, but none have the scope that Body Electric hoped to provide.

Body Electric is a visual language, which uses the data-flow model to program a virtual environment. By allowing the user to assign inputs from a variety of head and hand trackers to their virtual equivalents. The user connects boxes to lines (as is typical in data-flow based VPLs) to represent data transformation. The language is hierarchical in that nodes in the language can themselves be data-flow networks. These networks can be also be complied to increase the performance of the code. While designed for a virtual environment, the actually language representation is still a traditional 2D data-flow layout. The operations of the language still seem to be low level and it is not apparent how more abstract representations can be accomplished with Body Electric.

**Lingua Graphica**

Lingua Graphica [27] is one of the first 3D-based visual programming languages. Designed for a virtual environment it provides a graphical representation of a running program written in a base language, in this case C++. Programs are loaded into the environment and a 3D representation is displayed. Language elements are mapped to a 3D representation. Type and objects are represented by a shape that appears atop a transparent cube. Class inheritance, data flow, and function calls are represented by arcs called associative links. Once a program is loaded and its representation mapped, it can be manipulated and altered by a user.

```
double example(double num, char *str)
{
    printf("%s", (num*num));
    return(num);
}
```

**Figure 3: Function represented in Lingua Graphica**
For example in Figure 3 a small C++ function appear as it is rendered in Lingua Graphica. Although called by its designers as one of the first 3D visual languages it is really an interactive language for program visualization and algorithm animation. It can also be classified as a visual programming environment for C++. A similar language by the same authors is CAEL-3D which is based on Pascal [93].

3D-Visualan

3D-Visualan [61] is another 3D based visual language. It differs from the above system because as it is a rule-based visual language. The primary language elements are rules, which are represented by before and after pictures. A program is executed by matching the before picture of the highest priority rule with the current state of the world. Once a rule has been “executed” the new state of the object or world is that of the after picture. The program runs as long as there are matched rules to enable. When there is no possible rule to enable the program stops. A rule’s priority is determined by its position in 3-space. Rules farther away from the center of the “world” (program) have priority over closer rules. Multiple rules can be executed at the same time.

A more complex rule can be specified by composting two or more before states together. In this case all before states much match before a rule can be executed. 3D-Visualan was used to create several simple programs including a Space Invaders type arcade game.

Cube

Cube [60] is an example of a 3D visual language that uses data-flow paradigm for programming. Based on first-order logic, cube represents language elements as cubes, which are connected together by tubes. Cube’s programs are similar to those written in Prolog. Language elements, such as variables and predicates, are represented by cubes. These cubes are linked together by tubes which represent dataflow between the various language elements. Dataflow is bi-directional. Unbounded variables are bound to a variable if they are connected by a tube.

Figure 4: 3D-Visualan
regardless of how they are connected. Language predicates are represented a green cube with a unique icon decals on its top which represents it’s label. Predicate cubes also have ports, represented by holes on the cube's sides, where dataflow tubes can be attached. These ports are equivalent to parameters in textual programming languages.

Figure 5: A Cube program to convert temperature

Inside of a predicate cube are transparent boxes called planes, which are stacked vertically. Each plane is equivalent to a clause in a prolog program. A label on top of the cube, which displays its value, identifies value cubes. Variables are represented as transparent cubes called a holder cube. This cube can hold either a value cube or a predicate cube, or it can remain empty representing an unbound variable.

A program is constructed by visually placing cubed and connecting them. Once build a program is run by allowing data flow between connected cubes. Cube, like, Prolog can produce multiple solutions to a query. Cube’s usefulness is somewhat limited to the domain of first-order logic programming, however it was one of the earliest system to incorporate a 3D visual syntax as a crucial element of its design.

Toontalk

Toontalk[63] is a 3D visual programming system explicitly design for teaching children to program. It assigns a metaphor to a set of icons. The metaphor selected was that of a city. A basic building block of the programs is agents, which are represented by houses. Houses contain robots, which are trained via programming by demonstration to perform tasks. At any time the current state of a robot can be queried or modified by placing an icon of a thought bubble over it. Animated icons of birds perform communication between agents (houses). Birds deliver messages by delivering pieces of paper containing messages from one house to a next. Each house has one or more nests which represent communications ports to the agents. Messages are placed on the bottom of the nest when delivered and removed from the top of the pile of messages when read. Programs can be saved by storing the city in a notebook icon. The system is concurrent since each agent can perform action independently.
The basic design goal was to find a metaphor that children could understand so they could write programs. Using the city metaphor and visual programming, children can program complex tasks with having to learn a difficult programming language. Another benefit the author felt was that concurrent systems are conceptually easier to program than sequential ones, because the real world is concurrent, not sequential.

Del Bimbo

Del Bimbo et al.[42] developed a visual language for the programming of agent behavior in virtual environments. This system provides a visual language using Programming by Example to program reactive behavior of agents. 3D relationships, which form the basis of agent behavior, are described in a language called eXtended Spatio Temporal Logic (XSTL). XSTL allows for the expression of temporal and spatial relationships. Relationships can be expressed quantitively (move to 3 ft along the x-axis, 15 minutes) as well as qualitatively (near, far, sooner, later). Spatial relationships in XSTL are described by allowing for assertions describing the spatial location of an object in relation to a point in 3-space. These spatial assertions are combined by logical connectors and backwards temporal logical operators (cite) to form sequence assertions which defines the evolution over time of spatial assertions. Because specifying reactive behaviors in XSTL is difficult, the authors developed a system for supporting visual specification by example. The domain of this system is a driving simulator, in which there are automobiles and intersections. Cars are agents and have behaviors assigned to them such as stop at a red light, maintain appropriate distance behind a moving vehicle, and so on. By using an iconic 3D interface, a training scene is composed which defines the conditions in which the agents are to react. Each agent is "trained" separately once the scene is composed. In training mode, the programmer assumes the appearance and perspective of the trainee agent, in this case and automobile. For each situation, which results in an action by the agent, the programmer demonstrates both the enabling condition and the response. For example, an agent is trained to stop at a crosswalk that is not empty by approaching it and stepping on the brake to stop the car at the non-empty intersection. Each agent’s behavior can be programmed in this way.
SAM

SAM (Solid agent in motion) [85] is a 3D visual programming language for specifying message passing to simple agents. Like Lingua Graphica, it integrates 3D-program visualization with a programming environment. A SAM program is composed of a set of 3D objects representing agents which communicate by exchanging messages. Each agent is composed of a set of ports. Input ports are represented by an input cone \( \rightarrow \), output ports are represented by output cones \( \rightarrow \). It follows the design of Toontalk and Pictorial Janus. A message icon consists of a rectangular box containing a data value. Rules describing the behavior of agents are located inside the agent’s icon. A rule consists of an enabling condition and a set of actions. Actions are represented by different colored sections of a tube.

![Figure 7: A SAM agent's appearance](image)

SAM’s agents exchange messages synchronously while each agent is executed in parallel. A message is one of two types: direct (agent to agent) or broadcast (agent to all other agents). As each message is received, the enabling conditions of an agent’s ruleset are checked. A rule is non-deterministically chosen from the set of rules for which its enabling condition is true. Rules are members of a set of actions which include, sending a message, duplicating an agent, transforming an agent’s appearance, or playing a sound. Programs are constructed by combining two or more agents. An action is enabled by having one agent send a message to another. Upon receiving the message an agent performs a task depending on the message content. The overall effect is similar to a data-flow network in which the agents are filters and the messages represent data. The agent’s rules themselves are written in a scripting language.

Agents and VR

Since agents are useful for modeling behavior in dynamic systems, the application of agents to virtual reality systems is a logic step. Since the interest in agents arose about the same time as virtual reality system started to appear, agents have been used in virtual reality from the beginning, especially if one includes the research into artificial life.

Why use agents in virtual reality systems? Simply put agents are designed to function in dynamic environments where events occur asynchronously. Once a human user enters the environment, the system can become very unpredictable due to the events generated by the user. Since one trait of agent systems is that they are reactive, they perform well in this type of environment.
Intelligent agents are also used to “off-load” some of the work a programmer must code in-order to describe complex behavior. This has led to a post-object oriented paradigm, coined Agent Oriented Programming (AOP)[28],[34]. AOP is a programming paradigm in which agents are assumed to have mental states. These states along with messages an agent receives control its actions. While AOP has yet to become a popular programming practice it does provide and interesting alternative to object oriented programming.

Historical use of agents in virtual reality systems.

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 provided algorithmically driven behavior. About the same time, the interest in artificial life became another factor in behavior animation. While artificial life, research was primarily interested in what Brooks [22] termed “emergent behavior”, other people working in behavioral animation tried to design systems for incorporating higher level control of animation.

Agents use in virtual reality systems arose out of behavioral animation research for interactive real-time graphics. Behavioral animation specifies general rules for animations as opposed to keyframe animation which specifies certain keyframes (or points) for which the tweens or intervening frames are created by applying inverse kinematics to interpolate between the key frames.

Early Behavioral Animation

Autonomous agents or virtual creatures developed from two difference approaches – behavioral animation and artificial intelligence. Behavioral animation takes the approach of Brooks and relies on reactions to environmental stimulus, while the AI approach tries to develop a complete mind. Behavioral animation approaches are bottom-up: low-level rules are specified and the resulting behavior emerges from interactions with the environment. AI used the sense-plan-act model by developing a sequence of actions to perform tasks.

In the late-80’s Rodney Brooks criticized the classic AI approach of agents as focusing too much on trying to represent the complete state of the world. He advocated using the world as its own model and suggested a system that acted based on its perceptions. He argued that once agents are able to react to their environment other problems such as reasoning and problem solving are easy. He proposed an architecture in which lower levels implement basic behavior while higher levels are used for higher level abilities such as pursuing goals. Control over the overall behavior is dependent on two mechanisms: inhibition and suppression, which means high level actions inhibit low level actions to pursue their own interest. The ability of agents to meet both low and high level goals is dependent on their interaction. The work of Brooks was extending by Maes, Kaelbling, and Chapman and Agre who developed agent based reactive architectures.

This work in reactive agents gave rise to agents (or interaction machines) in which the complexity of the behavior of the agent is directly related by the complexity of its environment. These classes of agents are based entirely on sensory input. There is no high complex processing or modeling of the state of the world.
Brook’s work becomes the basis for most behavioral animation and agent based systems incorporated into virtual reality systems in the decade to come. To this day, a vast majority of the agent based systems described in this thesis use the fundamental design of Brooks.

Reynolds’s work

Early behavioral animation work includes the work of Reynolds [15] for simulating flocks, herds, and schools. While not explicitly referencing the work of Brooks, it does use the basic principles of reactive behavior and is often used as the starting reference point for behavioral animation. It is also interesting to note that this work is also considered one of the earliest artificial life systems.

Reynolds simulates the motions of flocks of birds by modifying a particle system so as each particle, called a boid, is actually an actor with it’s own perception and behavior. Simulating flocking behavior then becomes simply a matter of creating a set of boids and allows them to interact. The overall flocking behavior emerges from the individual actions of the boids. The main difference between Reynold’s system and a traditional particle system is that although individual particles have their own attributes they are typically updated by a global mechanism.

Each boid has it's own local coordinate systems which represents its position and orientation. Motion is enabled through the use of a geometric flight model. This model allows for forward motion, turning and banking, as well as climbing and diving in the case of flying boids.

Boids also have a set of behaviors, which allow them to be incorporated into a flock. These behaviors coordinate the behavior of a boid with that of its flockmates. These behaviors, listed in decreasing order of importance, are:

- Collision Avoidance
- Velocity Matching
- Flock Centering

These three rules not only control each boids local behavior but also the behavior of the flock as a whole. Each rule has an associated weight between 0 and 1 which can attenuated a rule's chances of being selected. The navigation module evaluates the set of rules for each boid, which determine which of the rules it should follow. This evaluation results in an acceleration and direction. This is passed to the flight model, which attempts to move in the direction calculated.

Each boid also has simulated perception of what is immediately in front of it. A boid can see in its immediate neighborhood, which is defined as a spherical zone centered at the boid's local center. Sensitivity is defined as an inverse exponential of distance from the center. The parameters needed for the simulated perception therefore are radius and an exponent.

Global flocking behavior can also be influenced by defining a migratory urge. This can be simply a point in three space for which the flock is attracted. This point can be move in real time to get the flock to move after the target.

Tu and Terzopoulos

One of the earliest works in behavioral animation and autonomous agents is the artificial fish of Tu [94]. The main goal of their work was to make the fish self-animation, similar to the work of Reynolds but with a more complex model. Tu’s fish can sense the world through the use of
synthetic vision, and combine simple behaviors to more higher level motivational behaviors which dependant on the fish’s mental state. The fish move through the use of a motor control system, which simulates hydrodynamic motion. The motor system also affects the fins of fish to achieve motions such as forward movement, roll/pitch, direction, and acceleration. The combining of the sensors, motor system and mental state make the fish an autonomous agents. In this case the fish are reactive agents as defined by Brooks. By applying a small set of simple rules, the fish exhibit emergent behavior.

The behavior of the fish is controlled by three functions, which describe a fish’s mental state: Hunger, libido, and fear. These functions have a range of 0 to 1, and form part of an intensity generator. This generator is a function that evaluates the three functions and determines which action is needed. Once an action has been activated the appropriate motor control is activated.

**ALIVE**

The ALIVE [33],[53] project’s primary goal was to incorporate autonomous agent in a virtual environment. Blumberg [64] combined behavior animation which an autonomous agent to build a virtual creature, in this case a dog. The ALIVE project extended the earlier work in autonomous agents by providing a generic framework for behavioral modeling of virtual creatures, and by providing a layered approach to behavioral specification. A main design goal was to allow for an autonomous creature, which could also interact with user in its environment.

The layered approach is an attempt to separate low level commands required to perform actions from the higher level behavior. For example, move-toward is a high level command which is independent from the motor skills which performs the action (walk, hop).

The motor system is composed of three layers: a controller, a motor skill, and the degree of freedom (DOF). The motor system design provides several advantages over previous systems. These include: a generic set of commands all creatures can perform, it acts as a abstraction level between higher level commands and creature specific implementation, and it provides resource management which enables support of concurrent motion. This motor control system is one of the major contributions of this work as it has been incorporated into a large number of similar systems.

The lowest level of the motor skills is the DOF layer, which provides resource management and provides a mapping from simple input values (range [0,1]) to more complex motions. Resource management is accomplished by locking and unlocking DOFs so as to allocate them to a Motor Skill.

Motor Skills are the second layer in the motor system. Motor skills allocate one or more DOFs to perform movement. A Motor Skill becomes active after it has allocated all of the required DOFs. Once active it adjusts all of the DOFs with each step of a global system clock. Once activated, a Motor Skill runs until completed or interrupted by the Control Layer.

The control Layer is the highest level of the Motor System. It provides a mapping of commands from the Behavior system to the Motor Skills. Commands such as move-to, turn, and halt are mapped into appropriate motor skills.

The ALIVE system also used sensors gather information about the environment. The sensors are of three types: Real World, Direct, or Synthetic Vision. Real World sensors are physical sensors
attached as input devices, while Direct sensors query other objects in the world. Synthetic Vision uses image techniques to simulate a creature’s vision. These sensors are useful for low level navigation and collision detection.

The Behavior system is the highest layer of the Alive architecture. It is a loose hierarchy of goals. These goals make be specific (chase ball) to general (reduce hunger). These behaviors compete for control of the creature. A winning behavior is selected based on an overall value that is updated in each cycle of the behavioral system. Behaviors influence the system by inhibiting other behaviors, changing the values of internal variables, and alter the function of motor commands.

Behaviors rely on objects called Releasing Mechanisms to filter sensory data and events that concern the Behavior. The object outputs a continuous value that depends not only on the stimulus received but also on other Behaviors. Behaviors are organized into groups of mutually inhibiting behaviors called Behavior Groups. The groups can be used to localize interaction among behaviors.

Creatures also have Internal Variables, which model the state of a creature. These variables can be constant or vary over time as the result of Behavior activation, or some autonomous growth and decay rate.

Behavior activation is controlled by functions whose overall values are computed from a variety of other factors. These include: Internal values, Releasing Mechanisms, sensory inputs, inhibiting values from high level behaviors, and level of interest. Level of interest is used to increase the likelihood that a particular behavior is activated by increasing at some determined rate during a behavior update cycle. This is also used to decrease the chance that a currently active behavior will continue to be active.

Global behavior is constructed by setting the value of Internal Variables and defined functions that affect level of interest values. A creature in Alive is defined by a C++ library of functions. The user is responsible for defining all aspects of the creature from DOFs to behavior groups. A highly complex creature therefore requires quite a programming effort. For example, a dog called Silas was implemented with 3000 lines of C++ code, 2000 of which were used for implementing its Motor Skills.

**Improv**

Improv [67] is an authoring system for constructing behaviors of animated actors. While it provides similar capabilities as the work of Blumberg it, major difference is that it is designed to be an authoring tool for end user programming of behavior.

Improv's design is similar in nature to Blumberg's ALIVE system in that it uses a geometry model that can be manipulated in real-time. It is comprised of two layers: An Animation layer which is responsible for low level animated action such as walk or grasp, and a Behavior layer which is responsible for high-level capabilities such as retrieve blue ball, and trigger actions of the Animation layer.

The geometry layer is very similar to the one in the ALIVE system. Actions are specified by defining degrees of freedom or DOF's. There are various types of DOF the simplest being a
rotation axis between any two connected parts. Additionally Improv allows for mesh deformations that can be used in muscle flexing and facial expressions.

Actions are composed of one or more DOF's. Each DOF represents a range in degrees that an axis can move through. DOF are composed of three angular ranges representing roll, pitch and yaw and three interpolents which can vary over time to enable movement through the allowed range. A user specifies a DOF in the following manner:

define ACTION "Gesture1"
{  
    R_UPPER_ARM  25:55  0 -35:65 { N0 0 N0 }
}

The first element is the part name followed by three angular ranges for each axis. The last 3 tuple is the interpolent values. For example, if N0 has a value of 0.5 then the current angle for axis 1 would be 40 degrees. The interpolent has an additional property, which represents time varying coherent noise of differing frequencies. The noise is introduced to give the motions a more "natural" look by reducing the "cleanness" of the motion.

Actions themselves can be composited together to allow for simultaneous action. This is done by allowing for non-interfering actions to be composed together much in the same way as images are. If two actions are non-conflicting (they do not access the same DOF) then the actions can be combined by decreasing the activity of one action gradually while increasing the activity of the other. If a set of actions do conflict then lower priority actions are gradually reduced while the higher priority action activity is gradually increased resulting in a smooth transition between actions.

This is accomplished by the authoring system by placing competing actions in the same group. At any one moment all the actions in the group have some weight. When an action is selected it weight gradually moves to 1 from zero while all other actions are ramped down to zero. Groups are prioritized by placement in a list with groups farther down the list at a higher priority that action group near the top of the list.

A model action per cycle is calculated by the following algorithm: for each DOF in the model a weighted sum is computed for all the contributions of each action to the DOF. Additionally transitions between two actions can be buffered via a finite state automata to prevent parts of an actor from passing through each other.

The Behavior engine is the highest level of Improv. This level is responsible for selecting one action over the other either non-deterministically or based on scripted behavior. The behavior engine allows for authoring of an actor behavior using scripts. The most basic script is simply a scheme in which multiple scripts are run in parallel. Scripts can be layered so that more abstract or long term actions are layered on top of more reactive and localized term actions. This layering is similar to the behavior engine of ALIVE. The layering of scripts is, again, text based with higher level scripts being declared first and shorter term and reactive script declared last. Scripts are organized as a sequence of clauses, which either call other scripts and/or actions, or query or modify an actor's properties.

Non-deterministic behavior allows for an actor to randomly choose an action from among a set of actions. These choices can be weighted so that some choices are more desirable than others. For
example: a set of three choices could be weighted \{ X 0.6 Y 0.4 Z 0.1 \} so that X has a 50% chance of being chosen while Y and Z have a 40%, and 10% chance respectively.

In addition to weighted choices there are also scripts, which affect the internal state of an actor. Called decision rules, they are used to describe an actor's personality, mood and relationship to other objects. As decision rules change, the actor’s behavior can change as well.

Actors in Improv can also store internal properties which store state or attribute information. These properties may be used to describe an actor's personality, mood, and/or attitude about other objects in the environment. These properties can be set in scripts as a result of an action or at the actor's creation time. In the case of a scripting change the script looks like:

define SCRIPT "Eat Dinner"
    { "Eat" }
    { set my "Appetite" to 0 }
    { "Sleep" }

A class of scripts which affect and use these properties are also provided in Improv. Called decision rules, these rules are used to describe an actor's personality, mood and relationship to other objects. As decision rules change, the actor's behavioral can change as well. Each decision rule consists of a list of factors, which are used to influence an actor's decision. Each factor has an associated weight, which the author can use to control how much influence it has on the decision. Decision rules typically of the form:

define DECISION-RULE "whose interesting"
factor { his/her "Charisma" } influence 0.8
factor { his/her "Intelligence" } influence 0.2

In this case the decision rule "who's interesting" will cause an actor to find other actor's Charisma more interesting than its intelligence. Fuzzy rules can be also specified by allowing for ranges of weight associated with factors. Using this an actor behavior will tend to favor on action over another based on how the fuzzy rule evaluates.

Additionally decision rules can be defined so that an action can be chosen from a set of actions by using different factors to determine which action to choose from. In this case a list of objects passed to the rule can be divided into subsets. The weight assigned to a subset may be scaled, which would represent a preference for one group of choices over another.

Multiple actors can be incorporated into Improv. An author can coordinate the actions of multiple actors. Actors are allowed to modify any property of any other actor, which allows actors to influence each other. Communication between actors is done using a shared blackboard.

Improv allows authors to control actors from several levels, from low-level atomic actions to directing groups of actors. The scripting interface is a textual based language. There is no support for any other mechanism for defining behavior. All creatures must be built from the ground up.

Although Improv is designed with a layered approach of behavioral specification it does not have a strong basis in the BDI architecture of AI based intelligent agents. Instead it follows Brook’s subsumption design as a result, the layered approach is arbitrary, with plans and coordination
being specified by the user with little enforcement of the hierarchy. Like ALIVE this can led to
difficulty in defining exactly what behavior will result.

OZ

Oz [4],[5],[21],[24],[32] is an agent-based authoring system for creating and presenting
interactive dramas. OZ combines believable characters, presentation, and drama to create a virtual
environment rich in context. Designed as both and authoring and presentation system. OZ is built
with three sub-systems, which allow for believable characters, appropriate presentation, and
dramatic direction. A world in Oz is composed of a simulated environment, agents who inhabit it,
a user interface and authoring tool, and a drama manager that plans and guides the overall flow of
events.

Believable characters are provided by autonomous agents. The creators of OZ felt that a reactive
agent architecture could not provide all of the abilities they felt were necessary for creating
believable characters. The presentation system is either text-based or 3D interactive graphics;
therefore interaction with the environment is either text-based or graphically based. A drama
system allows for an interactive story (with some restrictions), by allowing the specification of
long-term goals among a group of agents to form a narrative story.

Believable agents

Believable agents are a class of agents in which their sole purpose is to act in a believable manner
so as to allow for the suspension of disbelief. In other words, a user believes the agent is the
creature it is supposed to represent. These agents are constructed to deal with imprecise
perceptions and world knowledge. They must also be able to react quickly. The term broad agent
is used to define an agent who has broad and shallow capabilities. This means that the agent does
not necessarily have to be extremely active and smart, but just have just enough capability to
produce behavior that is believable. The term believable means that the agent has the following
properties:

- Goals and goal directed behavior
- Some natural language ability
- Some inference capabilities
- Exhibit emotions

In [21] and [5], Loyall and Bates argue the difference between broad agents and traditional AI
based agents is that while traditional artificial intelligence tries to build complete minds, broad
agents tries to build complete agents with both some intelligent and physical representation. They
go on to further make the distinction that behavioral agent’s success is measured in terms of
believability while AI agent’s is measured in terms of complex reasoning and problem solving.

OZ architecture

The agent architecture in OZ, called Tok, is comprised of several different parts: a reactive
system called Hap, and emotion component called Em, a natural language generation system, and
text and graphics based interfaces.
Hap supports goal-directed behaviors. Actions are selected by weighing several factors including perception, current goals, emotional state, and internal state. Hap functions by choosing an action from an action library. Actions are composed of either ordered or unordered collections of sub-goals, motor control actions, and internal state actions. Several actions can be active at one time as long as they do not require the same resources.

High-level actions are structured as a hierarchy of sub-actions. If at any particular time an action fails another is selected. The failing action is not selected again. Goals and sub-goals are stored in two places. All of the actions an agent knows are stored in the action library. All goals currently being pursued are stored in a structure called a plan tree. This tree is contently updated as part of a processing loop. At the top of the loop the current state of the environment is read. Active rules are evaluated to see if they are still valid. Just completed plans are removed from the plan tree as well. Next the activating state of all rules in the plan library are evaluated to see if any can be activated, if they can be they are placed in the plan tree. Finally, a goal is chosen to be executed. Once it is executed, it runs until it is completed.

The other interesting system in OZ is the emotion system called Em. Em provides creatures built with OZ to exhibit some emotional behavior.

Although OZ supports reactive architectures, it was explicitly designed to also be able to pursue goals and plans. The authors felt that the addition of deliberative aspect would result in a system that can create more characters that are believable.

Although OZ was incorporated into a virtual reality system, it most developed version is a text based MUD. Additionally there are few authoring tools provided for developing these agents.

Cremer’s Work

Cremer et al.[47], provide a system for behavior and scenario control of agents, by using a variation of a Finite State Machine. The behavior system is part of the Iowa Driving Simulator, which is a ground vehicle simulator that incorporates motion, force feedback, and high performance 3D graphics and audio.

The agents in the system are used to model autonomous vehicles in driving simulations. Agent behavior is specified by Hierarchical Concurrent State Machines, or HCSMs. Based on CCM [45], HCSM are essentially Finite State Machines in which each node of the FSM can itself be a FSM. This allows a state machine to contain multiple concurrently executing child FSMs. By using this design, complex behavior is easier to construct and debug.

Agent behavior is encoded into a HCSM. Since HCSMs can be organized hierarchically this allows similar goals to be grouped together. These agents are reactive and can exhibit some level of autonomous behavior. More complex scenarios can be managed through the use of a special class of objects called Director Objects. Director object manage overall goals for a particular scenario, by sending instructions to other agents (vehicles) and objects (traffic lights) in the system. The director object can be programmed to trigger other actions depending on a variety of inputs including time, and proximity. Messages can be sent directly to a particular recipient or broadcast to near by agents.

Scenarios and behaviors are built by specifying HCSMs. This is done through a high level language which is then translated into compiled C code. Although a high level language is
provide for creating behaviors, due the complexity of behaviors and the resulting largeness of the HCSM programming error occur. As a mean of debugging and visualizing the running system, a graphical interface allows user to observe but not modify HCSMs while the simulation is running.

Wavish

Wavish [68] has developed a system for a high level virtual actor. Based on the virtual human work of Thallman [2], it incorporates intelligent agent architecture to create an autonomous virtual human which can interact with each other and human user to an extent. The basic building block for behaviors in the system is Communicating Deictic Agents (CDAs). CDAs are a hybrid agent architecture combining reactive agent design with some symbolic representation.

These agents can directly affect other agent's internal state providing for the ability of these agents to interact. An intelligent agent is built by constructing collections of CDAs, which is responsible for a skill the actor can perform. Every object in the environment is also represented by a CDA. This allows a particular agent to affect object by simply altering the internal state of the CDA object. Although it is a interesting approach there still is a lack of authoring tools for this system.

Herman-the-Bug

Herman-the-Bug [87] is a believable agent that acts as a tutor in an interactive learning environment. The use of a believable agent as a tutor allows end users to believe they are interacting with a creature which has own beliefs and personality. The agent is used to give advice to users and to suggest actions, which aid in achieving the goal of the lesson.

![Figure 8: Herman-the-Bug and Design-a-Plant](image)
Design-a-Plant is a learning environment in which users are given the task of creating a plant which can survive in a given set of environmental conditions. The learning environment itself is a screen-based 3D world. Users graphically build 3D plants from a pallet of plant structures, including stems, leaves, and roots. The intended plant environment is depicted as a rendered landscape, with specific environmental factors depicted graphically. All interactions take place in this design studio.

The tutor, Herman, is depicted as an insect. He can fly around the screen, make useful, and sometimes humorous remarks, and various behaviors. These behaviors include body-based behaviors, such as sitting, standing, and turning cartwheels. More importantly, Herman has advisory and explanatory behaviors, which cover plant anatomy, environmental constraints, and physiology. These behaviors are context-dependent, based on the agent’s behavioral engine. This engine uses such factors as current goal, the user’s success rate, and interaction history to sequence the next required behavior.

**Agent Design**

The Herman-the-Bug agent selects appropriate behavior based on the user’s actions during a plant design session. Since Herman-the-Bug is designed to act as a tutor in a learning environment, the agent must select the correct behavior based on several factors. These include: the user’s current task, how often the agent has had to advise the user on the current task, and the environment the designed plant is to grow in. The agent must also correctly sequence transitions from one behavior to another to prevent abrupt changes in behavior. All of the above tasks are handled by the agent’s control architecture.

The agent’s control architecture shown in is composed of several parts: Behavior Space, Behavior sequencing engine, Multimodel Behavior Stream, and the Learning Environment. The Behavior space contains stored sequences of actions. The Behavior Sequencing engine responds to requests from the Learning Environment by navigating through the Behavior Space and selecting relevant sequences of actions. The Multimodel Behavior Stream then composites them together to form coherent transitions between actions.

Herman-the-Bug represents the leading edge of the use of agents for tutoring environments. However, the system seems not to have support for creating these agents, as they must be constructed by expert designers and programmers. Each system must be completely designed and implemented for each topic covered. It typically takes a team of people to create the behaviors and actions needed to create the tutor.

**Cosmo**

Cosmo [71] is another pedagogical agent embedded in a tutoring system. Based on the same basic agent design as Herman-the-Bug, Cosmo is part of a learning system called The INTERNET ADVISOR, used to teach students Internet routing. Cosmo is also a believable agent with a wide variety of behaviors both physical and verbal. Comos’ major difference is his ability to refer to objects in the training environment.
Cosmo’s major difference in agent design is the incorporation of deictic believability. This refers to an agent’s ability to move through the environment and refer to object through a combination of speech, gesture, and locomotion. This requires that the agents behavioral system consider the physical properties of the environment. This function, Lester argues, is of importance in virtual learning environments, since the agents must be able to refer to object in the environment to clearly explain their function.

Deictic functionally is added as an additional sub-system in the agents behavior system. When the behavior planner determines that an explanation or advice must refer to an object in the environment, it passes a reference to that object to the dietic planner. This planner then computes what actions are required to refer to the object. This can include verbal communication. Moving to the object and/or pointing it. Once the diectic planner determines what gestures are needed, the behavior planner computes any required motions and gesture. While motions can be computed by a form of inverse kinematics, gestures are chosen from a precomputed library of gestures.

Cosmo is almost identical in design as Herman-the-Bug with the added ability of dietic believability. There still appears to be little effort in allowing user construction of lesson plans. Although given the research interest of Lester et al., authoring may not be the major research area. Like Herman-the-Bug most base behaviors, gestures, and visual are pre-constructed by animators and designers and placed in the agent behavior library.

AHA

Another example of an intelligent agent approach for behavioral animation is the Asynchronous Hierarchical Agent (AHA) work of Bruderlin [69],[72]. Bruderlin used an existing agent architecture that supports the BDI model of agent intelligence. Like other systems described,
AHA uses a layered approach to controlling animation from low-level control to higher level goals and actions. AHA was designed to allow users to more easily specify behaviors.

AHA adopts the layered agent approach of Muller's InterRAP agent. The approach taken by AHA is to replace the three layers of InterRAP (BBL, LPL, CPL) with seven layers of control as shown in Figure 10. These layers are similar in design to the layered approach taken by Blumberg. The several layers handle the following functions:

- Motivations: handles an agent's goals
- Policies: determines which behaviors to activate
- Behaviors: goal directed sequences of actions
- Actions: automated sequences of atomic actions units
- Action Units: sets of DOFs
- DOF: sensors and effectors, which interact with the world.

The authors argue that this hierarchy allows users to manipulate agent activities at various levels of abstraction. These layers communicate with its immediate predecessor and successor via the knowledge monitor described below. As in InterRaP the perception travels upward through the hierarchy while responses travel downward. The intent from the InterRaP design was to allow a layer to handle a response if it was able to or pass it up to the next level if it was unable. There seems to be no indication that this procedure was incorporated in AHA.

The knowledge monitor acts as a communication buffer between the various layers. It also stores perceived knowledge about the world. The knowledge monitor is composed of three sections: the knowledge base, the activity manager, and the resource manager. The knowledge base contains global variables and state information shared between layers. The resource manager coordinates allocation of resources, while the activity manager acts as a server to the layers of the hierarchy. The activity manager maintains three classes of information: all currently active activities, events associated with the activities (starting stopping), and a list of all the filters waiting for events and activities to occur. A filter is represented by a four-tuple:

\{ \text{start activity} \} \{ \text{running activity} \} \{ \text{stop activity} \} \rightarrow \{ \text{command to execute} \}
When all conditions evaluate to true the command to execute is executed. One or two conditions can be replaced by nulls ( { } ) in which cast they always evaluate to true.

Each agent has its own knowledge monitor to maintain the current state if the agent. Additionally there is one knowledge monitor, which represents world information. This is used to track and monitor events, which occur in the world. This is essentially a blackboard for allowing message exchange between agents. Note that this design too is a departure from the InterRaP model. The InterRaP design has no blackboard as agents are able to exchange message between agents directly. The system was used to create a board game in which the participants are AHA agents.

**Virtual Jack**

Virtual Jack [41] is a system for accurate modeling of human bodies in a virtual environment. Jack is designed to be an intelligent avatar, meaning that basic functionality is programmed into Jack. This is similar to other systems mentioned in this survey primarily Alive, Oz, and Improv. Jack is composed of two levels: a layer which control basic motions of joints and limbs through manipulation of joints (DOF’s) and a higher level which can execute scripted motion and/or plan complex tasks.

While Jack is not agent based, much as in the same manner as Alive and Improv, it does provide support for some reactive behavior as well as some planning capability. Jack’s major strength is that the movement of the virtual human can be so accurate that it can be incorporated into ergonomic studies. This is evident in the level of grasping that the Jack avatar is able to perform.

![Figure 11: Virtual Jack](image.png)

Jack has two levels of operation: a low level marionette mode, and a high level, which can handle task-based operations. In the low-level mode, Jack can be used to “play” motion capture data, direct sensor data, or synthesized data. The high-level task mode provides for tasks based motions, and execute parameterized scripts.
Jack’s high level architecture is built upon Parallel Transition Network (PaT-Nets). These structures are Finite State Machines in which nodes represent actions. Edges represent conditions which when true enable transitions to connected nodes. Multiple PaT-NETs can be active at any one time. Coordination between active PaT-NET is achieved through message passing and a global memory.

Although Jack provides a high level interface for basic actions, there is little support for adding additional behaviors to Jack itself. This is primarily because Jack is design to be incorporated into other programs that provide higher level control. Some of these systems are: SodaJack which is a high-level planner attached to Jack, Steve (described below) an intelligent agent for training in virtual environments, and Jack Presenter, a presentation system.

Steve

Steve [78] is another virtual human system. Like Herman the bug, it is a pedagogical agent. It is perhaps the most “advanced” of the agent-based behavior models in this survey. Steve is designed for Virtual training environments in which he inhabitants are virtual humans. By combining the Soar agent model with virtual Jack, Steve becomes a virtual human that is able to perform problem solving and learning. Steve’s behaviors can be either programmed by using a scripting language or programmed by example in the virtual environment. Steve is incorporated into a virtual training environment called VRIDES.

Steve like many of the system in the survey is composed of several subsystems. These components are: Motor control, perception, and cognitive. The cognitive component is responsible for high level behavior such as determining what to demonstrate and explain. It also performs situation assessment, determining if a goal or task has been met, or if a new goal is to be constructed. The perceptual component provides the cognitive component with a model of the environment whenever an even occurs which effects the environment.

The motor component is controlled by the cognitive component. It performs the necessary motor actions to perform tasks given to it. It can perform two or more task simultaneously. It performs tasks asychornously and informs the cognitive component when actions are complete. The command that the motor control module can handle include: manipulating objects, pointing to object, moving to objects, and speech by using Entropic’s TrueTalk system text-to-speech system. Low level motor control is actually handled by Jack [5] since Jack is able to handle higher level behaviors.

The perception component receives messages from the environment including: object state changes, actions taken by other users in the system, whether they are other agents or users, position, and orientation of users, and current time of the global clock. Base on this, input the perception module update the agent’s symbolic model of the environment. Additionally the perception module keeps a running history of significant events that have occurred.

The cognitive component of Steve is build upon Soar [44]. Soar is an intelligent agent architecture, which has been in development for the past decade. It stores both task and control knowledge. It constantly reevaluates which operators to select based on the state of the environment at a rate of 10 to 100 HZ. It maintains an internal mental model of the current environment and uses this model to select operations to perform.
Additionally Soar is able to learn from previous experiences and infer new knowledge from its metal model. Soar has been used to build combat agent for distributed battlefield environments cite Tambe [50]. Since Soar does not have, support for pedagogical capabilities Steve extends Soar’s architecture to incorporate additional cognitive capabilities. These include: episodic memory, demonstration, explanation, and student monitoring.

Steve pedagogical capabilities make use of a hierarchical plan representation. Each plan has a set of goals that must be achieved, and the steps needed to complete these goals. Each step can be itself a plan or a primitive action such as grasping an object. Steps can be ordered so that there is a temporally causal relationship between them.

Steve was also designed to allow for end users to construct lesson plans, and teach Steve new tasks. To facilitate this Steve allows plans to be specified by example. To train Steve a user starts to demonstrate a new task. If it perceives an action that is does not recognize it asked the instructor for a name for the action along with a text describing the purpose for the action. Steve also incorporates GUI interface to allow instructors to define lesson goals.

VPL and Agents

Visual programming (or hyperprogramming) has been used to program agents of various types. We have seen several example of this in the above sections: Notably the work of Del Bimbo and the SAM system. Since these system in various degrees are based on virtual reality they were not included in this section. However, the following two systems described below are the best example of how agent behavior can be programmed visually

Agentsheets

Agentsheets is a system written by Repenning [9],[57],[38] in which allows for visual programming of agents. It differs from other VPL’s describe here in that it serves a dual purpose: to provide a domain-specific visual programming language for end users, and to provide a system for constructing these domain-specific VPL’s.

Repenning’s point is that general purpose visual programming languages have various pros and cons. The advantage of general-purpose visual languages is that then can be applied to a wide range of problems. Additionally syntax can be better enforced and illustrated through their use. General purpose VPL’s however suffer from some major difficulties. Because these languages do little to abstract the underlying syntax of their base languages they are difficult for end users to program and are of little use to expert programmers. The basic building blocks of general-purpose visual languages are often too closely tied to the underlying programming language. For example, procedures are represented by block and parameter passing is represented by incoming arrows.

Domain specific visual languages on the other hand, are often more usable since they provide representations of user’s tasks and object which are more relative to the programmer’s mental model. Repenning cites The Pinball Construction Set as one example of a domain-specific (and quite successful) visual language. While this language is suited for visually constructing pinball games, it would be of no use for numerical tasks. These visual languages can be of great use for end user programming, however providing a specialized visual language for every domain can be difficult and time consuming.
Agentsheets is primarily a system for creating domain specific visual languages. Using Agentsheets, visual languages have been created. These include: flow based visual languages like water flow and electrical circuits. In this application, a VPL designer defines a set of agents, which have the behavior of electrical components such as switches and wires. A user can create a circuit by composing it from these components. A user can interact (or run) the circuit by opening and closing the switch.

Ecosystem simulations in with the graphics elements represent land structures such as mountains, wetlands, and animals. End users program the simulation by specifying simple rules for each element, the goal being to establish a stable ecological system. A voice mail visual language, which allows for the creation of voice mail systems, is an example of a real world application of Agentsheets.

Agentsheets has a unique underlying design. It extends the spreadsheet design by create a spatial grid in which each cell of the grid is an agent. These agents form the basis for all visual programs. Since, again, the definition of an agent mean is quite varied, Repenning defines an agent to be: a fined grained, autonomous unit of behavior. The agents consist of:

- Sensors
- Effectors
- Behavior
- Depiction

Sensors are stimulus which invoke the actions of the agent. Effectors are used to communicate with other agents. Messages can be sent to grid coordinates or through links. Behaviors are rules-set defining the actions of an agent. Each agent has a default behavior which can be refined by sub-classing the agent. Depiction is a graphical representation (an icon) assigned to an agent, which often represents its behavior.

The grid structure of Agentsheets is used to specify spatial relationships such as distance, orientation, adjacency, and position. The use of the grid structure is useful for several reasons. One is that by splitting the space into a discrete grid the chances of a mismatch for a rule is minimized. The grid also simplifies communication between agents; communication between agents is implicit if they lay in adjacent cells. This results in the end user not having to worry about setting up agent communication.

End users program by placing agents in a meaningful manner throughout the grid. These agents can react to user action – such as mouse clicks, dragging, and keypresses. Agents can also be stacked atop one another to occupy the same grid cell.

The VPL designer’s job then is to define the basic agent and its tasks. This include each agent’s basic behavior, it’s sensors, it effectors. The designer must also define the behavior between agents. Behavior is specified by defining set of rules. Each rule has an enabling condition (the left-hand side of the rule) and its action (right hand side). Additionally the VPL designer must supply a set of icons, which (hopefully) have meaning to the end user programmers. In the circuit examples above, the designer of the visual language is responsible for building the agents so that they correspond to their real-world equivalents.
Agentsheets is an interesting example of how domain-specific visual languages can be used successfully for end-user programming. Evaluation of the success of Agentsheets can be found in [9].

Cococa

Cococa [43] formerly known as KidSim is a visual programming language written for children. Cococa’s visual programming paradigm is that of programming by example. In this case to visual program agents to execute a set of rules. Cococa allows for the construction and modification of dynamic 2D simulations. Cococa’s simulations consists of: a gameboard divided into grids (similar to Agentsheets), a system clock, simulation objects (the agents), a rule editor, and a few other miscellaneous elements. Cococa allows for end user programming by combing two visual programming paradigms: visual rule and programming by example.

The game board represents the simulation environment. Since the board is divided up into grids to allow easier interpretation of the actions to be performed during rule evaluation. This is similar to both Visualan 3D and Agentsheets. Furthermore, time is divided into discrete ticks, which proved fine grain control over the simulation.

Agents in Cococa are defined as “a persistent software entity dedicated to a specific purpose”. The author clarifies this definition by explaining that these agents have a set of tasks and know how to accomplish them and are not full-blown programs.

One of the more interesting aspects of Cococa is the compositing of graphical rewrite rules with programming by example. This composition solves a problem with visual programming languages that use graphical rewrite rules. This problem arises from the fact that is it hard to abstract action to perform between the condition and action states of a rule. This arises because the computer has to compute what action to perform to translate the before image into the after image. Cococa solves this problem in the following way: once a rule’s enabling state is defined, the computer records the actions the user performs to get to the result state. In other words the rules action is defined by the example given by the user.

Another problem addressed by Cococa is that of rule abstraction. Once a rule is defined how can it’s enabling condition be abstracted to fit general cases? This problem is address in two ways: picture abstraction, and property abstraction. In picture abstraction, objects in the “before” (the left-hand of the rule) may be selected and replaced by a more general class of objects. In property abstraction, tests can be added to the enabling condition of a rule to control the circumstances in which a rule can be enabled. Both of these properties can be combined. Therefore, it is possible to generalize a rule and add one or more property abstractions to it. In order for a rule to match its enable condition along with all of its property tests by evaluate to true.

Games

Although not in the mainstream of research, games have to solve many of the same problems encountered in virtual environments. Games are typically designed to run in a real-time, dynamic environment. Users typically interact with simulated characters, called (in the game developer’s world) Non-Player Characters, or NPCs. A NPC must exhibit believable behavior in order to make the game more realistic. In the past NPCs were typically constructed from very simple routines, which allowed them some level of believable behavior, which is typically predicable once a user has interacted with them.
Today there is more of a push for creating more realistic behavior. Game developers are starting to adopt techniques from AI and artificial life to produce more believable and intelligent NPCs.

AI in games usually falls into one of three categories [81]:

- Rule-based/FSM
- Artificial Life Based
- Genetic Algorithms

**Rule-based/FSM**

Rule-based AI is the most common form of AI for NPC. This is due to game developer’s familiarity with them and their ease of use. It typically is in the form of Finite State Machines, or FSMs. These rules are constructed into a hierarchy of rules and conditions. Each rule invocation is played out in exactly the same order. Condition states determine which rules are available to fire. For each condition there is exactly one outcome, which leads to predictable behavior in most games. A popular variation of FSMs are Fuzzy State Machine. The different in fuzzy state machine the main difference is that there is a set of possible outcome that can be either nondeterministically chosen or each outcome can be weighted to give an outcome preference over another. The PaT-NET’s of Jack and the work of Cremer et. al. also make use of finite state machines to enable actions.

**Artificial Life**

Artificial life is useful for creating simple reactive NPCs. The flocking behaviors of Reynolds have found their ways into game. ALife allows for more reactive NPC, which is not only affected by the user but also by other elements in the environment. For games, which take place in a biological environment, an ecological system can be built using ALife techniques.

**Genetic Algorithms**

Although not discussed before now, genetic algorithms allow an NPC's behavior to evolve over time. New behaviors are derived from previous behaviors over time. It works by creating chromosome sequences, represented by a string. These chromosome sequences represent the various rules that govern behavior of an NPC. In an environment with multiple creatures, the chromosome sequences are evaluated against form fitness criteria. Failing sequences are discarded while succeeding ones are combined with other succeeding sequences to form new sequences.

Cyberlife's [70] creatures combine neural networks and genetic algorithms to define the behavior of virtual pets. Called Norns they learn over time to speak, and interact with users. They live in a virtual world and learn as they explore the environment. Every behavior of the Norn is encoded into it and by breeding selected Norns, successive generations develop new characteristics.

Note that even simple genetic algorithm based behavior evolution has an advantage over existing agent-based systems, which do not typically add new behaviors over time. The exception being the Steve agents' Soar architecture, which can derive new behaviors from old one, although not "genetically" It should also be noted the Muller proposed doing this for the PoB in the InterRap aritecture.
Some systems also allow for authoring the behaviors of NPC in certain games. A brief survey of the current available systems is given below:

**Quake**

Quake is a first person based 3D interactive game. A successor to DOOM it provides an editor and programming language called Quake C. Using this, users are able to create new levels, as well as define new behaviors for the monster inhabitants, called Bots. Additionally this has led to an Internet Bot exchange, were player's can swap Bot's and plug them into their levels.

**Unreal**

Unreal a successor to Quake also allows users to define their own behaviors. UnReal, however, actually provides a high level-programming environment called Unreal Script to define behaviors. Instead of explicitly programming, users are able to give high level commands such as Run forward two meters, turn, fire weapon. Players have also been developing new Bot's from tough opponents to companions who assist players.

**Motivate**

Motivate [82] is a commercial product from the Motion Factory. Designed as both an authoring tool and a game engine Motivate is designed to allow a non-programmer to construct complex characters. It has several components which allow for constructing motions and behaviors the most interesting of which is the Behavior Modeling component. Behavior modeling in Motivate allows for goals, reactivity, and states of mind. The model is decomposed as follows:

- Actors
- Skills
- Behaviors
- FSM
- Inverse Kinematics

Behaviors are models using Hierarchical Finite State Machines. These are FSM in which the nodes (states) can contain sub-states, procedural code, variables, and transitions. Note that the HFSM are very similar to both Virtual Jacks Pat-NETs and Cremer's HCFMs.

HFSMs are editable using a flow-chart programming paradigm. States are represented as rectangles, transition by arrow. The hierarchy is handles by embedding the iconic representation of a HFSM inside a node of a higher level HFSM. Clicking on a state or transition allows a user to edit its properties, or add procedural code written in a scripting language called Piccolo. This editor does suffer from the screen space problem as large HFSM can quickly fill the screen.

A Skills editor is available to create simple skills. A user positions the actor into a position and saves it as a key frame. Key frames are then interpolated to produce a skill. Once a skill has been created it can be used by the behavior engine in Motivate. Motivate does have build in collision and path generation so that an actor can avoid obstacles without having to explicitly program it.

While Motivate is perhaps the most advance tool for end user programming, its underlying model of behavior is more computer graphics based than AI based. It provides no means of coordinating actors, and high level goals and belief are not provided for.
Conclusion

The systems surveyed in this chapter represent pieces of what can be done for programming of intelligent agents for virtual reality systems. This chapter has surveyed both historical and current work in these areas. It also has described systems built by combining virtual reality, agents, and visual programming. Each of these systems contributes to each of their parent fields. As we have seen, these systems are varied in design and use. Figure 12 below show a roadmap of how these systems relate to each other and their parent disiplines. The center area represents systems that allow for visual programming of agents that are incorporated in a virtual reality system. It can been seen in the map that while system have been developed for creating intelligent agents there has been little work on providing convenient programming interfaces to them. The hope of HAVEN is that it will be able to extend the work described in this chapter to provide a tool for the visual programming of intelligent creatures.

Figure 12: Programmable Agents for Virtual Reality a road map.
Chapter 3 - Building the Perfect Beast

Introduction

One consequence of the increased availability of low cost 3D graphics hardware is that the demand for providing quality content will also rise. The current state of home game machines has demonstrated that worlds which are both visually rich and interactively engaging characters are in demand. The current set of tools, which can be used to design such creatures are not as fully developed as they could be. Although there is a large based of modeling tools for creating object for 3D graphics the set of tools for creating interactive actions and behaviors is far smaller.

The current state of behavior systems that can be incorporated into VR was detailed in the previous chapter. Most of these systems allow for a hierarchy of behaviors, ranging from low level actions to high level plans. This stems mainly from the design of the reactive behavior model of Brooks [22]. Although these systems provide a basis for implementing behaviors, few provide authoring tools for building them. This lack of authoring tools and the limitations of reactive architectures leave room for a more fully developed system for behavior modeling. The existing systems reviewed in the previous chapter can be used as a basis for creating such a system.

Why is there a need

The current use of reactive systems for behavior modeling has resulted in issues that directly affect both the modeling of behavior and specifying it. Reactive systems are just that, they react to their environment. Consequently, overall behavior "emerges" from interactions with the environment. While these systems have proven to be successful, they do have inherent problems. One problem is that because these systems react to changes in their environment, overall behavior is difficult to predict. This does not imply that this approach is wrong: one simply has to attempt to hit a fly to know that this design does have its strengths. The reactive design is useful for creating simple creatures, but when it used to create higher level behaviors the hope that the correct behavior will emerge from the system leaves much to be desired for someone trying to construct a pedagogical agent for a tutoring system.

Systems such as ALIVE and Improv which provide high level behaviors, rely on the concept of competing actions. These actions are selected by evaluating activation and inhibitor values. Action selection then becomes a matter of evaluating all possible actions and determining which has the highest value. Typically these values are the primary way that behavior is specified. This results in finding, usually by trial and error, the best set of value which give the most consistent and desirable behavior. Determining these values become much more complex as the number of rules, goals, and conditions increase, as they are also dependent on the characteristics of the environment.

The arbitrary hierarchy of behaviors is another difficulty with reactive architectures. As there are no defined levels of actions in reactive system but a continuous range, there are no clear boundaries between purely reactive actions and more deliberative ones. While not undesirable, it can make specifying behavior difficult.
Hybrid agent architectures have been proposed to address the shortcomings of purely reactive agents. Hybrids combine deliberative and reactive architecture to form a reactive system that can also support goals, usually following the BDI design. The use of hybrid architectures for behavior modeling in 3D environments is largely an unexplored area, although there has been a call for more incorporating a more advanced agent design [52],[86],[77] in virtual reality systems.

Steve is the closest to a hybrid agent incorporated into VR. Steve combines the planning and learning capabilities of the Soar architecture with the reactive abilities of Jack. However, the Soar architecture is large and perhaps too complex for creating general-purpose agents. Steve was not intentionally designed as hybrid architecture. It is built from various components, which have proven to be difficult to cleanly integrate together [78]. AHA [69] claims to use the hybrid architecture of InterRap however most of the deliberative aspect seemed to have been removed from the resulting implementation.

Programming tools for these behavior systems are also lacking. Most systems provide an API bound to a high level programming language, or a scripting language for specifying behaviors. For example, Improv provides a scripting language for specifying behavior. Behaviors are prioritized by the order they are listing in the file. Lower level actions must be specified by defining each DOF used. Alive provides a C++ API for which to construct behaviors. However, a complex creature takes 1000's of line of C++ to define its behavior. Systems such as Herman-the-Bug and Cosmo, while very engaging also are also handcrafted. Other programming environments such as VRML 2 and Performer are tied to programming languages. These systems provide little in support for behavior specification. In order to construct behaviors everything from low level graphical transformations to high level actions must be explicitly programmed.

Motivate and Steve provide alternatives to writing code, although with limited success. Motivate uses a visual editor for composing behaviors with HFSM. Although interesting in design it seems to be mostly used for composing action sequences and cannot be used for creating complex deliberative actions. Steve provides a program by example system for the creating of plans. This system is the most advanced of any authoring tools. The designers of Steve considered authoring tools to be an important element of Steve's overall design. Most of the authoring support for Steve in the form of interfaces which are used to set parameters of the system. The PBE system, while intriguing is limited to specifying one subset of behavior. It does represent a step in the right direction.

**What is needed**

A system for designing and building intelligent creatures/humans, which can be used in virtual environments, would be advancement on the current state of behavior modeling systems. By expanding upon the work done in the past, a new system could be built that would reduce the limitations discussed in the previous section. This section will describe a wish list for such a system, following that will be the proposal for prototype of such a system.

The current use of variations on reactive agents should be replaced with a hybrid design. This not only would give these agents more “intelligence” in terms of planing and goal generation, but the design could help in programming these agents by provide clear boundaries for behavior specification. A generic agent architecture would be desirable. This has been suggested by Sloman [86] and others [49]. Such an architecture could be augmented with emotion, and
pedagogical capabilities to provide a system that could create creatures that range in abilities from purely reactive insects to virtual tutors.

Of primary concern is to provide support for authoring behaviors. While there are systems for creating the geometry of objects, there is little support for creating dynamic and interactive behaviors. There exist authoring tools for creating animation through keyframe and motion capture such as SoftImage, and Alias, however these are capable of creating only non-interactive repetitive animation. There has been little work in developing tools for creating and designing dynamic actions, autonomous behavior, and highly complex plans. For example, one of the most engaging creatures discussed in the previous chapter was Herman-the-Bug. However, Herman’s behavior and emotion are built from scratch. Such a creature is beyond the ability all of the surveyed systems to create without extensive programming.

A programming interface to autonomous agent architecture would allow for a range of behaviors from simple to complex to be easily designed and programmed. Because of the different requirements for low-level actions vs. high-level plans, it is highly unlikely that the same programming tools would work for the entire range. These tools could also be constructed to provide support for a range of programming from visual to a language bound API. Additionally these programming tools should work from within a virtual environment. The 3D visual languages described in the previous chapter could provide a foundation for such tools. Visual programming languages have been successfully applied to agents for specifying their behavior. Ideally, a system, which incorporated a generic agent design with a tier of programming tools, would provide an ideal programming environment for creating intelligent agents. This approach has been successfully been developed in AgentSheets. By providing for simple behaviors to be pre-built into these agents, the programmer can create complex behaviors without having to be concerned about all of the low-level details.

It is also desirable for any system describing behavior to run independently of the display environment. This allows the system to run at a different rate from which it will be displayed. This allows for multiple viewers with different from rates to be connected to the world model without the slowest viewer in the system affecting the others. This model has been used successfully in systems such as Improv.

Proposal

The proposal for this thesis is to provide architecture for the system described above by combining hybrid agent architecture with a visual programming environment. Called HAVEN (Hyper-programmed Agents for Virtual ENvironments) it incorporates features of the idealized system described above. The overall design is comprised of four modules: a generic agent design, a centralized management module, a programming environment, and a display module. The modular design allows each module to be developed independently and reduces the time incorporating the prototype into virtual reality systems.

Agent model

While the focus of this thesis is to apply hybrid agent architecture to behavioral modeling, it is beyond its scope to develop a completely new model. Therefore, it is advantageous to use an existing hybrid model. InterRap by Muller is such an architecture. The choice of InterRap over others such as Touring Machines and COSY is two-fold. Primarily this design has been used
before by Muller and others for dynamic simulations [30],[58],[88]. InterRap [58] was used as the basis for the AHA system [72], although it removed most of the functionally of the higher levels to form a strongly reactive architecture. InterRap is also considered one of the most advanced hybrid agent designs [52].

Secondly, a more grounded reason for choosing InterRap is that it is well documented in [58]. This is important since the system will be rebuilt with additional changes. This includes adopting the basic algorithms to a multi-threaded design and incorporating a distributed virtual scene graph to handle geometry, and adopting it for use in virtual reality environments.

Design

An InterRap agent is composed of three modules: Knowledge Base, World Interface, and the Control Unit. The Knowledge Base holds information about the agent’s environment. The Control Unit is composed of three layers: the Behavior Based Layer (or Reactive Layer), the Local Planning Layer (Deliberative Layer) and the Cooperative Planning Layer (Communicative Layer). The final component of the InterRap design is the World Interface (WIF). The WIF module is represents an agent’s interface to its environment. Through it an agent’s sensory perception, communication, and motor actions are controlled.

InterRap is a vertically layered agent. The hybrid nature of the agent results from these layers. All activity originates in the lowest layer, which is the reactive layer. The reactive layer is the only layer able to interact with the World Interface. Therefore, it is the only layer able to read sensory input and issue commands to the effectors. If a particular layer determines that it is unable to handle the current situation control is passed to the immediately higher layer. This process continues until a layer can develop a plan to respond to the situation. If no plan can be determined then an emergency plan is executed. The overall result is: as new situations occur they travel upwards through the layers until a layer competent enough that handle the situation is found. Once a layer determines the actions needed to handle the situation these actions are propagated downward through the layers.

![Figure 13: Original InterRap Design](image-url)
Each layer consists of two principle processes: the situation recognizer and goal activator (SG), and a planning and scheduling process (PS). The SG process enables a layer to identify the need for action. Situations in the reactive layer, for example, are the enabling conditions of rules, which govern reactive behavior. Goal activation is similar. In response to a situation that the agent has recognized, an agent’s internal motivational states are altered. This results in changing the current activity of the agent in favor of pursuing the newly activated goal.

The PS process is responsible for committing to an action generated by the SG process, or sending the situation to the next high level if it determines it is unable to handle it. This process is composed of three steps: planning, scheduling, and execution. The planner is responsible for computing if a situation/goal can be handled by the layer. If it cannot it is passed to the SG process of the next higher layer. If the situation/goal is to be handled then the steps required to respond to the situation are determined, and passed to the scheduler. The scheduler merges the steps required to complete this action with the currently active plans. This is then passed to the executor, which is responsible for executing atomic actions of all of the active plans. It is also responsible for informing the SG process that a particular Situation/Goal has been achieved or has failed.
Each control layer follows the same basic algorithm of: sense-recognize-decide-act. Each layer executes the following algorithm:

\[
\begin{align*}
\text{Beliefs} &= \text{update_beliefs(Beliefs)} \\
\text{Situations} &= \text{update_situations(Situations)} \\
\text{Goals} &= \text{activate_goals(Goals, Situations)} \\
\text{Intentions} &= \text{operation_select(Goals, Situations)} \\
\text{IntNotHandled} &= \text{check_competency(Intentions)} \\
\text{For all } I \text{ in } \text{IntNotHandled} \\
& \quad \text{Inform_layer(CurrentLayer+1, I)}; \\
\text{Schedule} (\text{Intentions}); \\
\text{Execute} (\text{Intentions});
\end{align*}
\]

**Figure 14 Control Layer Algorithm**

The agent’s current beliefs are updated with new situations and by messages received from adjacent lower layer. These situations are used to update the agent’s current set of pursued goals. These situations and goals are then passed to the planner/scheduler (PS) as pairs. These pairs are then evaluated to see if it can handle the Situation-Goal pair. If it cannot then it is passed to the next higher layer. Situation-Goal pairs, which can be handled by the layer, are then added to the agent’s intentions. These intentions are then schedules for execution.

The Knowledge base stores information, which represents the agent’s knowledge about its environment. Like the control unit, it is layered. Each layer of the control unit is able to access the knowledge base at its level and below. This restricts lower layers from accessing information from a higher layer in the Knowledge Base. The Lowest layer of the KB stores the agent’s environment attributes, current goals, and it’s behavioral rules. The mental model stores plans, goals, and the plan library while the highest level (Social Model) stores group plans, and shared information.

Rules in InterRap are named Patterns of Behavior (PoB). Each PoB is composed of two parts: attributes part and actions. The attributes include Name, State, Resources, Priority, Activation Condition, Success Condition, and Fail Condition. The three conditions are used as follows: The Activation condition determines what situation causes the PoB to become active. The Success condition is used to determine when the rule has been finished, and the Fail condition is used to determine when a PoB cannot continue to be active. The execution part contains a list of commands to execute. These can include activation of effectors, reading sensors, and activation of other PoBs.

Plans are compositions of PoB. Plans can be thought of as a way of grouping sets of rules, which perform coherent actions and/or are used to achieve a goal. Plans are stored in the Knowledge Base. Plans can also computed on demand by the planning system. This computation is a classic AI problem: to develop a plan to accomplish a goal. While InterRap does not provide this, agents of HAVEN will.

The topmost layer of InterRap allows agents to communicate. By communicating agents can negotiate to perform joint plans. These plans involve two more agents coordinating tasks to solve a joint goal. Joint plans are initiated by the agent if it determines that it cannot respond to a
situation on its own. If a joint plan is devised each agent involved is given a set of sub plans to perform. The communication layer also provides a mechanism for sharing information among group of agents.

Communication between agents is an active area of research. With the development of a standardized communication language for agents KQML, and a knowledge exchange protocol KIF, agent communication has become possible. If agents were built with the inherent ability for both agent-to-agent and broadcast communication it would be possible for virtual agents to communicate with other types of agents such as web-based agents. Agent communication would also allow the world to be distributed across machines. World knowledge could be restricted to agent querying a central store for global information and nearby agents for local information. The use of KQML gives an agent the potential to communication with other agent systems and knowledge stores.

HAVEN's InterRap Design

Although the agent design of HAVEN is based entirely on the InterRap design described above, some changes will be needed in-order to provide a visual programming interface and to incorporate it into virtual environments. Although the basic design is more than adequate, the world interface requires some redesign to work with the motor control system. Additionally agents require an appearance, and an improved effector system.

The proposed new design is show in Figure 3. The world interface has been expanded to incorporate an agent appearance and a Motor Skill layer which is based on the one used in ALIVE. The appearance is stored as a hierarchical scene graph. The hierarchical structure of the scene graph lends itself well to the motor control system, and provides opportunities for LOD and optimization. The scene graph will be based on VRML2 as that format can be both produced and read by a variety of modelers and graphics engines.

The motor skill system is incorporated into the WIF. It is built as an additional layer immediately below the reactive layer. It receives commands from the reactive layer in the form of motor control commands. As the commands are received, they are index against the known motor skills. A motor skill is then loaded. The DOFs required for execution are then allocated. Note that the DOF’s replace the effectors in the WIF of the original InterRap design. These DOF are linked to transform node in the agent’s appearance. Commands are received by the controller that is responsible for mapping commands such as forward, turn, and halt into the appropriate motor control actions. The Motor controller is responsible for allocating all the required DOFs needed for the action. Once all of the DOFs have been allocated, the motor skill becomes active. Once active the controller is responsible for adjusting all of the DOFs to produce coordinated action. The DOFs are allocated from the set of all the DOFs associated with transformation nodes in the agent’s scene graph. These DOFs can be member of more than one Motor Skill. Each DOF contains a lock, which is used to allocate DOFs to a motor skill. The Motor Skill only becomes active if it is able to allocate all of the DOFs it needs. The DOFs contain a reference to a transformation node in the scene graph. As they are updated, the related transformation node is updated. The motor skill layer can also send information to the Reactive Layer in case of error or allocation failures.

All agent regardless of how many DOF they contain always have on top level DOF which represents the root of the scene graph. This DOF then becomes the local coordinate system of the agent. The used of this root level node allows basic Motor Skills to be assigned without having to
develop ones that are more complex. Additionally basic motor skills such as move forward and turn, which can be applied, to a wide range of agents will be available as a library.

Motor skills also take parameters, which affect their execution. For example, move forward can be augmented with a parameter that can be used to alter the speed of the forward motion. The reactive layer can create these parameters if it determines that the agent default speed is inadequate for the required action. This allows for example an agent to slow as it approaches an obstacle as opposed to an abrupt stop.

Figure 15 HAVEN's InterRap Agent

The agent’s world interface is composed of sensors giving the agent perception. Sensors are of two types: omnipotent and direct. Omnipotent sensor queries the world model for details about it, or receives information as events. They have full access to any information stored in any agent,
For example, a sensor can directly query an agent’s position by accessing its attributes without having to see the agent and compute its location. Direct sensors perceive the environment through other means such as synthetic vision. These sensors are bound to node 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.

InterRap was originally written in a concurrent language called OZ, however it is not a common language and runs on a very restrictive set of machines. A version of InterRap, implemented for HAVEN in C++, has already been undertaken. As of now, the basic reactive layer of InterRap currently exists as a C++ library. The WIF and KB also have been implemented. A scripting language and API for creating agents and reactive rules is also available.

HAVEN also extends the original InterRap design by providing for multithreaded execution. This extension allows each agent to run as its own thread. This is preferred because each agent should run independently of each other. This prevents a slow agent from affecting all of the other agents. One design issue is how fine-grained the threading should be agent based (each agent as its own thread), layer based (each layer is its own thread) or action based (each action is its own thread). Threading support is provided through Object Orienting Concept’s JTC, which is a C++ implementation of Java threads built using POSIX or DCE threads under Unix and Win32 threads under Windows 95/98/NT. It allows for a system independent thread library, which functions similar to the threads in Java.

The agent’s appearance is loaded from a VRML 2 scene graph. The VRML 2 support is provided through a C++ implementation of a VRML 2 parser. In the current implementation, the parser loads the VRML 2 file into a memory resident scene graph. The Motor Skills layer of the agent then uses VRML 2 transformation nodes to implement DOFs. Each DOF is assigned to a VRML transformation node. Any changes in the DOF are sent to the transformation node. It should be noted that the scene graph is simply stored in the agent not displayed. The graphics display module discussed below is responsible for displaying the agent’s appearance.

All of the data structures required by HAVEN are provide by the Standard Template Library (STL). The STL is a library of C++ template, which implements common data structures as C++ templates. Data structures such as lists, trees, and hash tables are provided as well as functions to perform operations on them. Most C++ compilers support it.

HAVEN will run on SGI hardware running IRIX Performer and Intel machines running Windows NT and Cosmo. The availability of C++, which support STL, Run Time Type Identification (RTTI) and namespace for both platforms allow for this. With the increased graphics and CPU power of Intel-based machines, HAVEN should perform well on an NT platform.

World module

If agents are to be used in a virtual environment, they need a support framework from which to operate. The world module contains everything necessary to create and manage agents. This module is responsible for maintaining the world appearance, agents, user locations and appearance, and global state information. Its modular design allows for both the programming and world modules to interface with it without much programming effort.

The world module is design as part of a networked component. The overall structure is show in the figure below. The World Module servers as the center of a star network configuration. All
update to the system are received and propagated to all connected modules. The world module is designed as a server for agents, agent clusters, users, and display devices. The overall structure of the world module server is shown in the figure below:

![Diagram of World Module Design]

**Figure 16 World Module Design**

The world module consists of several subsystems: a connection manager, and agent manager, system clock, a list of active connections, and world geometry stored as a scene graph. The connection manager listens for requests from clients. These can be agents or users. Once a connection request has been received, the connection is placed on a list of active connections. As updates to the world are received, the connection manager updates the world scene graph and then propagates these updates to the remaining connected clients.

The system clock is used to log events, and coordinate the actions of agents. The system clock provides a “heartbeat” for agent coordination, interpolation of Motor Skills, and time stamping of communicative events. Its secondary purpose is to provide a mechanism for the scheduling of temporal events, which can include scheduling of plans at some point in time, or using temporal operators to specify reactive situations, such as move to location (x,y,z) in five minutes.

The world module is also responsible for maintaining all graphic information of the virtual environment. An entire world’s geometry is stored in the world scene graph. The scene graph contains the agent’s appearance, the scene graph representing all of the non-agents objects in the scene, and the representation of any users in the environment. The scene graph uses techniques for developed in [80]. The scene graph is used by the agents synthetic sensory system to query the
state and location of other geometry in the scene. The world module is responsible for maintaining the consistency of the scene graph between itself and all connected clients. By storing the master copy of the scene locally in the world module, all agents have access to any information they might need concerning the state of the virtual environment.

The World module is used as a container for managing a virtual environment. Its modular design allows it to interface with the various display modules. It also decouples behavior cycle time from display cycle time. This prevents slow display devices from affecting the response of the behavioral system.

Display Module

The display module is a base class (to use OOP lingo) which acts as a client of the World Module described in the previous section. The Display Module has several variants, which are described below, however the basic responsibility of the display module is the same regardless of the type of DM. The basic components shown in Figure 17 are: an agent or user interface, a mirror of the scene graph stored on the world module, and a connection to the world module. Each display module can contribute its own piece of the world scene graph. Any action which results in a change of state for any node in the Display Modules local scene graph is reflected back to the World Module which then update all of the other connected modules.

![Image of Basic Display Module]

**Figure 17 Basic Display Module**

This display module is used to interact with the virtual environment stored at in the world module. It is responsible for maintaining consistency between itself and the server and in the case of the user module, displaying the virtual world. It also provides a means of relaying user events back to the world module. The display module runs completely independently of the world module, which prevents a slow display device from affecting behavior performance.

Connection Protocol

A display module connects to a World Module by sending a request to the world module’s connection manager. Once connected the display module sends any geometry associated with the Display Module. In the case of an agent, this would be all agent appearances. In the case of a user, this would be a user avatar. After the geometry has been transferred, it is inserted in the world scene graph stored on the world module. The world module then transfers the entire world
scene graph to the display module. This includes all user avatars, agent appearance, and static object representation.

Once a scene graph has been transmitted to the Display Module, the connection between the behavior and display module is maintained. It is through this connection that any update from the world module is relayed to the Viewer. This connection is bi-directional so any action produced by the display module will be reflected back to the world module. Updates to and from the world module are of the form:

<table>
<thead>
<tr>
<th>Viewer ID</th>
<th>Node ID</th>
<th>Data</th>
</tr>
</thead>
</table>

As every node stored in the scene graph of the world module has a unique id, an update needs only to address nodes that have changed. Updates from the display module are similar in nature.

Display Modules have two variants, which allow for agents, and users to inhabit the virtual environment stored in the world module. These variants are an agent module and a user module. Both extend the Display Module.

The agent module, show in Figure 18 consists of one or more agents, and all of the components of the basic display module. Each agent’s appearance is inserted into the world module mirror. The user of this agent cluster allows for a simple agent to be grouped together. A more complex agent, for example a tutoring agent might be the only agent in its particular display unit since it would require a heavy amount of processing on the client machine.

![Figure 18 Agent Module](image)

The user module is shown in Figure 19. The user module is the only module which displays the virtual world. It consists of all of the components of the basic display module with the interface being a user interface. The other major difference is that the user module also contains a display module. The user module is tailored for the display device. For example, a CAVE user module would be written in Performer while a Windows NT version would use Cosmo. Another version could use Java 3D.
Figure 19 User Module

Programming Module

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 of programmers. However, the novice programmer is the primary user group for this system. While not accustomed to textual programming, almost all computer users are experienced 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 Coccola 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 will be used here.

The programming environment is a programming interface for agents. Since an agent control unit is composed of several layers, the programming support should also be built around these layers. Since each layer is responsible for composing 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 travels through its control unit.

Programming support is in three areas:

- Motor Skills
- Reactive Rules
- Local and Joint Plans

Motor Skills
Motor skills are specified by example. A user can assign a DOF to transform node in the agent’s scene graph. Determining which transform nodes in the agent’s appearance are available may not be apparent if the joints are not visible. Accessing transform node via the scene graphs is difficult as they do not follow any specific structure and often display more information than is needed. To solve this, the programming system can transform the agent geometry into a stick figure in which each transformation node is located at a joint, effectively transforming the appearance into a virtual skeletal system. Once visible the user can assign DOFs to these joints.

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 creature. The standardized coordinate system for an agent is forward/backwards assigned to the z-axis, up/down y-axis and left/right x-axis. This coordinate system is inserted into an agent’s appearance as a root transformation node.

Simple motor skills can be programmed by example. For instance in the figure below, 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 \textit{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 range if applicable.

It should be noted that Motor Skills are not limited to transformations of geometry. Motor Skill can also be video textures and audio clips. This allows, for instance, the use of sounds that can be assigned as a Motor Skill. These sounds can be loaded into the Motor Skill library for later use by the programming system.

\textbf{Reactive Rules}

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

\[ \{ \text{START Condition} \} \{ \text{RUNNING Condition} \} \{ \text{END Condition} \} \rightarrow \text{ACTION} \]
For example, in Figure 20 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 21), the agent turns until is can no longer detect an obstacle in its path. The agent is then move so that is past the obstacle. Once clear of the obstacle the agent is informed that this rule definition has been completed, Figure 22. The resulting rule is then translated into a PoB.
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. Each agent has its own local coordinate system. When the agent is instructed to rotate, it computes which appropriate motor commands to invoke. 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 precedence over other rules. Additionally, objects can be grouped so that a rule applies to the entire set of objects.

Once behavior rules have been specified, sequences of rules and goal directed behavior can be built by using plans. Plans allow for actions that are more complex and require deliberative action on the part of the agent.
Plans

Plans are compositions of rules for accomplishing goals. Plans are programmed in a similar manner to the system for rules. Plans, however, are more complex in that they are usually 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 Knowledge base. 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 plans can be composed of other plans. This allows very large plans to be composed by grouping smaller 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.

Joint plans can also be designed using two or more agents. In this model, each agent is shown which part of the overall plan it is responsible for, however each agent is aware of the entire plan. When the plan becomes enabled each participating agent elects which part of the plan it will perform, by using the communicative layer for coordination. Joint plans allow for coordination of groups of agents.

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 thing the user does is to import the geometry into the agent workshop. 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 creatures its basic motor skills. The creature is taught to move forward, turn, and jump. This is done in the manner described in the Motor Skills programming 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 programming module is an extension of the display module, however with several changes. Because of the level of interaction with the agent, shared memory architecture would better serve the Programming module. By using this design, the programming module can directly affect the agent data structures for programming purposes, preventing the need for a complex protocol between the programming module and the world module.

The use of the visual programming module should not be 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 should be available to support a range of authoring support. This allows users to move from an abstract but high-level VPL to an explicit specification. Since both a scripting language and an API will be developed as a result of creating the prototype, this support should not increase development time.

Sensors can be assigned to any part of the agent’s architecture. This is done in a manner similar to the DOF assignment for the Motor Skills. Sensors are supplied in a library, and the user can modify their operation by changing their parameters. Sensors will be supplied as a library, for the prototype users will not be able to develop their own sensors. This functionality can be reserved for future enhancements.

Everything created by the user can be stored in a library for future use. The user then would not have to create basic action for each agent, as they are available in the library. It has been noted [57] that it is often easier to modify existing code rather that start from scratch. This is in fact one of the basis for Object Oriented Programming. By applying an object-oriented design to each, new actions, plans, and goals could be constructed by deriving new actions from existing ones.

Another area which has not been address is the ability of agents to affect geometry of other objects in the environment. Since some of the behaviors are graphical rewrite 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.

Timeline

There are four areas of design and implementation that are required to build a workable system: the agent module, the programming module, the World module, and the Display Module variants. Each of these areas will require time to design and implement.

![Figure 23: Complete timeline for development of HAVEN prototype](image)

The overall timeline is shown in the figure below. The time allocated to each module is shown as a different colored bar. The tickmarks along the bottom horizontal axis measure weeks. The project
is designed in two phases. Each phase allows for design, implementation, and evaluation. The use of two phases allows an intermediate evaluation period to evaluate each module and the performance of the overall system. It also allows for a short time period to correct any major problems encountered with the system during the first evaluation.
The first phase is called Design Phase 1. In this phase the reactive agent architecture, world module, a visual module, and the programming module will be developed, designed, and implemented. In other words the basic infrastructure will be completed by the end of phase I.

Once designed and implemented there should be no reason to allocate any more time in their development. At the end of Design Phase 1 the system should be compete enough to allow creation of reactive creatures in the following evaluation phase.

The reactive agent architecture will be competed. As of now most of the reactive layer, Knowledge base and World interface has been implemented, as well as the VRML 2 parser. The Motor skill layer will be designed and incorporated in the control unit of the agent.

The world module needs to be completely designed and implemented. This includes the agent manager, the system clock, and the world scene graph. The world module side of the scene graph exporting routing will also be completed. The display module will also be completed, which includes receiving and display of the scene graph as well as update from the World module.

Agent Module – 4 weeks
- Complete Reactive Layer*
- Complete Knowledge Base*
- Complete Wif*
- Design and Implement Motor Skill System
- Complete VRML import functions for agent appearance

World Module – 4 weeks
- Create World Scene graph*
- Create System Clock
- Create Agent Manager
- Create Export of Scene Graph

Display Module – 4 weeks
- Protocol Development
- Import of scene graph
- Display of scene graph
Programming Module – 6 weeks

DOF assignment
Motor Skill assignment
Sensor Assignment
Reactive Rules

*Completed

Breakdown of tasks – Phase I

The programming module will also be designed and implemented. This includes support for motor control and reactive behavior programming. This module will take the most time since it requires the most design effort.

At the end of phase one enough of the system should exists to allow for the creation of agents, which have reactive behaviors. The test of this is will be the first evaluation.

Evaluation I

The next phase is called Evaluation Phase 1. Its purpose is to allow time to test and debug the reactive agents; it is programming interface, and the behavior and display modules. At this time, it should be possible to construct small autonomous creatures and populate and environment with them. If successful then the example system will be available to other users.

As a test of the capabilities of the system at this point in it development, several autonomous creatures will be built in the manner described. The system at this point should allow for the development of reactive agents. This creature can exhibit behaviors that have been seem before in a-life systems, however their development is much different that the traditional methods of hard coding behaviors. Several of these creatures will be developed with test the capabilities of the system. These creatures can include birds, which exhibit flocking behavior, herds of creatures, and creatures, which exhibit hunter prey behaviors.

For the first evaluation phase the following type of creatures will be developed:

• Flyer: bird-like creatures that can fly, take off and land, avoid obstacles and exhibit flocking behavior
• Cattle: land-based creatures which can run, walk, graze and avoid proximity to users.
• Orbiter: a creature which orbits a user by traveling with the user and maintaining both a minimum and maximum distance.

The development of these creatures will test the overall system. Any major complication encountered at this point can be adjusted before continuing on the next phase of development.
The following phase, called Design Phase II, will address the issues of deliberative behaviors, local and joint plans, and goals. The deliberative and communication layers of the agent will be designed and implemented, along with the upper layers of the Knowledge Base. By this phase, since most of the other modules have been complete more time can be spent on developing the deliberative programming support.

The agent programming will involve creating the two upper layers of the agent, the deliberative, and the Communicative. The more important work will be in developing the agent’s ability to devise plans in order to accomplish goals. The communication work will involve developing the Agents communication protocol, which will be based on KQML, and the protocol for executing joint plans.

Programming support will be in allowing for the construction of plans and the design of goals. This includes plans and goal specification. Once this design phase is completed, the system is ready for its final evaluation. This will demonstrate the systems abilities and provide a means of evaluating the success of the prototype. However before the final evaluation there is a one week period set aside to allow time to tweak the modules in preparation for the final evaluation.

**Breakdown of tasks – Phase II**

**Agent – 7 weeks**
- Deliberative Layer
- Communicative Layer
- Communication Protocol

**Programming – 8 weeks**
- Local Plan Programming
- Joint Plan Programming
- Goal Specification
- Plan computation

**Evaluation II – Final Evaluation**
The last phase is the final evaluation of the system. There is some time set aside for development of a few other features such as geometry manipulation as described above but by this point the system should be fully functional. By now creatures that can plan, accomplish goals, and demonstrate tasks can be developed. For example, the herd developed in the first evaluation might be controlled by a class of shepherding agents whose goal state is to prevent stray cattle.

The more interesting class of agent is an interactive agent, which responds to user interaction and can be used, in virtual environments as a tutor. The tutor will be able to demonstrate tasks, adjusting its position, gaze, and gestures to account for a user’s position and attention level.

As a means of testing this ability, a virtual tutor will be constructed. The tutor will be human in appearance and will instruct user on how to use HAVEN. The tutor will present the basic design of the system and instruct a user on how to construct virtual creatures. This demonstration will guide the user through the phases of importing the geometry, assigning of basic Motor Skills and reactive behaviors. The tutor will demonstrate the building of a reactive agent one of the ones from the first evaluation. The user will then be allowed to construct their own agent. These agents can then be set free in the tutoring environment and allowed to exist for the rest of the lecture.

The tutor will then demonstrate the more complex ideas of local plans, group plans, and goals. The tutor will then take the reactive creature previously developed and demonstrate how to assign plans and goals. It should be noted that the final evaluation not only allows for an investigation of the capabilities of the system but also its limitations. While it is believed that quite complex behavior can be developed by visual programming only there might exist a set of tasks for which visual programming is not the best means of programming. It might be better then to fall back to a scripting level or even the programming language level to develop a certain behavior. However, that would not mean a failure of the system since the overall goal is to provide a range of programming environments, which best suit, the task to be programmed.

The creation of the virtual tutor will demonstrate the capabilities of the system. If this can be developed, mostly visually then HAVEN will have accomplished its goal of providing a programming environment for programming behavior of agents.

**Conclusion**

The end result of this system allows for complex creatures to be developed with less effort than required by previous systems. HAVEN’s contribution is two fold: it demonstrates the capability of hybrid agents for modeling behavior in virtual reality systems. It moves beyond the abilities of previous reactive based architectures for behavior modeling. In addition, more importantly it provides tools necessary to develop these creatures. By taking advantage of the layered structure of the hybrid agent, it provides a clear hierarchy of tasks to create a wide range of creatures. The capabilities of these creatures range from simple reactive animal to virtual tutors capable of demonstrating complex tasks. It also provides a foundation for future extensions of the system to provide even more capabilities for agents and their programming.
Bibliography


2. Thalmann, D., Becheiraz, P. "A Behavioral Animation System for Autonomous Actors personified by Emotions" " pp. 22-29

3. Gent, R., Huber, D., Hayes-Roth, B. "Acting in Character" "


6. "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents" "


8. Conway, C. "Alice: Lessons Learned from Building a 3D System for Novices" "


16. Bratman, M. "Toward an architecture for resource-bounded agents" "CSLI-87-104" 1987
23. Brooks, R. "Intelligence without representation" "Artificial Intelligence" Volume 47 1991 pp. 139-159
29. Reilly, W., Bates, J. "Building Emotional Agents" "CMU-CS-91-143" 1992
34. Shoham, Y. "Agent-oriented Programming" "Artificial Intelligence" Volume 60(1) 1993 pp. 51-92
35. "Watch What I Do: Programming by Demonstration" "" 1993


38. Citrin, W., Repenning, A. "Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction" "IEEE Workshop on Visual Languages" 1993 pp. 77-82


45. Hasen, S. "Motion Control through communicating hierarchical state machines" "Proceedings of the Fifth Eurographics Workshop and Animation and Simulation" 1994 pp. 115-129


50. Tambe, M. "Intelligent Agents for Interactive Simulation Environments" "AI Magazine" Volume 16(1) 1995


58. Muller, J. "The Design of Intelligent Agents" "" 1996


65. Tokorol, M. "An Agent is an Individual That Has Consciousness" "Intelligent Agents II" 1996

66. Wooldridge, M. "Agents as a Rorschach Test: A Response to Franidin and Graesser" "Intelligent Agents II" 1996


68. Connah, D., Wavish, P. "Virtual actors that can perform scripts and improvise roles" "Autonomous Agents '97" 1997
69. Fels, S., Esser, S., Mase, K., Bruderlin, A. "Hierarchical Agent Interface for Animation" "SigGraph Course Notes" 1997


72. Fels, S., Mase, K., Esser, S., Bruderland, A. "Hierarchical Agent Interface for Animation" "IJCAI-97 Workshop on Animated Interface Agents" 1997 pp. 27-31

73. Mateas, M. "An OZ-Centric Review of Interactive Drama and Believable Agents" "CMU-CS-91-156" 1997


77. Elliot, C. "Hunting for the Holy Grail with emotionally intelligent virtual actors" "" 1997

78. Rickel, J., Johnson, W. "Integrating Pedagogical Agents into Virtual Environments" "Presence" Volume 7(6) 1998 pp. 523-545


92. Pandey, Rajeev K., Burnett, Margaret M. “Is it easier to Write Matrix Manipulation Programs Visually or Textually? An Emperical Study” IEEE Symposium on Visual Languages pp 344-351 1993

