Desktop Theater:

Automatic Generation of Expressive Animation

by

Steven Henry Strassmann

Bachelor of Science, Computer Science, MIT, 1984
Master of Science, MIT, 1986

Submitted to the Media Arts and Sciences Section, School of Architecture and Planning,

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

June, 1991

Author_____

Steven Strassmann
Media Arts and Sciences Section
May 3, 1991

Certified by

Marvin Minsky, Thesis Supervisor
Toshiba Professor of Media Arts and Sciences

Accepted by

Stephen A. Benton
Chairperson, Departmental Committee on Graduate Students

Desktop Theater:
Automatic Generation of Expressive Animation

by

Steven Henry Strassmann

Submitted to the Media Arts and Sciences Section, School of Architecture and Planning, on May 3, 1991, in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the Massachusetts Institute of Technology.

Abstract

This thesis describes a program which automatically generates animations expressing the emotions and intentions of the simulated actors. The user interactively directs a scene by typing English sentences, such as "John politely offers the book to Mary."

A knowledge base describing props, actors, behaviors, and other knowledge about acting is organized in a flexible, dynamic representation that allows natural language parsing and generation, common sense reasoning, and inference and retrieval of animation control parameters. A video tape of examples (duration: 6 minutes, 40 seconds) accompanies this thesis.

Thesis Supervisor: Prof. Marvin Minsky

Title: Toshiba Professor of Media Arts and Sciences

This work was supported by an equipment grant from Hewlett-Packard Corporation, and research grants from Toshiba and the Nichidai Fund.

Table of Contents

1 *Introduction*

Animation programs today are essentially CAD (computer-aided drafting) systems. The information they store and manipulate are polygons and splines, positions in space and trajectories over time. They're extremely good at representing geometric forms and landscapes that you can fly past at dizzying speeds.

But when most people think of "animation" they think of stories with characters who seem to laugh, cry, and live. Each motion has a *meaning*, each object has a *purpose*. A gun and a book differ in more significant ways than merely shape and color – they have very different *roles* to play in a scene.

Unless your software acknowledges this, there is a limit to how much help it can give you. Trying to make animations with a CAD system is a little like trying to write a novel with MacPaint – the user must think not just about the creative decisions regarding the scene, but also a host of petty details.

Perhaps the most difficult part of creating animation is making it *expressive*. It's easy to make a spinning cube, but how do you make an angry one? How can you convey to an audience a character's feelings, intentions, or attitudes?

In this thesis, I will describe a system I have built called Divadlo[1] which attempts to represent and help automate this process.

Divadlo's basic output unit is a *scene*, an animated sequence. The system parses a small paragraph of simple English describing a scene, and after a while, renders a sequence of frames which are stored in RAM for realtime playback, at variable speeds up to 30 frames/second, in full color.

Perhaps it is best illustrated by a few examples. The system can, for example, parse the following two scenes (as they are shown, as plain English sentences) and automatically generate corresponding animations:

Scene 1: *"In this scene, John gets angry. He offers the book to Mary rudely, but she refuses it. He slams it down on the table. Mary cries while John glares at her."*

Scene 2: *"In this scene John gets sad. He's holding the book in his left hand. He offers it to Mary kindly, but she politely refuses it. He puts it down on the table. Mary shrugs while John looks at her."*

Althought the basic plot in both scenes are similar, the mood is completely different. This is reflected in the resulting animations: the characters' behaviors are functionally equivalent, but they are differentiated by timing, mannerisms, and even choice of action performed.

A videotape which accompanies this thesis demonstrates how the user can interactively set up each of these scenes, as well as the animations that result.

1. *Divadlo* is the Czech word for "theater." In looking for a name for this system, I turned to the Czech language for two reasons. For one, it's the source of another influential word: *robot*, derived from the Czech word for "worker." Another good reason is that my father is a native speaker.

FIGURE 1. John and Mary, at the beginning of the scene.

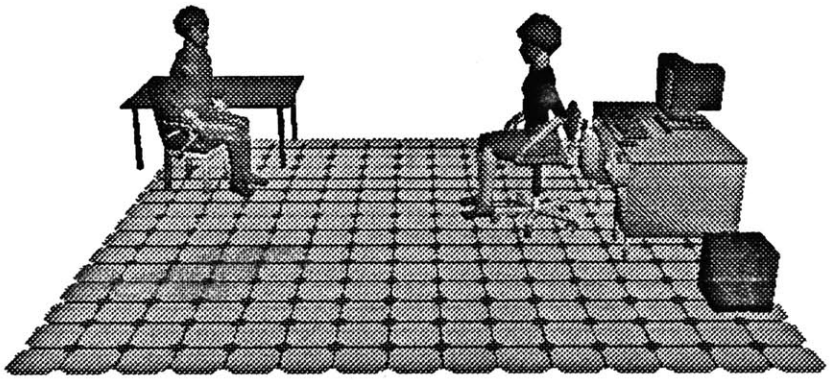


FIGURE 2. John, getting angry in the first scene, slams the book down, while clenching his right fist.

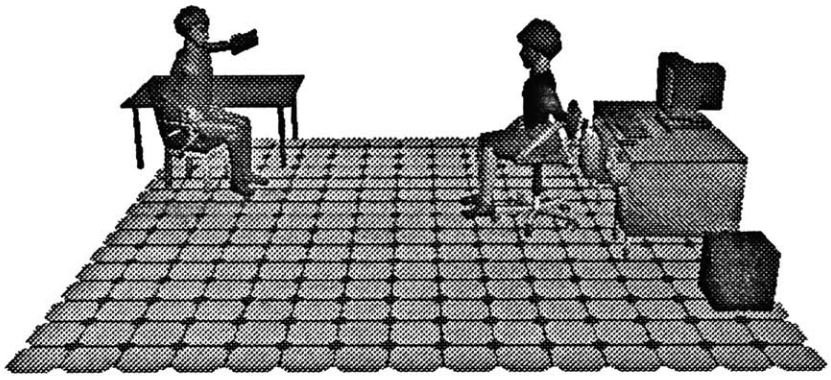
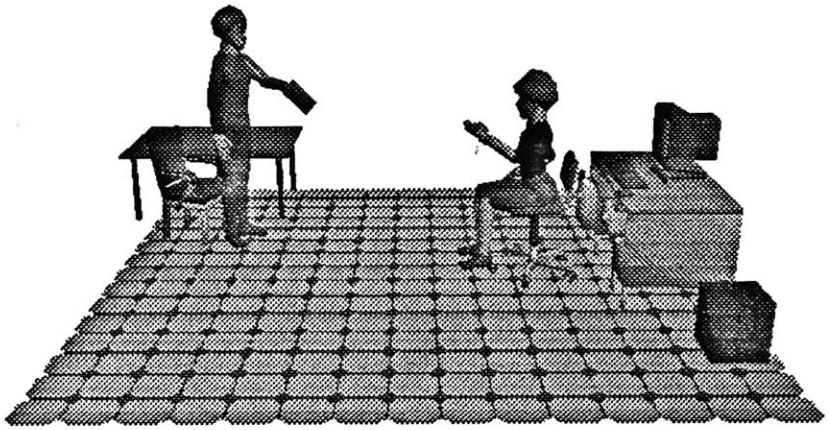


FIGURE 3. John slowly (sadly) starts to place the book on the table. The increased civility level causes John to stand up when offering the book, and Mary to indicate refusal with a slight wave of thehand.



This system differs from other animation systems in four major ways:

- It automatically adds emotions and expressiveness.
- It maintains and uses knowledge about the scene, both of a symbolic (qualitative) and a numerical (quantitative) nature.
- Behaviors are stored in an extensible library, analogous to "clip art."
- The user describes a scene in English, rather than a special-purpose programming language.

In the next few sections, I'll explain why I think adding these components makes an animation system better.

Why add expressiveness automatically?

Emotions and expression are why people like animation in the first place. Without emotive characters, animation is boring; just "flying logos".

Some people are lucky enough to possess the required combination of talent, perseverance, and opportunity to make animations the traditional way, by guiding inanimate objects in just the right way. But the rest of us, if we should want to create our own, need a tool to help us out.

There are many ways for an "angry" character to show anger, or for an "interested" character to express interest. Current animation software requires the animator to take direct control over every nuance of every gesture, like a puppeteer. This makes animation far too tedious for most people to generate productively.

If the expressive aspects of animation are automated, then we can concentrate on calling for displays of emotion, rather than on implementing them. We become *directors*, directing *actors* who attempt to interpret and carry out our commands without bothering us with the details.

For this reason, I chose to coin the term "desktop theater" to convey the notion that, like desktop publishing, an expressive animation of some dramatic performance is the sort of thing that nontechnical users should be able to create on their personal computers.

Why use both symbolic and numerical knowledge?

All animation software needs, at some point, to know the exact spatial position and orientation of each object in a scene, in order to render it. This is stored *numerically*, commonly as 6 or more numbers specifying a displacement and rotation.

However, it is also useful to represent information *symbolically,* so that the system can reason about the scene. Relationships like "John fears Mary" or "This chair's occupant is Sue" can be represented as symbolic links between representational units.

For example, there's a big difference between grasping an object and merely laying one's hand alongside it. If the hand (or the object) is moved, it may be extremely difficult for a system to deduce the consequences from knowing only the geometry of the situation. A stylized character may not even have sufficiently detailed fingers to make this distinction by geometric data alone. But if you note this symbolically, like adding a notation like `(HOLDING BOOK.1)` to the hand in question, the situation is unambiguous. The knowledge about a character's goals and mental state indicates *why* a character's hand is in a certain location.

Symbolic constraints are often far easier for the user to specify than concrete numerical values. For example, qualitative specifications like , "Walter is sitting on the sofa, watching the TV" or "the silverware is laid out on the table" imply the equivalent of lots of tedious numerical information, which would otherwise be entered directly with some pointing device or even as typed ASCII values.

Symbolic notation of relationships among objects allows the user to dispense altogether with some kinds of tedious input. For example, by associating a chair with a desk, the user doesn't have to guide it to a specific location on the floor; placing the desk suffices to place the chair in a reasonable location.

Why store behaviors in a library?

An animator's skill is a precious resource, so why not recycle it? "Clip art" is very useful for people who need static images but cannot afford to draw or commission original art for their application. In a similar vein, I think "clip behavior" can open up the animation bottleneck. Traditional animators may see little need for reusable snippets of behavior; it's almost painful to suggest making a film like *Bambi* by re-using some of the animals from *Snow White.* But an interactive com-

puter simulation might very well make good use of a library of animals and their behaviors.

The 100-minute film *Who Killed Roger Rabbit* (which wasn't even fully animated) represents an investment of roughly 200 man-years of animator effort[1]. Imagine having a studio like Disney or Warner Brothers investing this kind of effort into a library of reusable resources, say, as commercially available CD-ROM's. It would contain sufficient information to portray a variety of characters performing a variety of actions: walking, running, sneezing, kissing, etc.

In what format, then, should these be stored? Clearly not as sequences of cels, that wouldn't be flexible enough. Motor skills (page 26) capture the essence of an action in such a way that they can be adapted to a range of characters and situations. To make use of these motor skills, you also need to represent knowledge about *how* to invoke them and *when*.

This thesis suggests a few ideas for how to do this. By maintaining symbolic descriptions of the status and motivations for the characters in the scene, the system can reason about which actions to perform, and in which order. Where the motor skills are parameterized to afford a range of possible performances, this same knowledge can be used to decide appropriate values for these parameters.

Why use natural language?

The two most common methods for specifying animations are

- A programming language, in which the user writes code describing the scene. Perhaps a few dozen special-purpose animation languages have been written, such as ZGRASS[2], ASAS[3], BBOP[4], and CINEMIRA[5].

1. Ed Jones of LucasFilm, personal communication
2. ZGRASS, Tom DeFanti, Univ. of Illinois, 1976
3. ASAS; Craig Reynolds, Triple I, 1982
4. BBOP, Stern, Williams, and Sturman, NYIT, 1983

- Gestural input, which is used by just about everything that isn't a programming language. A pointing device, such as a mouse or tablet, is used to intractively position the objects and indicate their paths.

Natural language commands are more user-friendly than programming languages, especially for amateur users. Whereas programming languages are fine for professional animators, I would like to make animation accessible to children and casual users.

Parsing relatively simple, concrete English sentences has been demonstrated for years, if not decades, but there have been few attempts to drive animation from them. This is possibly due to the almost completely disjoint communities of researchers in computer graphics and computational linguistics. The only examples I am aware of are described at the end of this chapter.

Natural language is in many ways more expressive than gestural input. For one thing, the vocabulary of text is much larger than that of mouse-based gesture. With natural language, you can express abstract concepts like "A supports B" or "A is grasping B" much more easily than you can with a mouse.

Lastly, animation is not just for animators. Just as computer graphics researchers stand to gain a lot from ideas in AI fields such as natural language and story understanding, AI researchers stand to gain a lot by having some way of visualizing their work. One of the things I discovered is that getting a computer to understand some English statements is a lot easier when those statements are about specific objects that actually exist (or at least exist in detailed simulation).

Integrating it all

In order to achieve the four objectives I set for myself above, one theme became dominant quickly enough: this work would be primarily a matter of integrating several diverse elements:

5. CINEMIRA, N. Magnenat-Thalmann and D. Thalmann, Univ. of Montreal, 1984

- Natural Language
- Knowledge Representation
- Planning
- Robotics
- Computer Graphics

Each of these fields are sufficiently advanced for the demands of the system I envisioned. In a sense, I had a vision of a stereotypical "science-fiction" computer system, where you just tell the computer to do what you want and it does it. A system that could just combine the existing state-of-the-art from each of these five fields would serve as a blueprint for the sorts of intelligent applications that we've been reading about in pulp magazines since the 50's.

So why hadn't this fusion already been done? The papers in these fields hold the answer plainly enough: crossing the lines between disciplines is a rare thing indeed. Each field has become highly introverted, making progress by pushing the limits of its representations and algorithms. Yet it is almost impossible to find papers which describe systems significantly integrating work in more than two of these fields, let alone all of them at once.

It's easy to see why this happens. The mechanism of academic research is optimized for the specialist. By concentrating on one narrow area, one can quickly become an eminent authority. Things get sticky when you have to accommodate a broad range of knowledge outside your area of expertise. Thus, the expert on planning describes actions as logical propositions like "`move(A,B)`", whereas the expert on robotics builds an arm whose motions are accurate to a few thousandths of an inch. Very few systems operate on both levels simultaneously[1]. Another example: a sentence like "The more Aladdin rubbed the lamp, the brighter it got," indicates both a visual phenomenon and a causal relationship. There are several linguistic

1. A notable counterexample is Paul Fishwick's work [Fishwick] on reasoning and simulation at multiple levels.

systems which can extract this interpretation from such a sentence, and a few graphical systems which can animate it, but none which can do both.

This thesis is an attempt to cross some of those barriers and to integrate some of the promising technologies in each of these fields. It has a decent parser for handling natural language. It has a frame-based knowledge representation, which allows the other components to share objects and their knowledge about them in a single, unified scheme. It performs a modest degree of adaptive planning. It controls the motion of complex human figures using robotics techniques. And it generates smooth-shaded full-color animations.

Each component benefits from its connection to the whole. The linguistic problem of determining reference, for example, is so much easier when there's a specific context in which to interpret the meaning of a phrase like "the other one." Planning takes on a whole new perspective when you have goals expressed interactively by a director, instead of being set up as abstract propositions. Robotics becomes more interesting too: When your robots are more than arms and turtles, your behaviors can be more interesting than just "move block" and "go forward." Computer animation becomes more than just an exercise in automatic tweening – the characters become *characters*.

The components

So, although each component theoretically was a "solved problem," from the point of view of each corresponding field, the problem is that they don't fit together that easily. I found that each part had to be re-thought in terms of its interface to the other parts, and in many cases, had to be re-written from scratch. In the end, these are the parts I managed to put together:

- *Natural language*

  Using a parser mechanism from Tim Finin of U.Penn [Finin] (also described in [Gangel]), I had to re-write my own grammar with 94 rules (see Appendix). The lexicon contains a 650-word vocabulary (1474 words if you count morphological variations like plurals and verb tenses). The existing code for lexicon entries had to be rewritten to accommodate the units used for knowledge rep-

resentation, and the tokenizer had to be rewritten to accommodate niceties like contractions, mixed case words (like proper nouns), compound tokens like "New York", etc.

Responses to user queries like "where is John?" are synthesized by formulating the answer first as one of several classes of replies ("John is downstage", an absolute location, as opposed to "John is sitting on the chair", a relative location). Then the internal notation for *sitting on the chair* must be converted into a corresponding English phrase, perhaps "sitting on the red chair" if color distinguishes it from other chairs in the room.

Once input phrases are parsed, they must still be matched to referents in the scene. I had to build a subsystem for resolving pronouns and noun phrases like "the upstage chair" to particular objects or groups of objects. Verbs, once parsed, must be interpreted into specific actions to be placed on a character's agenda, constrained by other adverbs and prepositional phrases derived from the parse. Thus, the hard part of natural language is not so much identifying the grammatical roles of each word in a sentence, but in the computing the interpretation of that sentence in the *context* of the world being simulated.

- *Knowledge Representation*

  In order to draw what you're talking about, it's best to use one representation scheme for both the parser and renderer. I used a RLL (Representation Language Language) called Arlotje [Haase], but had to re-write major portions in order to support notions of objects and their properties, as well as actions and their scripts.

  Units do not correspond neatly to lexicon entries in the parser; I had to write lots of hacks to allow for handling of synonyms, morphological variations, and especially notions of properties such as "light blue," "very quickly," or "stage left center." Provisions had to be made to represent both numerical and symbolic information, such as the specific RGB triplet for light blue, or the test for when an object at coordinates XYZ is considered to be located at stage left center.

Units also do not correspond neatly to objects as used by the renderer. Figures like people, or even a swivel chair, have dozens of components, such as fingers, casters that pivot and roll, and little anonymous parts. These are automatically loaded and collectively manipulated when referring to the chair, but the system must also allow for individual reference to and manipulation of these parts.

The structural decomposition of such shapes, and attributes of same (such as color and rotation) correspond to the vagaries of the modelling process, and do not correspond well to the "common sense" anatomy that we casually use. For example, there is no one polyhedron in a swivel chair that can represent the "upholstery".

- *Planning*

  Although dozens, perhaps hundreds of AI theses on planning have been written, none are so domain-independent that one can casually borrow one and apply it to an application such as this one. I chose instead to write my own mechanism from scratch, incorporating as many ideas as I could from the plethora of other systems.

  My system doesn't solve complex plans so much as it compiles concrete plans from high-level ones, based on script-based rules [Schank] and optimization rules that tweak the plans when triggered. I included a mechanism for deferring choices when insufficient data is available.

  Unlike many planners, the input and output in this system are not symbolic tokens like "`move(A,B)`", and the system's state is not a long list of predicates like "`on(C, TABLE)`". The characters' state is a rich network of units representing posture, mood, and location, with both qualitative and quantitative values. A planned action involves the selection of values for several parameters, some of which may be chosen purely for aesthetic reasons in order to indicate the character's mood or style of acting.

- *Robotics*

  By and large, most robotic research is devoted either to disembodied arms, wheeled mobile turtles, or torsoless legs that walk and/or run. There's not much in the way of papers addressing the issue of how to represent a humanoid character with both arms and legs that would engage in typical domestic activities that humans perform.

For one thing, notations for arm joints and leg joints (not to mention fingers) are often different, and I found I had to accommodate both. Different algorithms are appropriate for different kinds of gestures and motions, so I had to adopt a pluralistic notation to allow free mixing of key-frame, inverse kinematic, and other control methods.

Rather than using straight keyframing, in which the beginning, middle, and ending poses are stored in advance, I used "adaptive" keyframes which accommodate other control methods by letting portions of the body and portions of the sequence to be overridden by other motor actions deemed to have higher priority.

I developed a tool I call "partial poses" which allow various parts of a figure to be managed independently. This allows, for example, any of several hand gestures to optionally override a default hand pose (typically an open palm) suggested by any other processes which might control the upper body and arm.

- *Computer graphics*

  I was lucky enough to be able to use existing rendering software [Chen] without having to write my own, but unfortunately it had no mechanism for interactive control by another program except as a library of C routines. It turned out to be a substantial amount of work to get the Lisp side of the system to control the C side reliably, efficiently, and interactively. Much of the information, such as the positions and colors of all the objects, are necessarily represented redundantly on both sides of this "great wall," a scheme which reduces both the efficiency and reliability of the system.

  In order to facilitate rapid prototyping of motor skills, I wrote a C program which stores about 100 medium-resolution frames of animation in RAM for rapid playback, in a sort of "tapeless preview" mode. The final output of animations can be recorded in full detail on tape.

Implementation note

Divadlo is implemented on an HP 9000/835 workstation with a Turbo SRX graphics accelerator. It is written almost entirely in Lucid Common Lisp 4.0, except for the renderer and a few other graphics routines written in C. User interaction is primarily textual, as English sentences are typed in and English responses are returned in a Lisp listener shell. This runs inside a GnuEmacs window on an X Windows display. Several other types of windows, created using CLM, CLX, and Motif (see page 59), are used to graphically display the sentence parses, the internal state of the planner, and to edit and inspect objects in the knowledge base and the 3d rendering environment.

Parsing the user's input text is almost instantaneous, but rendering the resulting animation may take around 20-30 minutes, and is limited by the workstation's image memory to about 100 frames, depending on resolution. This is ample to preview behaviors more or less interactively. Divadlo also supports recording scenes to a Betacam video tape recorder, allowing high-quality images and arbitrarily long scenes at the expense of a longer turn-around time.

All rendering is performed by a sub-process running Rendermatic [Chen] written in C. In fast mode, it takes advantage of the workstation's hardware, rendering most scenes in about 4 seconds per frame, including the Lisp-C communication overhead. In "pretty" mode, it renders images with better quality (including texture maps and shadows) at a somewhat slower pace, which varies according to scene complexity.

Previous work

SHRDLU [Winograd '72] was a classic AI system that demonstrated that a machine could understand natural language and execute simple tasks: stacking blocks and pyramids. Some simple 2d animation was generated of the blocks and the hand that moved them. The internal simulation was used to generate output and and responses to user queries in natural language. This system hasn't been

duplicated in nearly two decades; this alone was a large inspiration for my pursuing study of a modern version.

ANI [Kahn '79] was the first system to automatically generate animations from story descriptions. These were specified as Lisp code rather than natural language, and output was limited to four simple 2d shapes (triangle, square, etc.). Kahn explicitly represented the mental states of the characters, and implemented a mechanism for making aesthetic choices based on collecting suggestions from a variety of sources. This idea formed the basis of the choice and refinement mechanism I used in this thesis.

SDAS [Takashima '87] automatically generated an animation from a children's story expressed in natural language, using modules for story understanding, stage direction, and action generation. My system does not attempt to interpret stories to the extent that SDAS does; its input is more concrete. However, my system's scope is much broader, with more complex representations for 3d objects, their properties, and actions.

[Badler et al. '90] describe a system which generates animation from natural language instructions, termed narrated animated simulations. They present a generalized model for simulation and motion generation. Their notion of "strength-guided" motion demonstrates the ability to express fatigued and energetic motion, and they have demonstrated that simulated figures can operate controls on a control panel under verbal command. Their goals are to model movement about a complex environment such as the space shuttle. Their focus is on studying and designing ergonomic task plans for astronauts, however, whereas this thesis is more concerned with expressive animation of fiction.

2 *Assumptions about Actions*

"When I use a word," Humpty Dumpty said
in a rather scornful tone, "it means just what I
choose it to mean - neither more nor less."
- Alice Through the Looking Glass

The meanings and control of behavior is a large, tangled issue and if we want to get anywhere, we're just going to have to make some simplifying assumptions. This section is my attempt to describe what I mean by action and behavior, and to explain why I chose the representations I did.

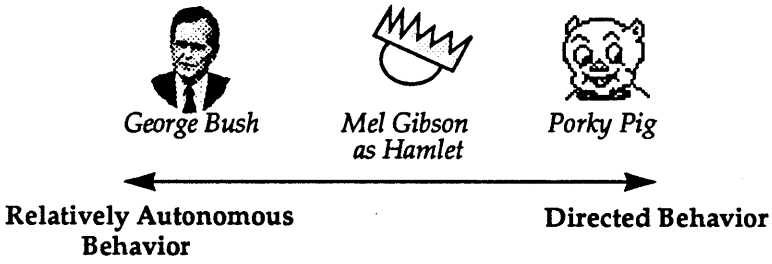
The word "actor" has been used in some contexts to mean a small portion of the brain (or other computer system) that performs some function, and this may be a source of confusion for some readers. I will, however, use "actor" in a slightly more traditional sense, namely, to refer to a complete human being, or a simulation of one.

Still, when I talk about an "actor", I tend to drift rather carelessly between the real world of flesh and blood people and the imaginary world of fictional characters. This is because I'm talking about characteristics they all have in common – they all *behave*, and what I'm trying to do is capture these behaviors in a computer-based simulation. I hope it is clear that the actors in this thesis are an attempt (primitive as it may be) to simulate certain aspects of this behavior, so I would like to reserve the right to draw my examples freely from both real and fictional domains.

To help clarify what kind of behavior I am addressing, let me bring up the notion of classifying actors by "degree of autonomy". For different kinds of actors, there is a continuum, from relatively autonomous behavior on one hand, to behavior that is more subject to external direction or control. This notion can perhaps best be illustrated by a few data points.

- What living beings do as they go about their lives — a sort of improvisational reaction to the world as they perceive it.

- What actors in a movie or theatrical production do — acting out a script, with some limited freedom to interpret a relatively fixed set of instructions.
- What animated cartoon characters do — where every nuance of motion is explicitly controlled by an animator according to his or her artistic judgement.



In the context of this thesis, I would like to make it clear that I choose to focus on behavior which lies somewhere near the middle of this scale. In my personal theory of acting, storytelling, or what-have-you, the director is omniscient and omnipotent – she has a pretty good idea of exactly what should happen in each scene. The script has already been written, everyone has their lines to say and their motions to go through. For the purposes of this discussion, however, I will not make any further distinctions between the scriptwriter and director, and refer henceforth only to the director as serving both roles in a single entity. This avoids dealing with the thorny problem of creation and interpretation of scripts at a different level than that which concerns us here.

By doing this, I am making a distinction between the responsibilities of the director on one hand, and the actor on the other. This is a blatantly reductionist move; an attempt to simplify the problem by dividing it into more manageable pieces. The director has primary responsibility for choosing what actions are worth performing, and roughly when they will be per-

formed. The actor, on the other hand, has some small leeway to apply his own interpretation to his performance as commanded by the director.

I think this model will also serve well when applied to simulations which lie closer to the free will side of the autonomy spectrum. Suppose you have actors at liberty to improvise behavior at will. Effectively, you're giving each actor their own director to carry around with them. There is still a separation between the decision-making and performance processes, it's just that the decision-maker is no longer in complete control of the entire world, and will occasionally be forced to accommodate the actions of other actors into its plans. This "individual director" is still making creative decisions about which actions to take, and the "individual actor" is still trying to perform them as best as it can.

Simulating the actors, not the director

Given this distinction, it is now time to say that this thesis does not attempt in any way to explain what goes on in the mind of a director or playwright – the agents I am simulating are just the actors.

To define the scope of the project, the actors are not given high-level goals, whose achievement remains a puzzle to be solved, such as "win this chess game." More than enough AI research elsewhere addresses this sort of problem-solving.

Instead, the domain I would like to address is that netherworld between the output of a successful problem-solver and the input to conventional animation software. There is a large gap between these two kinds of systems, because they arose from very different and separate sub-cultures of that larger field called computer science.

By and large, programs which do problem-solving and make plans are considered successful when their output is a symbolic description of actions to be performed. This, however, is still a little too abstract to pass along to an animation system as is. In order to generate a scene, animation software needs to be supplied with a lot more details, such as positions, velocities, or forces to be applied to the objects in the scene.

This system attempts to bridge that gap by taking a scene from a high-level symbolic description all the way through to a rendered animated sequence. A director is responsible for the creative definition of the scene, but needs actors to reify that definition. The actor then, whether real or simulated, is the tool, the output mechanism for a director's creativity.

I suppose I could have chosen to write a director program to direct the actors in this thesis – and perhaps some day soon I will. But for now, instead of an artificial problem-solver, I prefer to address the needs of a human problem-solver – one who has a pretty good notion of what kind of animation he'd like to see, but is too lazy to actually animate it on a frame-by-frame basis.

Another important consideration is that although there probably aren't enough theories about artificial directors yet, I feel the time is ripe to make an obedient actor. Perhaps if more actors become available, it would be much easier to make and test theories about directing. Another advantage, is that I consider directing to be the "fun" part of making an animation. For those of us humans who have plenty of ideas for animations we'd like to see, coming up with these ideas is not yet a tedious process that needs to be automated.

No surprises

Another important way in which I simplified the scope of this thesis is to rule out the possibility of taking an actor by surprise. In real life, you could try to pick up an object, and you'd be surprised by all the sorts of things that could go wrong. There's a chance that it might be glued to the table, or slip out of your hands, or burn you, or anything else might cause your attempt to fail.

This is not the case for these actors. Whenever the director commands them to do something that is properly within their behavior repertoire, they simply do it. This is a non-trivial decision on my part, but I feel justified in

making it for two reasons. The first is that it makes things much easier, because everything's predictable.

More seriously, removing the element of surprise is consistent with the assumption I made above, namely, that I'm sticking to the middle range of the autonomy spectrum. If you want an actor to act surprised when a monster suddenly leaps out from behind a door, you'll have to put it into the script. The director holds the responsibility for deciding if and when any surprises will happen, and will instruct the actors as to which reactions are appropriate.

The reader may point out that this removes the desirable quality of serendipity from the resulting animations, but I would counter this by suggesting that the serendipity can always be added back in, by changing one's model of how a director creates scripts. However, this thesis is not about the creative process of a director, it is about how actors peform the output of that process.

Actions are units of behavior

Now that we have separated execution from decision-making, we need to talk about what exactly is being "executed" or "decided on".

The next step is to subscribe to a sort of molecular theory of behavior. From this perspective, the things that actors do are built up out of atomic, indivisible elements called *motor skills*. The molecules, if you will, of behavior are combinations of these elements – groups of motor skills that are performed in conjunction. Throughout this thesis, I will use the word *action* to refer to these behavioral units, whether made of a single atomic motor skill, or a compound sequence.

For example, a high–level action might be "putting the book on the table." This can be broken up into the components "picking up the book," "walking to the table," and "putting the book down." Each of these can in turn be broken up in to smaller and smaller units – "picking up the book" consists of moving the hand to an appropriate location, closing the fingers, and so forth.

If an actor posesses a repertoire of basic motor skills, such as *sit, stand, walk, reach, grasp, place,* etc., then these define a stopping point for this decomposition process. All behavior can be reduced to the peformance of these skills in the right order, at the right time. We break up the general problem of "how to act" into the smaller problems of

- Creating a repertoire of motor skills
- Figuring out how and when to invoke them

An objection – the fluidity of natural action

Right away, I know that this gives rise to a serious objection – behavior is not atomic, and attempts to formulate it as such will cause one to overlook its fundamentally fluid and analog nature. Indeed, the very sorts of behaviors we admire most (on a visual level, at any rate), in dance, sports, and cartoon animation, seem to derive their appeal from graceful and continuous motion.

One way to think of an agent's behavior is to examine, say, a typical actor on a standard daytime TV soap opera, to take a random example. His movements can be analyzed in terms of a set of time-varying signals, each indicating rotational and translational displacements of the bones in that actor's skeletal structure, as measured at dozens, if not hundreds, of locations. These displacements are caused by the forces applied to that structure by various muscle groups in the body acting in concert and opposition, as well as external forces imposed by gravity, impinging objects, the constraints of any garments or restrictive devices that may be worn, etc.

It is this, the most detailed description of motion, that any system actually requires in order to generate animation of the human form. Ultimately, the rendering system needs to know the specific location and orientation of every part of the body, at every frame, in order to draw it in its place to give the illusion of motion. For animators working in traditional media like cels or clay, these positions are draw out or sculpted explicitly on every frame.

When the control is this close to the signal, no generalizations or simplifications will hold true. The animator is free to violate any general rules on the slightest whim, as long as it satisfies any other aspect of his creative license.

Short of complete reproduction of an animator's mind, we will necessarily have to make some simplifying assumptions in order to codify behavior into a more tractable form. In accordance with the molecular analogy mentioned above, I have chosen denote the smallest meaningful groups of these signals into *motor skills*. For example, by treating the act of "sitting down" as a motor skill, it becomes a unit of behavior that can be performed whenever the need arises. By doing this, we can segregate the *signal processing* aspects of behavior into manageable units. We can then attempt to reason about these units at a symbolic level.

Unlike real atoms, however, not all instances of these skills are necessarily identical. It is convenient to add some range of flexibility to these skills by parameterizing them. Thus, if we are clever about how we implement these motor skills, a character may be able to "sit down" quickly or slowly, on a tall chair or a short one, etc. depending on the values of a few parameters which are provided to the "sit down" motor skill.

This rephrases the original question of how to control behavior into a few new questions:

- how are motor skills performed?
- how are they combined into sequences?
- which ones are worth defining?

How are motor skills performed? — We will pragmatically define a *motor skill* as being whatever it takes to perform a single, atomic action. How these skills are implemented depend on what tools are available for describing motion. Many different techniques are currently available, and new ones are actively being developed. These are described in more detail on page 42, but a key point here is to assume that a *motor skill* is a black box, a functional abstraction. No further analysis is necessary – all that one needs to know about each motor skill in order to use it properly is that, if it is properly invoked, it will perform as expected.

This begs the question of what "properly invoked" means. A motor skill is properly invoked when

- certain essential *preconditions* must hold true at the skill's commencement
- any required *parameters* must be provided, to control or modify its execution

The *preconditions* of a skill impose restrictions on the context for invoking that skill. For example, one cannot walk if one is lying down – standing upright and having both feet on the floor a necessary precondition of the *walk* motor skill.

For those motor skills which are applicable in a range of circumstances, the *parameters* of a skill indicate how the skill is to be performed. For example, before one can pick up an object (i.e. execute a *pick-up* motor skill), one must specify the values of parameters such as "which hand to use", "how to grasp the object", "how fast to move the hand", etc.

The matter of which parameters are available to control a skill depends on how that skill is implemented. A simple implementation of a skill may provide no flexibility in its execution – it's exactly the same, every time you invoke it. No parameters are necessary, since there's nothing to change. A more sophisticated implementation provides many parameters, corresponding to all the different ways in which the skill can be performed.

So, the important tasks of the animation system is then to make sure that all the necessary preconditions are in fact met, before each motor skill is invoked, and to provide it with whatever parameters it may need in order to operate. Both of these tasks are the strong points of this thesis – by automating them as much as possible, a large burden is removed from the user.

How are they combined into sequences? — We will presume that the user does not want to invoke each motor skill manually – although this might arguably save a certain degree of effort in the animation process, I think we can

do better. The goal of this thesis is to automatically generate a sequence of motor skills from a simpler, more abstract description.

In essence, this process is not significantly different than the process of compiling a high-level computer language down to machine code. In both cases, each high-level statement is used by the compiler to either constrain its operation, or generate some more explicitly detailed sequence of function calls. Some of these function calls may not have been explicitly called for, but are implicitly necessary in order for the resulting sequence to be valid.

Chapter 4, page 35 describes in more detail what these abstract descriptions look like, and chapter 7, page 94, describes how these sequences are generated.

What action units are worth defining? — The purpose of chunking is to make the job of description as easy as possible. For me, it seems like a good starting point is to take as chunks those actions which correspond to single English verbs. This is not because of any significant or mysterious link between linguistics and the spinal cord, rather, it's because verbs were invented because people found it convenient to give names for actions which form clear, distinct concepts.

This one-to-one correspondence of words to actions comes up lacking rather quickly, however. It's clear that there are many ways to act out a given word-concept, and there are important kinds of actions which don't really correspond to any one particular word-concept. I will attempt to address these issues later, but for now I'd like to stick to the notion that each verb in English corresponds to an action.

At the lowest level, each motor skill corresponds to each distinct algorithm available for controlling actors. For example, if you have one highly expressive `WALK` algorithm, perhaps it can control a walking figure over a broad variety of possible situations. By supplying appropriate values for its parameters, it could conceivably govern the walking of men and women, fat and thin people, tall and short people, fast and slow walking, etc. In this case, a single `WALK` motor skill suffices.

An alternative view might be to break up this functionality into a collection of less flexible, more specialized motor skills. In this case, `WALK` would necessarily be a

high-level action, whose script would be responsible for selecting the appropriate motor skill to invoke. This process is described in chapter 7, page 94.

3 *Expression*

Perhaps the most interesting aspect of Divadlo is the automatic insertion of expressive qualities into the animation. Qualities like ANGER, IMPATIENCE, or DELIGHT can be manifest by invoking one or more *expression traits*. For example, IMPATIENCE can be alternatively expressed by tapping one's toes, drumming one's fingers, looking repeatedly at various clocks, or yawning. Minsky describes a theoretical basis [Minsky, p. 313] for the correspondence between gesture and emotional expressiveness.

Suppose the user tells the system, "Rick waits on the sofa impatiently." This creates a new action unit[1], WAIT.1, which is an instance of the WAIT action class. Because the adverb "impatiently" was used, the MOOD resource of WAIT.1 will contain the mood IMPATIENT, which is the adjectival form derived from the adverbial unit IMPATIENTLY, which was one of the properties of the parsed sentence. When WAIT.1 is considered for refinement, depending on how its script was written, it will be refined in a manner which indicates that somehow the notion of *impatience* has to be expressed.

The MOOD resource for an action like WAIT.1 doesn't necessarily derive its information wholly from the parsed sentence which invokes it. Suggestions for the MOOD of an action are also gathered from the AGENT of that action, and the parents of that action (in case it is the SUBTASK-OF another action). For example, Rick might be in an IMPATIENT mood today, so he does everything impatiently. Or he might be waiting on the sofa as a SUBTASK-OF another skill like ACCEPT-

1. See chapter 4, page 35, for more about units and knowledge representation.

DIPLOMA, which itself is being done impatiently. In this case, the mood is inherited from ancestors in the subtask hierarchy.

When it actually comes down to expressing the notion that Rick is being impatient, there are many ways of storing this knowledge in Divadlo. There is, of course, no one best way, so whichever way the user decides to use serves as an expression of the user's theory about what emotion and expression is.

Divadlo (and its predecessor ANI [Kahn]) make all their decisions based on *policies* provided by the user (or the builder of the animation knowledge base), in the form of the scripts and optimization rules used in the process of refinement. By choosing different scripts and rules, you can change from one school of acting or filmmaking to another. This makes Desktop Theater a malleable tool for exploring the effects of changing your rules to see what kinds of animations get generated.

For example, a certain character may have some stereotyped traits, like Mr. Spock on *Star Trek*, who always raises his eyebrow. Another character may have an aversion to using the same trait more than once. Given 10 different ways to express ANGER, such a character will use a different one in each of 10 different scenes. The difference between these two types of actors are as much a part of their capabilities as the walking and reaching skills they can perform. Of course, if no other information is available, one of the ten methods can be picked by random choice.

Certain kinds of knowledge are best expressed as what *not* to do: for example, "never turn your back to the camera," "don't go near Rick's desk," or "don't bump into the furniture." These sorts of rules are known as *suppressors*, and can be expressed either in the global optimization rules or in the functions used to compute resources for actions.

For example, the three rules mentioned in the previous paragraph are all relevant to the PATH resource of GO actions, which govern walking from one place to another. Before any value can be determined for the PATH of a given act of walking, these constraints may offer their advice to the path-

planner. As general, global rules, they may be always enforced. But if only certain characters are constrained differently than others, these rules can be stored on specifically those characters.

For example, if only JOHN was forbidden to walk close to downstage (i.e., near the audience), then JOHN's BEHAVIOR-SUPPRESSORS slot would contain a rule stating that. Then, whenever a WALK action needed to compute a PATH over which JOHN could walk, it would necessarily take into account that rule, and any other relevant rules that might be stored there.

Divadlo currently makes its expression decisions in both its scripts and its motor skills. In the example videos which accompany this thesis, JOHN offers a book to MARY on two occasions, once RUDELY, and once POLITELY.

The rude offer is a simple thrust forward of the hand, whereas the polite offer consists of the sequence of JOHN standing up, and then moving his hand forward. This is because two different cases are applied in evaluating the script for OFFER.

The rude actions are quick and abrupt, whereas the polite actions are slower and more smooth. Although the same motor skills are used in both cases, the difference in resource values cause the rude actions to use linear instead of spline interpolation for their motions, and to be performed in a shorter period of time.

Optimization rules afford some extra augmentation of the scene with expressive behavior. For example, as JOHN gets mad in the first scene, he clenches his free hand into a fist in parallel with the other actions going on. By contrast, in the second scene, as he gets more sad, he simply slows down, giving the impression of his disappointment.

4 *Knowledge Representation*

An essential part of a system that does some smart things is its knowledge representation scheme. For that, I chose ARLOtje [Haase], for its richness and flexibility, and because it is actually supported for (and by) a small group of users. ARLOtje is a Representation Language Language (RLL) implemented in Lisp with many of the features of CYC [Lenat], expressly designed for the creation of representation systems like that used in Divadlo. Although the concepts presented in this section may be familiar to those in the AI community [Winston], many of these ideas have yet to make a strong impact on programmers in the graphics community.

Units in ARLOtje resemble the objects of object-oriented programming languages [Goldberg], or more closely, frames [Minsky]. They are organized into class hierarchies, and each unit is distinguished by the values stored in slots it possesses or inherits from other units.

The important thing about a unit is that its slots store everything the system could ever need to know about it. For example, a unit representing the notion of a chair needs to store *grammatical* information for the parser ("chair" is a noun), *geometric* information for the renderer (the shape and color of a chair), and *functional* information for reasoning about plans (chairs are for sitting on). By simply using one representation scheme for all these different components, it greatly simplifies the task of sharing knowledge among them.

Objects

When Divadlo starts up, there are about 700 units which represent various objects, processes, and properties. (A complete list is given on page 69). Many more are created on the fly as scenes are generated. Some typical unit categories which represent objects include

- **actors** — Each character is itself made from a tree of parts. A specific actor can be constructed easily from an available body structure, and customized with various features such as skin or hair color, clothing style, etc.

  The current system has two stock human body structures, one male and one female, each with 44 movable sub-parts represented by rigid polyhedra.
- **props** — Divadlo currently has a stock of about 30 props, including furniture, office equipment, plumbing fixtures, and appliances, all of which come in a variety of decorator colors.
- **cameras** — A camera is a configurable object, much as an actor, with its own complete set of behaviors (dolly, pan, zoom, etc.)
- **lights** — Lights can be created and modified in the usual way. For quick previews, a simple gouraud-shaded lighting model is used, but final output may include shadows, texture maps, and other state-of-the-art rendering goodies.

It is important to note that each unit has slots with all relevant knowledge about the thing that it represents. This is crucial, because a desktop theater system doesn't simply render scenes, it also reasons about them. For example, a unit named `CHAIR.3` can be defined with the following code:

```
(define-unit chair.3
   (instance-of desk-chair)
   (upholstery-color red)
   (associated-furniture desk.3))
```

This simple expression actually creates a unit with a wealth of information, containing dozens of slots created by inheritance and implicit (demon) methods. These slots provide information about the object's ontology, structure, and attributes.

ontology - CHAIR.3 is an instance of class DESK-CHAIR, so it inherits all sorts of DESK-CHAIR-like knowledge. It is a candidate for matching when Divadlo parses phrases like "Sit on the chair." Chairs in general are not known to be edible, so CHAIR.3 is not even considered as a candidate for matching commands like "Colin should eat a snack". As a desk chair, however, it inherits a default association with some instance of class DESK, which in turn causes it to possess such properties as a default stage placement relative to that desk.

structure - Its physical structure is represented by a hierarchy of objects representing its component polyhedra, and the various renderer-specific attributes (such as position and color) associated with each. (This is described in more detail in chapter 5, page 47.)

The unit representing a class stores this structural data in its 3D-STRUCTURE-PROTOTYPE slot. As each instance of an object is created, this prototype is copied, and the copy becomes the value of that object's 3D-STRUCTURE slot. Thus it can provide polygonal data to the renderer on demand, as well as geometric data (such as the height of a chair's seat) to behavior routines (such as SITTING) as needed.

To facilitate building up a database of useful structures, Divadlo has a data-conversion tool for importing articulated 3D shapes from a variety of formats, including Symbolics' S-Geometry, Paracomp's Swivel 3D, Pixar's Renderman (RIB file), and Ohio State University's OSU format.

attributes - If CHAIR.3 has an UPHOLSTERY-COLOR of RED, this is useful in several ways. This datum is combined with information from the 3D-STRUCTURE slot to provide the renderer with appropriately-colored polygons. Additionally, when parsing English commands, a phrase like "Sit on the red chair" can be used to disambiguate CHAIR.3 from other chairs in the room.

ARLOtje provides powerful inference and search mechanisms which greatly ease the task of describing and retrieving knowledge, especially handy for things like multiple inheritance and exceptions. It's relatively

easy to describe such representational subtleties as "the default COLOR for a CHAIR can be derived from its UPHOLSTERY-COLOR", or "if a thing has a COLOR, any part of that object has that color, by default."

Since DESK-CHAIRs are adjustable, CHAIR.3 contains slots for values of properties like seat rotation and back-support angle, as well as the chair's placement in the scene relative to the other objects. These are also used by the 3D-STRUCTURE slot in responding to geometry-based queries from the renderer or other subsystems.

Other important attributes arise in the course of developing a scene, including properties such as ownership (e.g. declaring (CHAIR.3 OWNED-BY RICK)), weight (to be used by dynamics-driven behaviors [which, alas, are not implemented]), or even narrative-related properties like price or date of purchase.

Properties

Attributes such as COLOR, MOOD, and OWNED-BY are in fact represented as units themselves. They correspond to adjectives as far as the natural language system is concerned. Each adjective belongs to a category called a *property*, which that adjective carries a specific value for. For example, the adjectives TEAL, MAUVE, and NAVY-BLUE are all adjectives which indicate the COLOR property. TEENY, LARGE, and JUMBO indicate the SIZE property, et cetera. Other properties include HANDEDNESS (for distinguishing left- and right-handed people), MOOD (for distinguishing happy and sad people), etc.

A certain amount of special-purpose code is requred for each property. For example, talking about the COLOR of a person is just too different from talking about their MOOD to be handled by a single general-purpose mechanism. The solution is to have some general-purpose code which handles the common aspects like declaring which adjectives are in which properties, and some special-purpose code which does things like testing an object to see if it matches a description, or updating that object when a value is asserted.

An example of special-purpose code is the handling of the COLOR slot of an object. One could imagine a far more elaborate system, but for convenience, an object's color is stored as a list of several keyword-value pairs. For example:

```
(get-value 'ken 'color)
⇒((hair blond)(skin suntan)
   (pants dark-denim)(shirt purple)
   (shoes sneaker-white))

(get-value 'desk-chair.1 'color)
⇒((upholstery dark-red)(casters black)
   (frame steel))
```

The mechanics of actually applying a specific color to a specific portion of an object are rather messy, so special code must be written specific to each category of object. This fortunately hides the messy details from the user, allowing you to simply say "Ken's hair is blond" without having to mess with selecting polyhedrons and applying RGB color values to them. You can also refer to "the guy with the purple shirt", which in some cases (as in real life) might be more convenient or natural than having to remember the character's name.

Some properties are sufficiently specified by a simple symbolic value, such as FLAVOR being TASTY or YUCKY. Others properties involve both a symbolic value (like COLOR being PINK) and a numerical value (like PINK being [1, .53, .93], using RGB notation). Wherever possible, the system tries to infer and maintain both symbolic and numerical representations for values, because both are useful. For example, the phrase "the pink flamingo" would still match a flamingo even if its color were [1, .6, .9], because colors are considered to match if their RGB coordinates are close, without necessarily matching exactly.

Properties are modified by an assertion mechanism which attempts to maintain consistency in the database. When a new value of any property is asserted, the old value may have to be retracted, or other side-effects may occur. For example, when you assert that Ken's shirt is PINK, you have to

replace the old value of PURPLE, and update the color of the corresponding objects in the renderer by downloading the appropriate RGB values.

The mechanism for consistency-maintenance is necessarily more complex for some properties than others. The MOOD property, for example, is a list of clauses indicating a person's mood. In this simple model, you can be HAPPY and CRAZY, but not HAPPY and ANGRY at the same time. Each mood unit has a slot listing its opposites, so that when you assert a specific mood, the opposite moods are automatically retracted.

The POSITION of an object is handled in a similar way. An object may have an absolute position, such as (AT DOWNSTAGE LEFT) or a relative one, like (ON-TOP-OF DESK.1) or (IN-FRONT-OF SUSAN). These are translated into specific numerical values by a suite of routines, and in many cases, a symbolic description can be inferred from specific numerical values. For example, the relationship ON-TOP-OF can be asserted by comparing the bounding boxes of the two objects, and placing one so its bounding box is centered over the bounding box of the other. The inverse of this is simply the test of whether the centroid of one is located over the centroid of the other, allowing for a certain degree of tolerance.

Although these relations work pretty well for simple objects, exceptions can be written whenever more specific knowledge is available or required. For example, in order to place a NeXT workstation ON-TOP-OF a desk, a special routine is used instead. The bounding box of the monitor, keyboard and mouse are placed according to the default ON-TOP-OF code, and the CPU box is placed BESIDE the desk, at floor level.

An object's orientation is maintained by its FACING slot. It's very similar to the POSITION slot, except the value (the *target*) is usually either another object (as in "facing the desk") or an absolute stage direction (as in "facing the audience").

An object's POSTURE indicates which of several possible configuations it could be in. Although I was primarily concerned with human figures, any articulated object can have various POSTUREs, for example, a refrigerator may have a FREEZER-OPEN posture and a FREEZER-CLOSED posture.

Postures don't necessarily specify a figure's complete posture; this allows several postures to be combined. For example, a person can be SITTING and CRYING at the same time. Each posture definition includes the notion of *hard* and *soft* resources, which describe the parts of the body involved in attaining that posture. HOLDING something in the left hand uses one's left hand as a *hard* resource. Two postures which have the same hard resources are mutually exclusive; this is a simplifying assumption which prevents one from using the left hand for both holding an object and making some gesture like a "peace" sign. Asserting a new posture which has a hard conflict with any current posture causes that posture to be retracted for that figure. For example, STANDING and SITTING both use the legs as hard resources, so asserting that JOHN is STANDING causes him to no longer be SITTING.

The *soft* resources for a posture are those which are modified if none of the figure's other postures have hard resources which take priority. For example, SITTING uses the arms as soft resources. By default, when you sit down, your arms are placed at rest at your side in a comfortable position. But if you are SITTING *and* HOLDING an object, the latter posture's hard resource causes it to govern that arm's position, overriding the default offered by the SITTING posture. The overlapping specification of postures is made possible by the systems ability to represent and combine partial poses (see page 54).

Adverbs

Adverbial qualities are stored in a manner similar to adjectives, as units which indicate specific values for a property. For example, the SPEED property contains units for QUICKLY and SLOWLY. In many cases, adverbs are linked to their related adjectives, so ANGRY and ANGRILY have mutual pointers to each other. This makes it easy to infer things like "a person who performs an action ANGRILY is probably ANGRY."

A special adverbial property is AMOUNT, which includes the adverbs SLIGHTLY, MORE, VERY, and NOT. This allows you to denote things like "the really blue sky," "he's a little upset", and "she is not sitting."

Actions

The basic unit of behavior in a scene is an *action*. Actions are units that represent processes that occur over time. The majority of actions are concerned with human behavior as performed by actors, but actions may also describe such things as camera moves or changes of state (like an object changing its color over time).

As described in chapter 2, page 22, actions can be one of two basic types:

- a high-level action – something that can be decomposed into simpler units
- a motor skill – a simple action (i.e. which can be performed without decomposition)

High level actions are units which cannot in themselves be performed by an agent. Each action is defined by its *script*, composed of one or more *cases*, which describes how to decompose the action into simpler components, depending on its context. The actual mechanism for processing this script is described in more detail in chapter 7, page 94.

Motor skills

A motor skill is implemented by a function which actually does the "dirty work" of positioning the joints and limbs of the actor at a given point in time.

To implement this function, a variety of techniques are used, appropriate to the skill. The simplest skills, like WAVE-HELLO, are implemented by interpolation of stored keyframes specifying joint angles on the actor's arm. Divadlo has a rich set of tools that supports a variety of keyframing techniques, including linear and spline interpolation, several different representations for joint rotation and dis-

placement, and merging of multiple, partially specified keyframes (e.g, allowing an actor to snap his fingers while waving hello).

Skills may also use more advanced techniques, like inverse kinematics [Paul], which is used by skills like REACH, which places the actor's hand at a given point in space.

Many special-purpose algorithms are available for canned skills such as walking or juggling, or controlling motion using dynamics. This is a rapidly progressing field, and each algorithm has its strengths and weaknesses. The philosophical position taken by this thesis is that they all should be accessible to the animator. What they all have in common is that they are parameterized to allow some flexibility of invocation; for example, most walking algorithms allow one to specify stride length, speed, bounciness of the pelvic region, etc.

These algorithms are not so flexible that they will accommodate all actors in all situations; certain preconditions must be met before invocation. For example, before a character can walk, both feet must be on the floor, etc.

Divadlo attempts to ensure that a motor skill is invoked with all necessary parameters provided, and all preconditions satisfied, so that the actual algorithm has nothing to worry about. The code for a motor skill describes how to move a character's body, assuming all the other problems have been taken care of.

The timeline

The timeline of the scene is a grouping of actions to be performed in the scene. An essential problem for any animation system is the representation of time. A good representation should allow one to

- Describe sequences of actions
- Describe actions which overlap and occur in parallel.

- Edit the timeline by adding, removing, and replacing actions.
- Determine which actions are occurring at any given time.
- Represent partial orderings of actions (such as "A comes before B") before specific start and end times are known.

Time representation, first attempt

My initial idea for representing the timeline was to have a bag of all the actions stored in the `SCENE-ACTIONS` slot of the scene. Each action had a `TIMING` slot, containing a list of constraints on when it could occur. For example, if A came before B, the timing slot of A would contain the form `(BEFORE B)`, and conversely, B's timing slot would contain `(AFTER A)`.

The work of [Allen] in particular gave me hope that a formal calculus of time representation could be supported by such a system. With a relatively small number of relationships, I wanted to represent all possible relationships among plans. In particular, I finally chose these primitives: `BEGINS-BEFORE`, `BEGINS-AFTER`, `ENDS-BEFORE`, `ENDS-AFTER`, `CO-BEGINS`, `CO-ENDS`. These, I hoped, would be sufficient to support all the temporal relationships I would need. As more information about the partial ordering of actions became known, I could add new constraints to the `TIMING` slot of the affected actions.

Ultimately, as decisions became concrete as to the duration and starting times of each action, there would be enough information to solve the constraints, and deduce the durations and starting times of other actions in accord with these constraints.

Although I still think this approach would ultimately be the right way to go about it, I was forced to abandon it since it became apparent that it was too ambitious. Were I to do this one part right, I would have no time left to attend to the many other aspects of the thesis that needed addressing. The common operations for refining the timeline involve the removal, insertion, and deletion of actions from the timeline, and each operation had to be performed while assuring the consistency of all remaining constraints. It became apparent that there was quite a large

number of possible combinations of constraint configurations, and each would have to be considered and addressed in turn. Second, detecting violations, or over-constrained timelines, would be a comparably challenging task. Third, extracting a particular concrete ordering from a network of constrained partial orderings would not be all that simple, either.

In the end, I abandoned this scheme reluctantly for another scheme. Although it is far less expressive than a general approach, it has the compelling advantage of simplicity, both for implementation and in the practical matter of understanding it while using it. Fortunately, it turns out to be good enough for my needs.

Time representation; what I used

I decided to borrow a notion from electrical circuit design, and consider a timeline to be similar to a ladder network of resistors.

A timeline is simply a tree of either parallel or serial blocks of actions. A serial block denotes actions to be performed in sequence, and is a list of the form `(SERIAL` *x1 x2 x3 ...*`)`. Likewise, a parallel grouping denotes actions to be performed in parallel, and is a list of the form `(PARALLEL` *x1 x2 x3 ...*`)`. Each element of a `SERIAL` block may be either an action or a `PARALLEL` block, and conversely, each element of a `PARALLEL` block may be either an action or another `SERIAL` block.

To illustrate this, here is the text of an example scene, and the timeline to which it corresponds.

"In this scene, John gets angry. He offers the book to Mary rudely, but she refuses it. He slams it down on the table. Mary cries while John glares at her."

```
(PARALLEL BECOME.1
          (SERIAL (PARALLEL OFFER.1 REFUSE.1)
                  SLAM.1
```

```
(PARALLEL CRY.1 GLARE.1)))
```

The action BECOME.1 corresponds to the process of John getting angry, described in the first sentence. The phrase "in this scene" indicates that this occurs over the course of the entire scene, so all other actions ought to occur in parallel to this.

Other phrases, like "while...", "as...", and "meanwhile..." indicate events which occur in parallel with other events. Events happen in serial order when they contain phrases like "then..." or "afterwards...". More sophisticated techniques could be probably used to extract temporal clues from phrases like "Between the two explosions, he managed to duck behind the sofa," but I did not attempt to go this far.

5 *3d Graphics*

Of all the things I learned when writing this system, perhaps the most astonishing was how difficult it was to make effective use of 3D graphics from a program written in Lisp.

In retrospect, there's absolutely no reason why it should have been such a pain, but all the same, it was. My development platform is a Hewlett-Packard workstation with excellent 3D graphics capabilities and a perfectly reasonable implementation of Common Lisp, made by Lucid.

HP supports graphics by means of a library of routines, collectively known as Starbase. These routines allow one to specify shapes, colors, positions, and other attributes of objects in 2D and 3D graphics in a manner portable across the entire HP product line. On my system, special-purpose hardware allows especially fast rendering.

Unfortunately, Starbase is only supported for developers using C, Pascal, and Fortran. As a Lisp user, I was just out of luck. Calls to HP were met with astonishment; most of their technical support people for graphics didn't know what Lisp was, let alone that HP sold and "supported" it. At one point, a technical support person asked me "if you're doing graphics, why would you need that AI stuff?"

Conversely, discussions with technical support people for Lisp at both Lucid and HP revealed that they had simply never heard of anyone writing any applications in Lisp that made any use of graphics. "I guess it's because

Lisp programs don't really need graphics," one person explained, in all seriousness.

One approach that they suggested was to make use of the foreign-function interface (FFI) provided in Lucid Lisp. This is a mechanism for declaring C functions and C data structures to allow them to be used and created by Lisp functions. For each C function, I would have to declare the argument list and returned values, and for each C data structure, declare its analogous structure in Lisp. All I would have to do, as was unanimously suggested by the helpful people at HP and Lucid, was to manually type in all the functions and structures for the entire Starbase library.

It didn't seem like a particularly fruitful endeavor, but at the time, it seemed like the only way. About a year later, as it turned out, some poor fellow at HP Labs ended up doing exactly this, and sent me a copy of his code. Well before he did so, however, I had abandoned this approach for a few other reasons.

Lucid's foreign function interface, (in release 3.1 at the time, as well as their current 4.0 release) is not a very well thought-out hack, with incomplete support of several essential features found in C, such as macros. The FFI documentation is poorly written, ambiguous, and has precious few examples. Any errors caused by, say, invoking a foreign function with slightly incorrect arguments, or accessing a foreign structure improperly, would result in a hopeless mess. The best one can hope for is an inscrutably non-descriptive message such as "Bus error". At worst, and not at all uncommonly, it will crash your Lisp process, leaving you with a 15-megabyte core file (which, if it doesn't fit on your disk, will merrily continue be written by unix anyway, producing a cacophony of "disk full" messages on your screen and starting a domino-like failure of any other applications you may be running which may expect to have a few bytes of spare disk space to use, such as your editor).

These problems would arise again in my search for a user interface, when I was attempting to display simple graphics on my console, which was running X Windows. But more about that later (on page 59).

Clearly, finding some alternative was pretty important. As it turned out, several people at the Visible Language Workshop at the Media Lab had been using Lisp

and Starbase graphics, so I went to see how they did it. It was achieved by running two processes, one in C, the other in Lisp. All communication took place by writing data out to a file on the disk, and then waiting for it to be read in by the other process.

It seemed like a terribly inefficient method, but the one I ended up using is not much better. Lucid Lisp provides a function called `RUN-PROGRAM` which spawns a child process by calling an arbitrary C program. The child is connected to the parent by means of a Unix pipe. To the parent (the Lisp process), this pipe is a standard bidirectional Lisp stream, on which common Lisp stream functions like `WRITE-CHAR` and `READ-CHAR` can operate. To the child (the C program), the pipe is the stdin and stdout (standard in and standard out) of conventional C programs. It is this technique which I ended up using for all my starbase applications.

One disadvantage is that this is slow; although lisp can push strings down the stream very quickly, it takes a long time for the C program on the other end to get around to reading them. In some very informal timing tests, I can execute a Lisp program that prints 100 no-op commands to this stream in about .09 seconds, yet it takes about 10 seconds for the C program on the other side to get around to reading them all in.

This technique also means, of course, that graphical data cannot be shared between Lisp and Starbase, which is definitely a disadvantage. The system has to maintain redundant copies of all relevant information, such as the position, orientation, color, and shape of objects, on both sides of the pipe. Much of the graphics code in this thesis is unfortunately occupied with maintaining the consistency of these two separate but equal copies.

For all but the simplest of communications, I eventually went back to a variation on the batch file technique used by the Visible Language Workshop. For example, when changing the posture of a single human figure, one must update the positions of the 44 polyhedra which make up its shape. The command for moving each polyhedron consists of the operator, the name of the polyhedron, and 16 floating point numbers forming the matrix which specifies a transform to be applied to that object. Communication is

necessarily ASCII-only, so a number such as "3.14159" must be spelled out as 7 ASCII bytes in order to be communicated, instead of a more efficient 16-bit or 32-bit floating-point notation.

Sending 44 such commands over the pipe provided by Lucid's RUN-PROGRAM would take about 30 seconds, an intolerably long time. Instead, the system batches up all 44 commands by writing them out to a temporary ASCII file, and then sends a single command to the C program containing the name of that file. The C program then reads in the commands and executes them. In the case of the 44-command file, this takes about 3 seconds.

3d

The situation was not entirely bleak, however. Although I seriously contemplated writing my own 3d rendering program in C, it turned out that I could adapt a pre-existing program for just about everything I needed. David Chen, a fellow grad student at the Media Lab, had been developing a rendering library in C called "Rendermatic" over the course of several years. He wrote a front-end interpreter to Rendermatic called "3d", which provides a read-eval-print loop to describe and render objects – just the sort of thing one can call from the lucid RUN-PROGRAM utility and pass ASCII commands to. I ended up using 3d for all the rendering in this thesis, and David kindly implemented or helped me implement the few routines that I needed to support my thesis.

3d has several extremely useful features, or at least allowed David or myself to add them easily enough as their need became apparent. It supports hierarchical objects, so that I can rotate a figure at the elbow and the attached parts (such as the hand and fingers) will be transformed accordingly. It can be toggled easily between "fast" and "pretty" mode. "Fast" mode takes advantage of the Starbase hardware to draw scenes in about 3-4 seconds, whereas "pretty" mode runs entirely in software to produce images with antialiasing, texture maps, shadows, and other niceties.

Shapes

In order to actually populate the world with objects, it became necessary to find or construct some polyhedral shapes that 3d could render. Most important of all was to get some decent human figures.

Early in my work, I built a simple human stick figure, using cylinders and boxes for the limbs, hands, and feet, and a sphere and truncated cone for the head and chest. For these, I represented the joints using Denavit-Hartenberg notation, since I had already written a good deal of code which made use of it, for example, in computing inverse kinematics for motion control.

However, I found a commercial product called Swivel 3D, which is a 3D modeling and rendering package for the Macintosh[1]. It came with a rather well-sculpted human figure, which looked like exactly what I needed. I contacted Young Harvill, who not only wrote the software, but also sculpted the human figures and many other shapes to boot, including some nice-looking pieces of furniture and office equipment. He was extremely helpful and nice, granting me permission to use the shapes, as well as sending the Lab a free copy of the advanced version, Swivel 3D Professional. Most of the objects you see in the illustrations of this thesis, are derived from Young's work, for which I am greatly indebted to him.

Making use of the shapes however, was not as straightforward as one would hope. It turned out that Swivel 3d stores its shapes in a baroque format unique to the Macintosh, and interpreting it or translating it to any other format was extremely difficult. Young sent me some fragments of source code for Swivel 3D, and along with a copy of *Inside Macintosh* and some cryptoanalysis, I was able to reverse-engineer the file format. Well, mostly. I still couldn't figure out how to extract some of the polygons.

1. Swivel 3D is made by Paracomp, 1725 Montgomery St., San Francisco, CA 94111

However, Swivel 3D Professional is able to export its objects in a format used by Pixar's Renderman known as a RIB file. These RIB files do not contain essential information about joint rotations that were contained in the Swivel files. But RIB files *do* contain the polygonal information I needed that I couldn't extract from the Swivel files. So, by saving every object out twice, once as a Swivel file and once as a RIB file, I was able to glean all the information I needed.

3d prefers its objects in a format developed at Ohio State University which we call, simply enough, "OSU" format. The only industrial-strength modeling software available at the Media Lab, however, is S-Geometry, from Symbolics. I used S-Geometry to model several other objects for my early work, and for other projects in the animation group, such as the film "Grinning Evil Death". It's not too surprising that S-geometry uses its own idiosyncratic format for shapes.

In the end, I wrote a general-purpose conversion utility that reads and writes objects in all of these formats: Swivel, RIB, OSU, and S-Geometry. This utility greatly simplifies the inevitable tedium of editing and transforming various objects into various incompatible formats.

The tiled floor, by the way, was generated by a quick hack I wrote in a fit of boredom at the prospect of having the usual checkerboard for a floor. Not that this tiling is any more exotic, but I was going for a 50's kitchen-tile sort of feel. This utility automatically generates tiled floors from a description of a single unit of the tiling pattern. The user specifies the shapes, colors, and positions of the polygons for one repetition unit, plus a few other parameters, and generates a complete wall-to-wall tiling as an OSU object.

Human figures.

Finally, I was able to have a figure that looked more like a human than a Calder sculpture. And not just one; Young had the foresight to provide both male and female figures. OK, so maybe she doesn't look like Jessica Rabbit, but at least the difference is detectable.

Each figure is made from 44 polyhedra, with a total of about 2000 polygons per figure. This is just about right; significantly more detail would make the rendering sluggish, but less detail would make the figures less recognizably human.

The hands are detailed down to the knuckles; the thumbs have 3 objects each and the other fingers have two. This allowed me to create about a half-dozen hand gestures.

Separate objects comprise the eyes, nose, and lips. Hair is always a problem for polygonal representations; as you can see from the illustrations we have to make do with a sort of football helmet.

A certain amount of touching up of the data was necessary. For example, the female figure had an extra polyhedron that needed to be merged in to make it isomorphic to the male figure. The various parts of the body were originally named "`object1`", "`object2`", etc., so I replaced this with an orderly scheme. Parts on the right side of the body begin with "`R`", those on the left with "`L`". The left upper arm is "`Luarm`", the left lower arm is "`Llarm`", and so on. Brevity turns out to be important, since reference to an object in 3d is in a manner analogous to Unix pathnames. Therefore, in order to refer to the second joint of John's left pinky, one uses "`/john/torso/Luarm/Llarm/Lhand/Lpink1/Lpink2`". A few other changes were made to the objects to clean up some cosmetic defects and make the orientation notations more convenient.

I discovered that judicious use of specifying colors for the various parts of the body could give the illusion of having them wear clothing. Baggy sweaters and knit scarfs are out of the question, but they could effectively be costumed in short or long-sleeved shirts, long pants or shorts (however, only the kind of shorts that cover your thighs all the way to the knees). Changing the hair and skin color also turned out to be very handy, and easy to do.

These sorts of cosmetic details may seem trivial, but I found them to be very important to give the illusion of having identifiable characters. At one

point, I populated a scene with about 12 characters, each with different costume and body coloration, and it looked like a room full of 12 individual characters, rather than 12 robot clones. If I had more time, I'd really like to investigate incorporating some more variations of body shapes. Having some children, obese people, etc. would really make the characters more interesting. It would also be important to demonstrate that the motor skills were written in a way that accommodated wherever possible the variations in a figure's body dimensions. Even more interesting would be to support non-humans, such as household pets and menacing aliens.

Pose notation

The humans aren't the only articulated objects in the shape library – as a matter of fact, almost all the objects are made of linked, flexible parts to some extent. The swivel chair swivels and tilts, the NeXT monitor tilts and pans, and the desk drawers can be opened or shut.

In order to bend a figure at the joints, the thorny matter of notation for these joints arises, especially for notating an arbitrary rotation in three-space. Many notations have been proposed and used by various systems, and they all have their advantages and disadvantages. Unfortunately, no one notation is clearly the best for all applications, so each system tends to be rather idiosyncratic about what they use.

Swivel 3d is no exception, and it uses a particularly idiosyncratic notation. In retrospect, perhaps it would have been a good idea for me to abandon it in favor of a more computationally tractable one. But I had to weigh the cost of translating all the objects in my growing shape library into yet another notation versus getting on with the work of animating them. I chose to use Swivel's notation, figuring it couldn't be all that bad.

For each joint in a Swivel object, consisting of a child attached to a parent, the following transforms are applied:

1. Scale the child by $[S_x\ S_y\ S_z]$
2. Rotate in the Z axis by z degrees ("roll")

3. Rotate in the X axis by x degrees ("pitch")
4. Rotate in the Y axis by y degrees ("yaw")
5. Translate the object by $[T_x\ T_y\ T_z]$

Thus, the transform at each joint is specified by 9 numbers: 3 for scale, three for rotation, and three for translation. Applying these 9 numbers to perform the above operations produce a homogeneous transform matrix describing the object's position and attitude in space. For the sake of rendering applications, like 3d, this comes out to be the matrix M_o:

$$\begin{bmatrix} S_x(\cos y\cos z + \sin x\sin y\sin z) & S_x\cos x\sin z & S_x(\sin x\cos y\sin z - \sin y\cos z) & 0 \\ S_y(\sin x\sin y\cos z - \cos y\sin z) & S_y\cos x\cos z & S_x(\sin y\sin z + \sin x\cos y\cos z) & 0 \\ S_z\cos x\sin y & -S_z\sin x & S_z\cos x\cos y & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} = M_o$$

M_o describes the position and orientation of a linked object O relative to the coordinate system of its parent.

Because of the redundant nature of the data representation between the Lisp and C sides of the rendering pipeline, there was the matter of exactly where the authoritative copies of the 9 numbers and the transform matrix would be stored, and who would be responsible for interconverting them. I went back and forth with several experimental configurations, but the final strategy I settled on was that the Lisp side would not only store all 9 numbers, but also compute the matrix and download it to C every time something was changed. To save time, I use the above closed-form equation, rather than applying 5 matrix multiplications. This turned out to be unnecessary, since it turns out that the time consumed by the math is dwarfed by the time consumed by the delay in transferring data over the Lisp/C pipeline.

For the sake of some code (like that used in computing inverse kinematic motion), it was also necessary to move some objects directly. What this means is that sometimes the position of an object is derived by operations

directly on an object's matrix and other objects in its hierarchy. While such algorithms give new values for the transform matrix of an object, they do not tell you what the proper values of the 9 input parameters are. In order to maintain consistency, I was forced to write an extremely ungraceful program which attempts to examine the values in the matrix, and extract the 9 input parameters from it.

This turns out to be impossible in the general case, unless several assumptions are made. This is because in many cases there are multiple, even infinitely many solutions for the three rotational parameters. Nevertheless, it seems to do an acceptable job for all the cases it has encountered so far.

Poses

I defined a new type of data structure, called a "pose", to store configuration information about an object. One way to think of a pose is a snapshot of all the joints in an object, or of some portion of an object. The object doesn't have to be a human figure; poses are equally applicable to any object in the scene, such as the swivel chair or the camera itself. In practice, I never got around to doing much experimentation with animating these other non-human objects, but they are fully supported.

A pose can be a partial description. For example, I created several hand gestures, including pointing, a peace sign, a clenched fist, etc. An important property of poses is that they can be merged, analogous to two images which can be matted together. For example, `STAND-POSE` may describe all the joints in a standing person, and `LFIST-POSE` may describe all the joints in a person's clenched left fist. By applying both poses to a person, they become standing, with their left fist clenched.

A pose is a data structure with several slots:

- `name` – The name of this pose
- `file` – The name of the file where this pose is stored
- `loaded?` – Whether loaded or not

- `object` – Object or object-class this pose applies to
- `base` – Operation to perform on object itself
- `hierarchy` – Operation to perform on children of object
- `frame` – A frame number, for use when keyframing
- `misc` – Miscellaneous other information

To save poses between sessions, they are stored in a special directory, one pose per file. An automated utility provides for loading them upon demand, by name. They're stored in a human-readable ASCII format instead of a more compact binary format, so that numeric values can be touched up with a text editor if desired. This turns out to be a very useful feature.

The `OBJECT` slot of a pose describes the class of objects to which this pose may be legitimately applied. For example, if the value is `:person`, the pose describes all the joints in an entire human body. Other values indicate that the pose describes a person's left or right hand, head position, a camera position, the tilt of a computer's monitor, etc.

The `BASE` slot of a pose describes the position of an object relative to its parent, and the `HIERARCHY` slot describes the positions of the descendants attached to that object.

Since several notations are commonly used, I decided to support them all, or at least as many as I could. Both the `BASE` and `HIERARCHY` slots have values which are lists. The `CAR` of the list is a keyword indicating the notation, and the `CDR` is the value, written in that notation. Some of the values are optional, in which case they default to whatever value already held by the joint in question.

Supported notations include

- `YPR` – This is a list of between 3 and 9 numbers, describing a position in "Yaw Pitch Roll", or Swivel's notation. The first three values are the yaw, pitch, and roll, as described above. The second three values are optional, and describe the translation components in X, Y, and Z notation. Since

joints tend to be purely rotary in nature, modification of translation values is not very common.[1] The third set of three numbers describe an optional Scaling to be applied.

- `XYZ` – This is a list of 3 numbers, indicating a translation in space. It's used most commonly for the BASE of an object, indicating a change in position without any change in orientation.
- `4x4` – This is a 4x4 matrix indicating a homogeneous transform to be applied to an object.
- `DH` – This is a list of between one and four values in Denavit-Hartenberg[2] notation, commonly used in robotics. The first value, theta, is required. The other three values, (`d alpha a`), are defaulted, since they do not usually change as an object moves.
- `QUATERNION` – This is a list of 4 numbers representing a quaternion, or a rotation in space, with no translation.
- `CAMERA` – This is a special notation describing the parameters for a camera. It's a list of alternating keyword-value pairs, saving various aspects of a camera position, such as position, view normal vector, view up vector, and view distance[3].

With the exception of `CAMERA` poses, functions are provided to convert between the various notations wherever possible. This is not always possible, since the representational spaces of the different notations do not overlap in many cases. The 4x4 matrix serves as a lowest common denominator in such cases, but unlike the others, it is not subject to interpolation. In practice, I was pretty much forced to stay with `YPR` and `CAMERA` notations throughout.

Two last slots of the `POSE` structure should be mentioned. The `FRAME` slot is useful when interpolating poses for keyframing (see below), and the `MISC` slot is useful for storing various temporary annotations and comments about the pose.

1. An exception is the SHRUG motor skill, where the characters's shoulders slide out of their sockets temporarily by a small amount.
2. George Lee, *Robot Arm Kinematics, Dynamics, and Control*, IEEE, December 1982, pp 62-80
3. See *Computer Graphics*, 2nd edition, Foley and VanDam et al, Addison-Wesley, 1990

One might make the observation that most poses which could be applied to a person's limbs and extremities (such as hand gestures) are in fact symmetrical, and do not need to be described redundantly for both left and right sides. It would be a good idea to write some code that could automatically apply, say, a "salute" pose to either the left or right arms. I did not have the chance to implement such a scheme, so therefore my pose library is perhaps twice as large as it needs to be.

Editing poses

In order to create, edit, and review the pose structures, I created an interactive utility called `CRISCO`.

`CRISCO` lets you select an object, bend and pose it to a new configuration, and make a snapshot of that pose for future use. In this way, it's easy to quickly build up a large library of poses that can be recalled on demand.

I have on my desk a small (12" tall) wooden model of a human figure, which I bought at a local art supply store. It has 15 ball joints with a reasonable range of movement. Visitors to my office find it very entertaining to bend this figure into a variety of entertaining and provocative poses, some of which I would capture using `CRISCO` when I had the chance.

Building a user interface

In attempting to write `CRISCO`, I encountered some difficulties not unlike those I encountered in trying to use Starbase. It turns out that until very recently as far as I could tell, nobody had really addressed the problem of building user interfaces from Lisp on a Unix workstation. On Symbolics Lisp machines, where the window system is written in Lisp, building up a user interface is doable, if somewhat baroque. But in Lucid Lisp on an HP workstation running X Windows, the only alternative suggested by Lucid

was their CommonWindows system, which was extremely uncommon, and far too slow to be of any use.

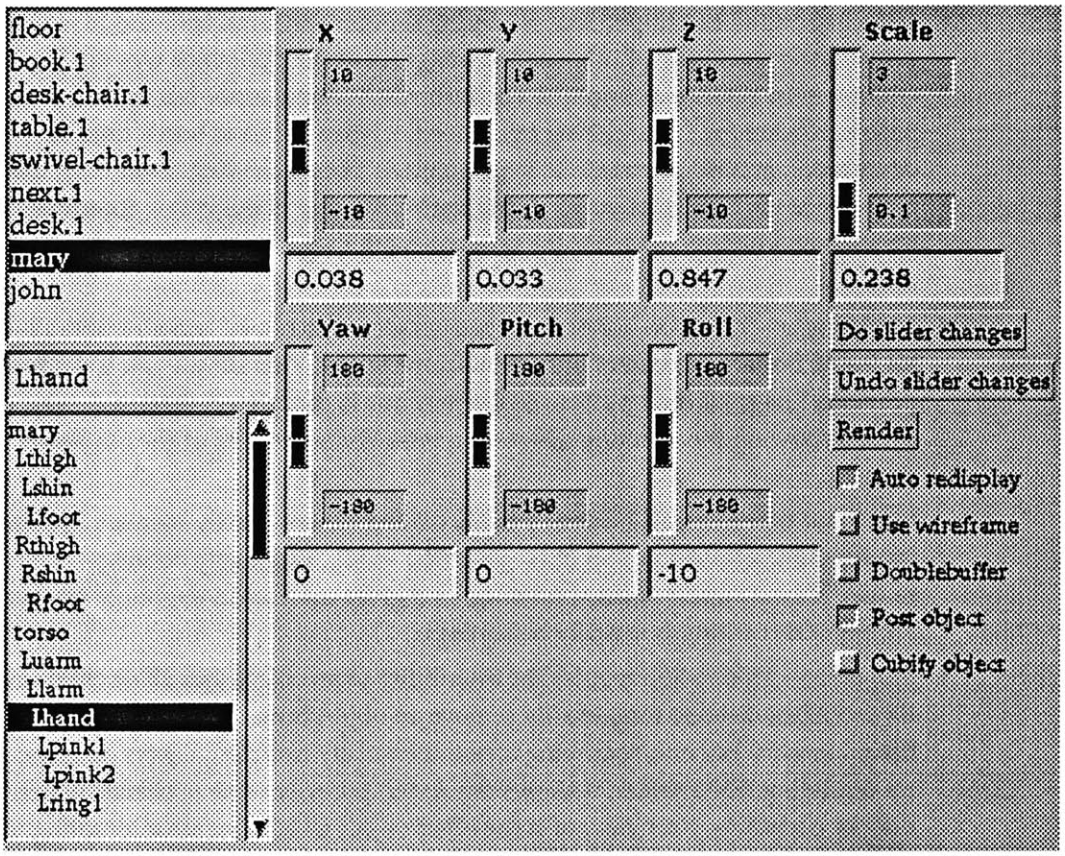
After an extensive survey of the field, it seemed that almost a dozen different groups were developing their own incompatible systems for user interfaces. The most promising included the Garnet project at CMU, the Winterp project at Hewlett-Packard, the CLUE system from TI, and the YY system from Aoyama University. Two systems bear special mention: the CLIM (CommonLisp Interface Manager) system, which is a highly ambitious project, now nearing imminent almost-final beta release as I write this. In the spirit of CLOS, it is an attempt to merge the best features of all previous systems into a standard whose applications would be portable across all window platforms, including X, Symbolics, Macintosh, and Microsoft Windows.

The other system, CLM (Common Lisp/Motif), is ultimately the system I decided to use, primarily because it alone had the desired properties of being available, functional, and relatively efficient. Written by Andreas Bäcker at GMD (Gesellschaft für Mathemathik und Datenverarbeitung mbH, or, The German National Research Center for Computer Science), it allows Commonlisp programmers mostly-full access to the Motif/X11 widget library. It had only been tested on Suns, but with a bit of work, I was able to get it to compile on the HP.

Windows are created and managed by sending messages over a network socket connection to a process running independently from one's Lisp process. This has the combined advantages of speed, and of giving each program a certain degree of immunity to fatal errors which might occur in the other.

Under CLM, each window, scroll-bar, button, and text field is a standard Motif widget. The most charitable thing I can say about Motif is that at least it works, once you learn how to overcome its many inadequacies, its inconsistent and mediocre design, and its almost worthless documentation. None of these problems, however, are Dr. Bäcker's fault – it is to his credit that he was able to make CLM function as well as it does.

Using CLM, I defined the interaction pane shown above. The user can select any object in the scene using the upper object menu, and select any component of that object from the lower menu. Once an object is selected, it is highlighted with a dis-



tinctively distasteful green color in the 3D window. The user may then move it around either by moving the sliders or by typing in new values for the various parameters. The "undo" command proved to be enormously useful in trial-and-error positioning of objects.

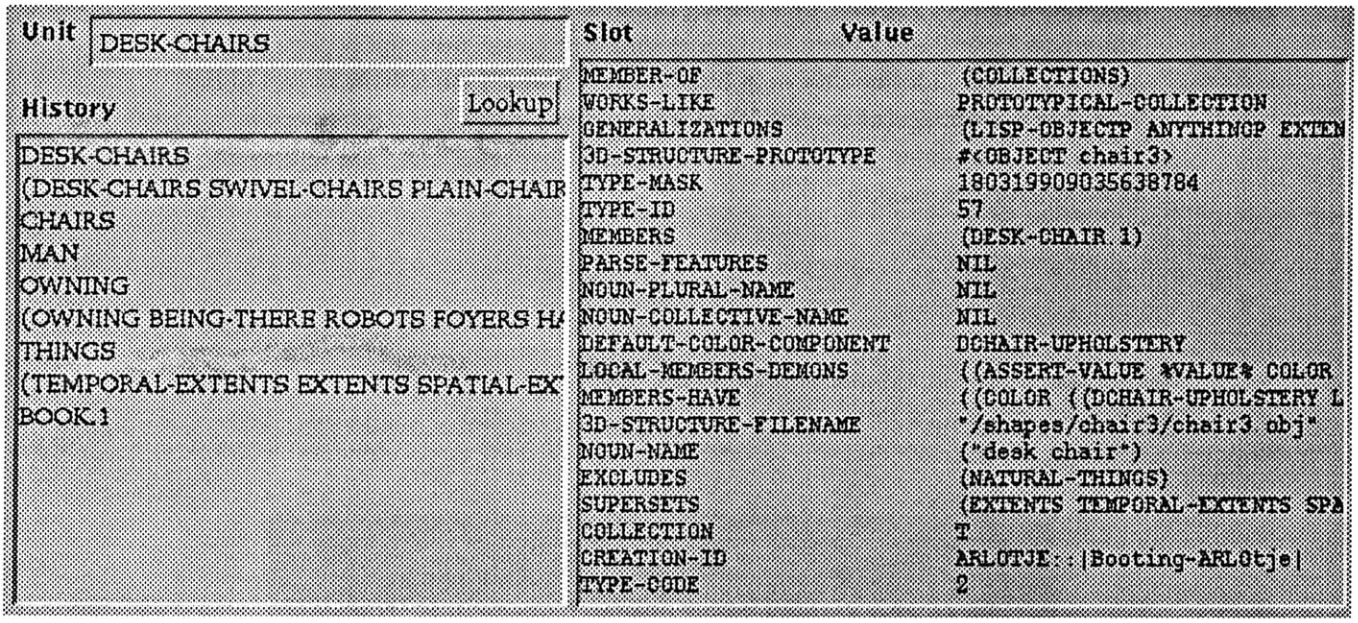
It turned out not to be as interactive as I would have hoped. Even though the sliders can be manipulated in more or less real time, it still takes about 3-5 seconds to refresh the screen after each modification. The control buttons in the lower right-hand corner of the editor allow some degree of speedup:

- *Auto redisplay* – When turned on, automatically updates the screen after every modification. Turning this off greatly enhances performance.
- *Use wireframe* – I had hoped that rendering the scene in wireframe mode would prove faster than fully phong-shaded images, but the difference was not big enough to be noticable. Since the HP Turbo SRX frame buffer has special-purpose hardware for 3d rendering operations, it takes about as much time to draw a polygon in wireframe as to do so using phong shading. Evidently, the majority of the time is taken up by the transformation of the vertex coordinates for each polyhedron, which is the same no matter which rendering style is chosen.
- *Doublebuffer* – When on, renders successive scenes to one of two alternate 12-bit frame buffers, instead of one 24-bit frame buffer. This allows you to keep a previous image on the screen until the new one is completely rendered, which avoids showing the user any distracting intermediate stages of the rendering process.

  I tend not to use double buffering while editing shapes, partly because I prefer the superior color resolution of single buffer mode, and partly because I prefer to watch the progress of the rendering, as it gives me a small degree of reassurance that something is actually being done.
- *Post object* – This button toggles the invisibility of an object, and all its children. It can be beneficial to temporarily remove an object from the scene for a number of reasons. For one, removing certain extremely complex objects can speed up rendering. For another, it allows one to temporarily remove objects which may be occluding your view of some other object which you are attempting to edit.
- *Cubify object* – This replaces the polyhedra representing an object with simple cubes indicating the bounding box surrounding that object. This was a somewhat challenging feature to implement, but like the wireframe capability, it turned out to be very disappointing in the actual payoff of improving rendering speed.

Other user interfaces

I used CLM to create three other user interfaces in the course of this thesis. One was a browser for Arlotje units, shown below.

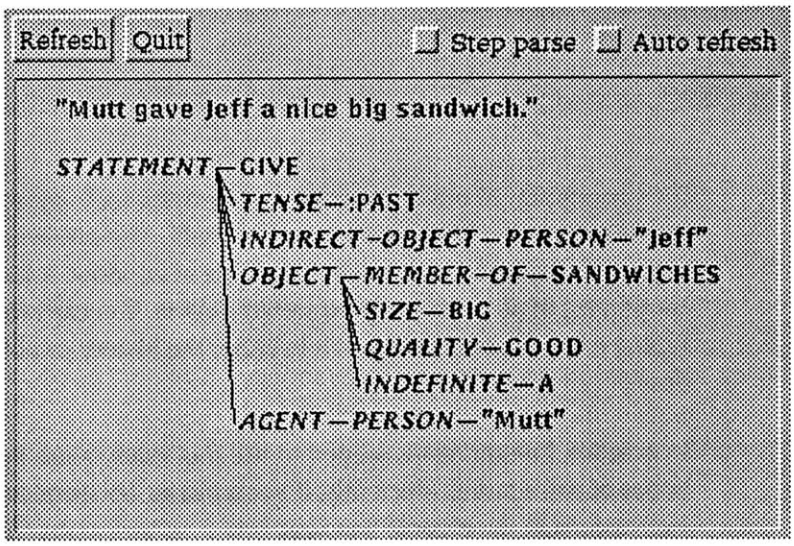


It's very similar to other inspector and browser tools common to environments such as the Lisp Machine or Smalltalk. The contents of the current object are displayed in the right hand side of window. A history mechanism keeps a list of all previously-inspected objects. You can select an object by typing its name, or clicking on it in either the history or object-contents windows.

In practice, I never really needed to use this much, since I could browse the lattice of units just as efficiently, if not more so, by simply evaluating lisp expressions in the console window.

I created two other windows to display specialized graphs. One, shown below, is used to make it easy to view the sometimes inscrutible forms returned by the parser.

The other, (shown on page 105), indicates the current state of the scene's timeline. Unfortunately, Motif is not expressive enough to allow one to draw arbitrary graphics and text. For that, I had to dig down to a lower level of representation, known as Xt, or the X toolkit. For CommonLisp programmers, this is provided by a library called CLX (the Common Lisp X library), which is available as part of the standard distribution of the X11 Windows system. Using CLM, I requested a blank "drawing-area" widget, and on this I used low-level CLX calls to actually draw the images you see.



6 *Parsing English*
- and doing something useful with it

One of the problems with a phrase like "natural language interface" is that the word *interface* implies that it can be implemented as a black box through which all user interaction is filtered. The popular illusion is that English goes in one end, and commands pop out the other, to be passed along to the simulation. The critical flaw with such a view is this: in order to refer to objects and events in the simulation, the interface must necessarily be intimately connected to the innermost details of that simulation.

I decided to take the approach that the same data structures used by the simulation would be those used by the sentence parser and generator. Unfortunately, this makes the task of explaining the representation in an orderly way a little more difficult. Which should I explain first, the data representation, or the mechanisms of natural language processing? Chapter 4 goes into more detail about the representation of objects, their properties, and about actions.

My Earley attempts

My first approach to parsing was to borrow the Earley parser [Earley] then being used by the speech group at the Media Lab. To represent the objects, I needed a flexible object representation that would provide the basic features of frames [Minsky] and semantic networks [Fahlman], including mul-

tiple inheritance, slots, etc. I implemented such a system in Lisp, and used the Earley parser to conduct some experiments.

I envisioned a simple mapping of phrases onto Lisp structures. For example, the phrase "a red apple" would be parsed as

```
(NOUN (INSTANCE-OF #<CLASS APPLE>)
      (HAS-COLOR #<COLOR RED>)
      (ARTICLE INDEFINITE))
```

Most significantly, the lisp object `#<CLASS APPLE>` was the frame indicating the `CLASS` of all `APPLES`; it would contain all pertinent information about what an `APPLE` is (see page 36). This intermediate parse could then be further refined in context to indicate a particular apple, say, `APPLE.4`.

I had to write my own grammar for this parser specifying the rules of what exactly consitutes a valid noun phrase, and what would be returned should one be encountered. Unfortunately, defining the grammar in that particular implementation proved highly awkward. Debugging a grammar involves typing in as many variations as you can think of, and seeing what kinds of parses you get back. Very often, the returned values were garbled, and it was very difficult to determine why, since the intermediate states of parser were very cryptic.

After a bit of further searching, I contacted Tim Finin at the University of Pennsylvania, who wrote BUP, a bottom-up parser for augmented phrase-structured grammars [Finin] being used by Norman Badler's group at U. Penn. He graciously sent me a copy, and I found it to be much easier to deal with. I still had to write my own grammar, but the environment for writing and debugging grammars were much friendlier. I converted my system to use BUP, and it worked pretty well.

Later on, I decided to switch my knowledge representation to Ken Haase's Arlotje [Haase]. It had many more features (see chapter 4, page 35) than the simple knowledge representation language I had built myself. Even better, since Arlotje was primarily written to understand stories, it came with its own integrated parser and natural language representation component, known as "Huh". I abandoned BUP and my frames, and converted my system over to Arlotje.

Using Huh proved to be less of a good idea than I thought it would be. It was much slower, taking as many as 10-20 seconds to parse some sentences. It was also much more awkward to define the kinds of grammars I was used to using, so, so I decided to convert back to BUP. This, then, is the current state of affairs, with BUP doing the parsing into units represented in Arlotje. Perhaps if I had more time, now that I know exactly what I want to represent, I would go back to my initial frame language, which in several ways was more efficient and appropriate to this application than Arlotje, but at least Arlotje has the advantages of being a functional, supported, and mostly debugged system, letting me attend to the other components of the thesis.

Interaction

A typical interaction with the system then looks like this:

```
>(top-level-loop)
 Creating a new scene: SCENE.1
p> The characters in this scene are John and Mary.
 Loading structure for MEN...
 Loading structure for WOMEN...
 Adding actors to the scene: (JOHN MARY)
p> John's shirt is green.
 OK.
p> He's left-handed.
 Assuming 'HE' is JOHN.
 OK.
p> There's a desk at stage left.
 Introducing a new object: DESK.1
 Loading structure for DESKS...
 Adding an object to the scene: DESK.1
 One minute while I start up Rendermatic...
 Loading DESK.1 into rendermatic.
 OK.
p> It's facing center stage.
 Assuming 'IT' is DESK.1.
 OK.
```

```
p> There's a swivel chair in front of the desk.
 Introducing a new object: SWIVEL-CHAIR.1
 Loading structure for SWIVEL-CHAIRS...
 OK.
p> Where's the blue chair?
 The blue chair is in front of the desk.
```

Calling TOP-LEVEL-LOOP starts up an interpreter which accepts a single sentence at a time, parses it, and reacts to it. TOP-LEVEL-LOOP also has a batch mode. Doing the following has the same effect as manually typing the interactive session.

```
(top-level-loop "The characters in this scene are John
    and Mary. John's shirt is green. He's left-handed.
    There's a desk at stage left. It's facing center
    stage. There's a swivel chair in front of the desk.
    Where's the blue chair?")
```

This allows canned stories to be saved and read more easily. The paragraph is broken up into single sentences, each of which is in turn printed on the console, parsed, and executed.

Building up a lexicon

The two components to building a parser using BUP is to define the lexicon and the grammar. The lexicon is automatically extracted from Arlotje units representing other concepts known to the system. These fall into 9 broad categories: NOUN, VERB, ADJECTIVE, ADVERB, PREPOSITION, PRONOUN, NUMBER-WORD, PROPER-NAME, and MISC. The functions ADD-NOUNS-TO-LEXICON, ADD-VERBS-TO-LEXICON, and so on, collect all the relevant units, determine their attributes, synonyms, and possible alternate morphologies, and install the appropriate entries into the lexicon.

At last count there were 622 units representing various concepts known to the system which could be expressed as a word in English. Considering that each noun may have both a singular and plural form, and each verb conjugates into

several forms indicating past, present, first person, third person, singular, and plural, this expands into 1474 lexicon entries.

Nouns

8-track, actor, actress, ad, advertisement, airplane, alcove, amphibian, appliance, arm, automatic, bag, baked potato, ball, baseball, bathroom, beard, bed, bedroom, beer bottle, beverage, bicycle, bike, bird, boat, book, bottle, bottom bunk, box, bunk bed, burger, camera, can, cap, car, card, carpet, carrot, cassette, cassette tape, cat, chair, character, cheek, cheeseburger, chin, chocolate, chopper, clock, cobra, computer, container, cookie jar, couch, crowbar, cup, cycle, deck, den, desk, desk chair, dessert, dining room, dish, dishwasher, disk, diskette, dog, door, drier, drinking glass, dryer, duck, ear, eye, eyebrow, faucet, faucet knob, file cabinet, filing cabinet, finger, fish, floor, floppy, floppy disk, food, foot, fork, foyer, fridge, frisbee, frog, fruit juice, frying pan, frypan, gadget, glass, goatee, goose, guitar, gun, hall, hallway, hamburger, hand, head, heater, helicopter, hot dog, humidifier, ice box, instrument, jack, jar, journal, juice, ketchup bottle, king-size bed, kitchen, knife, lamp, laptop, leg, lid, light, lip, living room, lizard, lounge, love seat, Mac, machine, machine gun, Macintosh, magazine, mammal, man, marker, mayonnaise jar, microcomputer, microscope, milk bottle, missile, moped, motorbike, motorcycle, mouse pad, moustache, mouth, mug, newspaper, newt, NeXT, NeXT computer, NeXT machine, nose, onion, orange, orange juice, pan, patriot, PC, pelvis, pen, pencil, pepper shaker, person, personal computer, phone, piano, pickle, pickle jar, picture, pistol, pizza, plain chair, plate, porch, pot, potato, prop, range, refrigerator, reptile, revolver, rifle, room, rug, salad, salt shaker, sandwich, saucer, sausage, scene, scooter, scud, shaker, shelf, shin, ship, shirt, shoe, shower, single bed, sink, six-pack, sixpack, snake, sock, soda bottle, sofa, softball, spoon, stapler, stool, story, stove, sweater, swivel chair, table, tape, tape dispenser, telephone, thigh, thumb, toaster, toe, toilet, tomato, tomato juice, tongue, tooth, top bunk, toy, truck, tub, tuna, turtle,

utensil, vacuum cleaner, vegetable, vehicle, video cassette, Video Toaster, videocassette, wall, washer, washing machine, weapon, weenie, window, woman, wrist.

Verbs

attract, bake, become, begin, betray, bow, bring, broil, buy, cheat, close, commence, contain, cook, copy, create, cry, curse, dance, describe, dislike, do, dolly, drink, drop, eat, end, enter, face, fear, feel, finger, fool, fry, gesture, get, give, glare, go, grab, hack, hand, hate, have, help, hit, hold, honor, hop, hug, include, is, kiss, laugh, leave, let, lift, like, log, look, love, manipulate, move, need, new, offer, open, operate, pan, pause, pick, pull, push, put, record, recount, refuse, render, respect, restore, run, sang, saute, say, scam, see, shake, shout, show, shrug, sing, sit, skip, slam, smack, stand, start, steal, stop, swindle, take, tell, think, throw, tilt, took, truck, trust, turn, wait, walk, want, wave, weep, whack, work, zoom.

Adjectives

African, agnostic, ahead, alien, American, angry, arrogant, Asian, atheist, awkward, back, backward, backwards, bad, big, black, blonde, blue, bodacious, bold, Bostonian, bottom, brown, brunette, Buddhist, butterscotch, Californian, calm, calmed, caucasian, center, center stage, cheered, cheerful, chilled, Chinese, Christian, clear, closed, clumsy, cold, comfortable, comfy, cool, crazed, crazy, dark blue, dark brown, dark denim, dark gray, dark green, dark grey, dark red, delicious, delighted, delirious, denim, dependable, disturbed, down, downstage, dull blue, energetic, English, excited, extra-terrestrial, eye-level, fair, fatigued, floor-level, forward, forwards, French, frenzied, front, frosty, furtive, fussy, German, giant, glassy, gold, good, gray, great, green, grey, gross, happy, horrid, hot, huge, humungous, immortal, impatient, in, in love, irritable, Islamic, Japanese, Jewish, joyful, jumbo, kind, large, lazy, left, left-handed, lefty, light blue, light denim, long, lousy, lower, lumpy, mad, mauve, mellow, metallic, mortal, mustard, navy, neat, nervous, new, nice, non-simulated, offstage, old, open, orange, out, overhead, pagan, pasty-nerd, patient, peeved, peppy, pink, pissed, plaid, polite, poor, precise, prissy, proud, puke green, purple, red, redhead, relaxed, responsible, reverse, right, right-handed, righty, romantic, rough, rude, Russian, sad, saddened, short,

shy, silver, simulated, sloppy, small, so-so, spoiled, stage left, stage right, stealthy, straw, Subgenius, suntan, tall, tan, tasty, teeny, terrible, timid, tired, top, transparent, trustworthy, uncomfortable, unhappy, untrustworthy, up, upper, upset, upstage, violet, warm, white, wonderful, yellow, yucky, yummy, Zoroastrian.

Adverbs

a little, a lot, a wee bit, angrily, arrogantly, awkwardly, boldly, calmly, cheerfully, clumsily, crazily, deliriously, dependably, energetically, energetically, excitedly, extremely, fast, furtively, fussily, happily, impatiently, irritably, joyfully, kind of, kindly, lazily, lethargically, madly, more, neatly, nervously, no longer, not, not so, not too, patiently, ploddingly, politely, precisely, pretty, prissily, proudly, quickly, quite, rather, really, responsibly, romantically, roughly, rudely, sadly, shyly, slightly, sloppily, slowly, so, stealthily, timidly, totally, trustworthily, untrustworthily, very, way, zippily.

Prepositions

along, among, at, away from, before, behind, beneath, beside, between, by, for, from, in, in back of, in front of, inside, into, next to, on, on top of, onto, to, toward, towards, under, using, with, within.

Names

Adam, Al, Alan, Alex, Amy, Ann, Arlene, Barb, Barbara, Barbie, Barry, Ben, Betty, Bob, Carl, Cathy, Cinderella, Cindy, Cooper, Danny, Dave, David, Diane, Dustin, Eddie, Ellen, Eve, Frank, Fred, Gene, Gilberte, Gloria, Greg, Greykell, Guido, Henry, Hiro, Iggy, Igor, Jane, Jean, Jeanne, Jeff, Joe, John, Jon, Julie, Karl, Kathleen, Kathy, Kayla, Ken, Kenji, Kermit, Kevin, Kimiko, Laura, Lisa, Marcia, Marg, Margaret, Mariko, Marvin, Mary, Matt, Mel, Meryl, Michael, Mickey, Mike,Molly, Mutt, Nicholas, Nick, Norm, Norman, Penn, Pete, Peter, Porsche, Ray, Rick, Rob, Robert, Robin, Sally, Sarah, Steve, Straz, Sue, Teller, Tod, Tomoko, Walt, Walter, Wave, Wendy.

Pronouns

he, her, her, here, herself, him, himself, his, i, it, me, my, myself, our, ourselves, she, that, their, them, themselves, there, they, this, those, us, we, you, your,

```
yourself, yourselves.
```
Numbers
```
   couple of, dozen, eight, eighteen, eighty, eleven,
fifteen, fifty, five, forty, four, fourteen, gross of, half
a, half of, nine, nineteen, ninety, one, pair of, seven,
seventeen, seventy, six, sixteen, sixty, ten, thirteen,
thirty, three, twelve, twenty, two.
```

Clearly not all of these are used to generate animations, but it's extraordinarily useful to have a vocabulary this big. Maybe one could get by with perhaps only a few dozen words, but such a system would prompt some important questions about extensibility.

- *How hard is it to add new words to the lexicon?*

  Not hard at all - an entry basically identifies a word's spelling, synonyms (if any), and part of speech. Once the mechanism works, I found typing the first 600 entries relatively painless.

- *How well could this scale up for a practical application?*

  Since the lexicon is stored in a hash table, parsing is very efficent. Most sentences I use (typically 7 words or less) parse in about 1/10 second. More challenging examples, such as a compound sentence of 20 words, still parse in under a second.

A large vocabulary is more enjoyable to work with. While developing the parser, I tried to never enter the same sentence twice. Aside from making the work less monotonous, the diversity of possible combinations was a strong reminder that I was trying to build a system that would ultimately be capable of a broad variety of possible scenarios. There are many opportunities with a smaller system to "cheat" and handle a few simple sentences in a specialized way. Having to support a large vocabulary in many ways kept me honest, forcing me to think out more general solutions to a number of problems, such as object reference, pronoun reference, and modifiers such as adjectives, adverbs, and prepositional phrases.

Lexicon entries

The lexicon is a hash table which maps strings into lists of entries. Each entry is a list of three elements:

1. part of speech
2. representational unit
3. a list of parse features

Here are some typical examples:

```
(gethash "orange" *lexicon*)
⇒ ((NOUN ORANGES (SINGULAR))
    (ADJECTIVE ORANGE NIL))
(gethash "threw" *lexicon*)
⇒ ((VERB THROW (PAST UP DOWN TRANSITIVE
                INTRANSITIVE INDIR-OBJ)))
```

We see that "orange" has two entries, one noun and one adjective, and "threw" is a verb with several features. The representation for the noun "orange" is ORANGES, which is the set of all oranges, i.e. a subset of the class FRUITS whose 3d-structure (i.e. shape) is a sphere, etc. The adjective "orange" maps onto the the unit ORANGE, which is an element of the set of COLORS whose property-value, as far as the system is concerned, is `[red: 1, green: .5, blue: 0]`.

The parse features will be explained in more detail later, but they are used to augment parses. Thus, the noun "orange" is known to be a singular noun, and the verb "threw" may accept an indirect object.

Any unit in the knowledge base may have one or more slots identifying grammatical information. For example, the entry for THROW looks like this:

```
(define-collection throw (action-classes)
  (verb-name "throw")
  (verb-past-name "threw")
  (parse-features '(up down transitive
                    intransitive indir-obj)))
```

The VERB-NAME slot indicates that the unit THROW is a verb, whose various forms may be derived from the root "throw", except for the past tense, which is "threw". Corresponding slots exist for all the other parts of speech. Multiple values can be provided for slots, allowing a single unit to have many synonyms. Thus the unit REFRIGERATORS appears in the lexicon redundantly as "refrigerator", "fridge", and "ice box".

Word morphology

To cut down on the pain of entering variations, I wrote several utilities to automate generating the various forms of a word's root. These include:

1. plurals, i.e. "french fry" → "french fries"
2. participles, i.e. "hop" → "hopping"
3. past tense, i.e. "bake" → "baked"
4. third person singular, i.e. "get" → "gets"

Of course, irregular variations (such as "man"/"men") may be explicitly specified, but the great majority of cases can be captured by a few simple rules.

I also wrote some handy utilities which go the reverse way, namely, guessing at a word's root, given its extended form. Given the vagaries of English, the guesses need to be confirmed by looking them up the lexicon to see if they actually match a known root. In the end, I never needed to use this reverse lookup since all possible forms were generated and placed into the lexicon explicitly.

Tokenizing

The first step to parsing a sentence is to break it up into tokens. For this, simply separating words at spaces and punctuation marks would not suffice, since I wanted to support compound words, such as "New York". Such lexicon entries, which contain embedded characters which would normally be construed as delimiters, are kept in a special list called *FUNNY-WORDS*[1]. Also compounds

may be validly parsed as separate words, provided that each of these separate words are themselves to be found in the lexicon. This gives rise to the possibility of multiple ambiguous parses.

Another source of ambiguity during tokenization is the expansion of contractions. I use contractions heavily in this code for several reasons; for one, I'm a lazy typist. For another, it makes the dialog seem less stilted and more natural. Lastly, perhaps it's because I've seen so few other systems that were capable of handling contractions properly. It's not so hard to do, but for some reason it's the sort of nicety that people tend to pass over.

For example, "John's" could expand into either "John is" or a special form indicating the possessive form of John. The routine EXPAND-CONTRACTION returns a list of all possible expansions of a contraction, thus:

```
 (expand-contraction "I'd")
⇒ (("I" "would") ("I" "had") ("I" "did"))
 (expand-contraction "Harry's")
⇒ (("Harry" "is") ("Harry" *possessive*))
```

The function TOKENIZE therefore returns a list of all plausible tokenizations of the given string. For example, here's a sentence containing both an ambiguous contraction, and an ambiguous compound word:

```
 (tokenize "Steve's NeXT machine is black.")
⇒ ((("Steve" "is" "NeXT" "machine" "is" "black") NIL)
    (("Steve" "is" "NeXT machine" "is" "black") NIL)
    (("Steve" *possessive* "NeXT machine" "is" "black")
      NIL)
   (("Steve" *possessive* "NeXT" "machine" "is" "black")
     NIL))
```

If one or more unknown words are encountered during the parse, they are returned instead, as the second element of the tokenization.

```
 (tokenize "The goons are glorking their blorts.")
⇒ ((NIL ("goons" "glorking" "blorts")))
```

1. idea inspired by this capability in Haase's HUH

It's much nicer to tell the user which words were unrecognized, than to merely fail. The more informative the system is about failing to recognize input, the more quickly the user can adapt their expectations to the system's shortcomings. I found that while giving demos of the parser to people, they'd readily suggest sentences they thought the system should know about. Since it was relatively easy to extend the vocabulary, I incorporated most new words as soon as they were discovered to be missing.

For the purposes of this system, punctuation within a sentence, as well as capitalization, are ignored. Thus "No, John is free now." and "No john is free now!" would be broken up into identical token lists.

Numbers and embedded strings are preserved, and passed along to the parser. This is very handy, since it allows processing of words and numbers that are not explicitly in the lexicon.

```
 (tokenize "The camera marked \"Sony\" is 3.6 inches long.")
⇒ ((("The" "camera" "marked" "\"Sony\"" "is" "3.6" "inches"
"long") NIL))
```

Categories of input

Each tokenization is given in turn to the parser, in an attempt to determine which one gives the best parse. The function PARSE returns a weighted list of possible outcomes; the one with the highest score is the one that is used. If no valid parse is returned, the user is informed that the input couldn't be understood by the system.

A good parse, for our purposes, is some lisp form which can be readily used by some other component of the system. I identified three categories of user input, each of which are described in more detail below.

1. QUERY - This is a request by the user for some information about the scene. For example, these are some valid questions: "What is John holding?", "Is he crying?", "What's the chair's color?", etc. Each of these questions cause an appropriate response to be given to the user, in a format as close as possible to English.

2. COMMAND - Some immediate action should be taken. The system should perform the action, and print "OK" if it was successful. In practice, commands are only used for a few, non-story-related functions, such as refreshing the screen, debugging operations, and quitting the interactive loop.
3. STATEMENT - The majority of input sentences are statements, describing various aspects of the scene, including the properties of objects and describing actions to be performed. For example, a statement can declare that a character is tired, or will pick up a book.

What the parser returns

The grammar is the core of the parser - it defines the rules for interpreting the input sentences, and it determines what they produce. In this section I will describe the grammar I wrote for my system, using the features provided by BUP.

BUP starts with the sentence as a list of tokens, and tries to build a "syntax" tree and a "semantics" tree in parallel. The "syntax" tree is constructed by applying the grammatical rules to the input sentence, and returning a tree indicating which rules were applied:

```
(parse "John gave Mary some Chinese food." :syntax)
⇒ (S (STATEMENT
        (NP (FIRST-NAME "John"))
        (VP (VP1 (V (VERB "gave"))
            (NP (FIRST-NAME "Mary"))
            (NP (INDEF-ARTICLE "some")
                (NP1 (ADJ (ADJ1 (ADJECTIVE "Chinese")))
                     (NP1 (NOUN "food")))))))))
```

Although in one sense, this is an accurate interpretation of the sentence, it's still extremely awkward. It's very painful to compute anything with such a parse, unless one writes a second parser to extract useful chunks from it. Fortunately, BUP allows one to specify the "semantics" of a parse in a somewhat better way. The semantics of a rule is an arbitrary Lisp expression for-

mulating the value returned by that rule. Thus, the semantics of the same sentence would be:

```
 (parse "John gave Mary some Chinese food." :semantics) ⇒
(STATEMENT GIVE
     (TENSE :PAST)
     (INDIRECT-OBJECT (PERSON "Mary"))
     (OBJECT (MEMBER-OF FOODS)
             (NATIONALITY CHINESE)
             (INDEFINITE SOME))
     (AGENT (PERSON "John")))
```

This gives us all the information we need as a simple list of handy slot-value pairs. What we want is a parser that will convert all input sentences into an intermediate structure like this, which I will call "the parse".

If you're not particularly interested in how this is done, skip on to the next section (on page 79), which describes what to do with a parse like this once you have it. The rest of this section is an analysis of the rules I wrote.

The CAR of the parse is a keyword, either QUERY, COMMAND, or STATEMENT, indicating the category of the parse. As each sentence is typed in by the user and parsed, it is passed to a function called REACT-TO-PARSE, which simply looks at the category and passes it off to an appropriate handler; one of REACT-TO-QUERY, REACT-TO-COMMAND, or REACT-TO-STATEMENT. Each of these are described in more detail below.

There are two special kinds of variations on STATEMENTs. Compound statements can indicate two or more actions happening, either in parallel or in sequence. In these cases, the CAR of the parse is either the token PARALLEL or SERIAL, followed by two or more STATEMENTs. For example,

```
 (parse "As he walks to the window, Mary picks up the book.")
⇒ (PARALLEL
   (STATEMENT WALK
      (TENSE :PRESENT)
      (TO (MEMBER-OF WINDOWS)
          (DEFINITE THE))
      (AGENT (PRONOUN HE)))
   (STATEMENT PICK-UP
```

```
(TENSE :PRESENT)
(OBJECT (MEMBER-OF BOOKS)
        (DEFINITE THE))
(AUXPREP UP)
(AGENT (PERSON "Mary"))))
```

Matching noun phrases

Here are some examples of noun phrases, and what the parser returns for them:

```
"He"
((PRONOUN HE))

"A pair of Ken's not so large, extremely
 blue pizza boxes"
((MEMBER-OF BOXES)
 (CONTAINING PIZZA)
 (COLOR BLUE VERY)
 (SIZE BIG SLIGHTLY)
 (BELONGS-TO (PERSON "Ken"))
 (QUANTITY 2)
 (INDEFINITE A))

"His shirt"
((MEMBER-OF SHIRTS)
 (BELONGS-TO (PRONOUN HIS)))

"Penn, Teller, and that English guy"
(AND ((PERSON "Penn"))
     ((PERSON "Teller"))
     ((MEMBER-OF MEN)
      (NATIONALITY ENGLISH)
      (DEFINITE THAT)))
```

This structure must then be interpreted further in one of two ways. Either it is a reference to something already in the scene, or it is the specification for something entirely new to be constructed.

The function MATCH-NP, when given such a phrase, attempts to determine if such a referent already exists, and if so, to return it. It can be called with one of three optional keywords to control what it returns:

- :one - (the default) attempts to find a unique referent in the scene which fully matches the description, or else signals a failure.
- :some - returns a list of at least one, and possibly several referents, each of which must fully match the description. If none are found, signal a failure.
- :any - same as :some, except if none is found, simply return NIL instead of a failure.

There are several classes of noun phrases, and MATCH-NP behaves differently for each of them. It looks for one of these keywords

(PERSON "John") - Although it's exceedingly rare in movies and plays for two characters to have the same first name, it's rather common in real life. The parsed name is retained as a string, so as not to confuse a name like "John" with the unit JOHN, which represents a particular person by that name. It's conceivable that (PERSON "John") could refer to two or more referents, like JOHN-LENNON and JOHN-ADAMS.

(MEMBER-OF CHAIRS) - Due to Arlotje's classification scheme, objects are identified by a collection (set) that they belong to, rather than a prototype (class) of which they are an instance. Thus the unit CHAIRS is the set of all CHAIRS. If the noun phrase contains a MEMBER-OF clause, MATCH-NP starts with that set of objects. The other clauses are filters, each of which reduces the set by eliminating members which do not fit the description. For example the filter (COLOR BLUE) compares each object's color to see whether it's blue[1]. After all filters are applied, the remaining object or objects are taken to be the referent.

1. Actually there's a little leeway built into this. A color is taken to be a point in a unit cube with the range 0-1 in each dimension representing the red, green, and blue components. An object of color C_o is considered to match a given descriptive color C_d if the euclidean distance between C_d and C_o is within some threshold, which I arbitrarily picked to be .3. This allows the word "blue" to describe objects that are light blue, dark blue, and navy blue, but not those that are green or yellow. Perhaps another more sophisticated method could be implemented, but this suffices for my needs. A comparable degree of flexibility is built into filters for other descriptions, such as "upstage", "downstage", etc.

(PRONOUN IT) - Each pronoun in the lexicon is tagged as to gender, plurality, and whether the referent is necessarily animate or inanimate. As sentences are parsed, referents are brought to the top of a queue which keeps track of the chronological order of noun references. Each scene has its own queue. Matching a pronoun, therefore, is done by searching down the queue until a referent is found which matches the pronoun in gender, plurality, etc. This effectively matches each pronoun to the most recently used noun, while ensuring that pronouns like "her" only refer to women, etc.

Second-person pronouns ("you", "yourself") have no referents. All first-person pronouns refer to a special token, THE-AUDIENCE, so that a command like "John is looking at us" causes him to turn toward the camera.

Self-evaluating referents - These include strings like "No Smoking", or Lisp symbols like 'CHAIR.5 (indicated by the leading quote) for unambiguous references to objects. The parser allows a user to type either of these in place of a regular noun phrase, permitting sentences like these:

The title of the book is "Moby Dick".
Put the book on 'CHAIR.5.

In practice, I never really used either of these very much. In the case of the latter, it never became necessary to refer explicitly to an object's internal ID when phrases like "the red chair" or simply "it" suffice.

Verb phrases

A verb phrase is a list whose CAR is a unit denoting a verb, and whose CDR is a list of clauses (lists), each identifying various pieces of information gleaned from the parse. For example, here's the verb phrase

```
"Slowly he turned the gun toward her with his left hand."
(TURN-TO
   (TENSE :PAST)
   (OBJECT (MEMBER-OF GUNS)
           (DEFINITE THE))
```

```
(TO (PRONOUN SHE))
(WITH (MEMBER-OF HANDS)
      (POSITION LEFT)
               (BELONGS-TO (PRONOUN HIS)))
      (AGENT (PRONOUN HE)))
(SPEED SLOWLY))
```

This makes it very easy to extract all the relevant information from the parse. Each clause within the verb phrase tells you the tense, object, subject, instrument, etc. of the action. Those clauses which are nouns, such as OBJECT or AGENT, contain a noun phrase in their CDR, which can be passed to MATCH-NP (see above) to determine their reference. Other clauses, such as TENSE and SPEED, are handled idiosyncratically.

It's no misprint that the prepositional phrase "toward her" was converted into (TO (PRONOUN SHE)). For the sake of simplicity, words like "to" and "toward", or "she" and "her" are treated as synonyms, since the distinctions are on a level of subtlety beyond the scope of this thesis. Having lots of appropriate synonyms makes it easier to enter sentences flexibly without the frustration of having to remember a specific "correct" word.

Queries

Here's how the parser interprets a question asked by the user:

```
 (parse "Where's the desk?")
⇒ (QUERY LOCATION? ((MEMBER-OF DESKS) (DEFINITE THE)))

(parse "What is John doing?")
⇒ (QUERY DOING-WHAT? (PERSON "John"))
```

There's quite an enormous range of questions one could conceivably ask a system like this one, and attempting to answer them all is beyond the scope of this thesis. For that matter, it has been suggested to me by some observers that having it answer *any* questions was not particularly critical to this thesis.

I thought it would be useful to allow users to interrogate the system in plain English, rather than mucking around and evaluating lisp variables. Remember, I'm trying to stay with strictly non-technical interaction methods.

As it turns out, pretty much all of the questions you could ask are kind of silly, since the answer is usually obvious by simply looking at the screen. Nevertheless, I still feel there's a need to support such queries. As these systems get more complicated, inspecting the screen will become less practical, and verbal explanations will become more useful.

If the user's sentence is parsed into a query, it is passed to the function `REACT-TO-QUERY`, which attempts to respond in an appropriate way. There are several categories of queries, indicated by a special token in the second element of the query. Here are the possible categories, their format as returned by the parser, and their meanings:

`(query location? NP)`	where is John?
`(query doing-what? NP)`	what is John doing?
`(query who-is PPHRASE)`	who is near the bookcase?
`(query who-is-doing? VP)`	who is eating the pie?
`(query object? VP NP)`	what is John eating?
`(query doing-y/n? VP NP)`	is John eating? (yes or no)
(query apt-description? NP ADJ)	is John left-handed? (yes or no)
`(query same-object? NP1 NP2)`	is John an actor? (yes or no)

There's no particularly methodical way to answer these questions. For each category, there's a function which accepts the parse and composes an answer to that question. Yes/no questions are answered easily enough by printing "yes" or "no."

Other questions are answered by inspecting appropriate slots of the noun referents in question, extracting the information, and presenting the answer as an English sentence. For example, the question "where is Frank's chair?"

is answered by printing something like "The red chair is in front of the desk."

There are two interesting aspects to an answer like this: why did the system use a phrase like "the red chair" to identify the chair, and how did it know to say something clever like "in front of the desk" instead of "at coordinates `(45.32, -3.21)`"?

The function `TEXT-FOR` returns a text fragment describing an object. So, `(TEXT-FOR 'FRANK)` simply returns the string "Frank". If there were one chair in the scene, say, `CHAIR.3`, `(TEXT-FOR 'CHAIR.3)` would return a string like "the chair". However, if there's more than one chair, that would still be ambiguous, so another routine is called to supply sufficiently disambiguating adjectives. The resulting string "the red chair", is returned by `TEXT-FOR`.

As for the chair's location, one way to answer such questions is to indeed find another nearby, large object in the scene, such as the desk. One could then see if a prepositional phrase such as "in front of" would be a fitting description, and if so, generate a phrase like "in front of the desk" (using `TEXT-FOR` to get the description of the desk.) This would be pretty slick, but it's not what I did.

One of the advantages of creating and maintaining symbolic descriptions of a scene is that it greatly simplifies things when you need output. As we shall see below, the chair was actually put in its place by a statement from the user like "the chair is in front of the desk". Until the chair or the desk is moved, then, the chair's `POSITION` slot contains the value `(BEFORE DESK.1)`. The prepositional unit `BEFORE` has two synonymous lexicon entries: "before" and "in front of". So, with no thinking required, the phrase `(BEFORE DESK.1)` can be passed along to `TEXT-FOR`, which turns this into a string like "in front of the desk."

Other relationships, like "the book is on the TV", and "the spoon is in the cup" are managed in the same way. The system makes no attempt to reason about the consequences of actions; when objects are moved by forces other than a user's command, these symbolic descriptions are simply deleted whenever they are deemed undependable. As a last resort, the location of an object is indeed given as spatial coordinates. Under such circumstances, some mechanism like the slick one described above would be needed to deduce a qualitative relationship from the available quantitative data.

Whenever possible, however, these symbolic relationships are preserved, to augment low-level numerical representations of position and orientation. These are very useful to other parts of the system which may take advantage of geometric knowledge readily available in a symbolic form.

Extending the repertoire of possible queries is pretty straightforward. For each new kind of question, one must add a new grammar rule to generate an appropriate parse. Then, one must extend `REACT-TO-QUERY` to accommodate a new module which answers such questions whenever they are asked.

Commands

A few simple commands are handled by `REACT-TO-COMMAND`. These include

- *Render* - refreshes the screen
- *New scene* - creates a new scene and makes it the current one.
- *Describe* - prints a verbose lisp description of an object
- *Recount scene* - retell the events in this scene up to now.
- *Quit* - terminates the program

A few other commands involve camera movements, repositioning the camera to one of a dozen preset locations, such as "low front", "right", "left", "overhead", etc.

Statements

Aside from queries and commands, most of the input consists of "statements", which is how the user describes the scene to the system. As these are parsed, they are passed to a function named `REACT-TO-STATEMENT` which decides how to handle them.

Immediate versus delayed response

One important issue I had to resolve fairly early was distinguishing *when* things happen, and how the user could express control over it. There are two modes of command:

- *Blocking* is the technical term used in theater to describe positioning characters on the stage. A user should be able to interactively block a scene by saying things like "John is near the window." or perhaps move him elsewhere instead with a command like "He's in front of the table."
- When the action actually starts and the camera is rolling, characters move and perform various actions.

The problem is, when a character is told to go to location *X*, does that mean that *X* is where she starts at the beginning of the scene? Or does that mean that within the scene, the character should walk over to location *X*?

One way to resolve this would be to provide the user with a switch specifying *setup* and *script* mode. While the system is in setup mode, all commands would be interpreted right away and their consequences immediately performed. Conversely, all commands in "script" mode would be queued onto a timeline to form the script.

I decided not to have any such switch, partly because it would have detracted from my objective of having the interface wholly language-driven. Also I thought it might be a pain for a user to have to go flip a switch in order for his input to be interpreted correctly.

Instead, I decided to take advantage of the fact that my parser could distinguish among various tenses of verbs. Thus,

"John is sitting on the chair" becomes a "setup" statement, establishing an initial condition for the scene.

"John sits on the chair" becomes part of the script, an action to be performed during the scene.

There was a third possibility that I discarded. Early on, I presumed that I would have a single actor to boss around, so I assumed I would be giving direct commands, like "Go to the window. Open it." As I started describing scenes with multiple actors, it became necessary to say things like "John, go to the window. Open it." I suppose it's purely a matter of taste, but I decided that the first two modes of address seemed to have more of a "screenplay" feel to them. After an interaction session with the system, the transcript can be saved into a file and when read, seems to be more like a screenplay: "John is sitting on the chair. He looks out the window.."

In retrospect I should probably go back to supporting this third mode as an alternative to "setup" statements; it would be quite easy to do. There would be the small matter of keeping track of who the user was addressing with any given command. A new imperative sentence beginning with a character's name, such as "John, get the ball", specifies the focus of attention.

Setup statements

A setup statement is a statement that is either

- a `HAS-PROPERTY` statement, or
- has the `:PARTICIPLE` keyword in its `TENSE` slot.

Here are some typical setup statements:

```
(parse "John is sitting on the chair")
⇒ (STATEMENT SIT
      (TENSE :PRESENT :PARTICIPLE)
      (ON (MEMBER-OF CHAIRS)
          (DEFINITE THE))
      (AGENT (PERSON "John")))

(parse "He's facing stage left center."))
⇒ (STATEMENT FACE
     (TENSE :PRESENT :PARTICIPLE)
     (TO (POSITION LEFT CENTER))
```

```
      (AGENT (PRONOUN HE)))

 (parse "Sue's book is blue")
⇒ (STATEMENT HAS-PROPERTY
       (OBJECT (MEMBER-OF BOOKS)
               (BELONGS-TO (PERSON "Sue")))
       (PROPERTY (COLOR BLUE)))
```

Each of these statements is turned into an assertion which is immediately acted upon. After the assertion is acted on, the system prints "Ok." and waits for another sentence from the user.

There are several categories of assertions, most of them dealing with various attributes of objects, such as their location, orientation, and color, or in the case of characters, such attributes as mood, handedness, etc. These are described in more detail in the section on properties (see page 38).

An important aspect of these sentences is that they can introduce new objects into the scene. Human objects are "actors" or "characters". In this thesis I use these two words interchangably, but when writing code, I use the term *actor* purely for brevity. An inanimate object is a *prop*.

When a scene is created, it contains no props or actors. New actors are usually introduced by statements like these:

```
 (parse "John is an actor in this scene.")
⇒ (STATEMENT BE
      (TENSE :PRESENT)
      (OBJECT (MEMBER-OF ACTORS)
              (INDEFINITE AN))
      (IN (MEMBER-OF SCENES)
          (DEFINITE THIS))
      (AGENT (PERSON "John")))

 (parse "The characters are Larry, Moe, and Curly.")
 ⇒ (STATEMENT BE (TENSE :PRESENT)
    (OBJECT AND
        ((PERSON "Larry"))
        ((PERSON "Moe"))
        ((PERSON "Curly")))
```

```
    (AGENT (MEMBER-OF ACTORS)
           (DEFINITE THE)))
```

In both cases, the verb is BE, and either the AGENT or OBJECT slot of the verb refers to the set of ACTORS. These statements are interpreted as calling for the introduction of new characters to the scene, if they are not there already.

If a setup statement refers to an object that is not part of the scene, we have to find or build one, and add it to the scene. This principle applies equally to both props and actors, and uses the same mechanism in both cases. For example,

```
 (parse "She's holding a dark blue book.")
⇒ (STATEMENT HOLD
   (TENSE :PRESENT :PARTICIPLE)
   (OBJECT (MEMBER-OF BOOKS)
      (COLOR DARK-BLUE)
      (INDEFINITE A))
   (AGENT (PRONOUN SHE)))
```

When trying to establish the referent in the OBJECT slot, a search is made of the props in the scene, to see if there any objects matching the description

```
((MEMBER-OF BOOKS)
 (COLOR DARK-BLUE)
 (INDEFINITE A))
```

If none are found, the search extends to the global list of objects. These include objects which may have been created and used in other scenes. If this fails, a new object must be created, matching the given description.

Once an object matching the description is found or created, it is added, as appropriate, to either the scene's list of actors or list of props. The former is stored in the SCENE-ACTORS slot of a scene, the latter in SCENE-PROPS, and their union in SCENE-OBJECTS.

Another list, called SCENE-REFERENCES, keeps track of every object referred to by the user (in all parsed sentences) in chronological order. The

most recently referred object is at the beginning, followed by older references. As mentioned above in the discussion on MATCH-NP (see page 79), this is used to determine the referent of a pronoun.

Script statements

Any statement which is not a setup statement is taken to be a script statement. This describes actions to be performed when animating the scene. These create what I call "scene actions". Here's some examples:

```
 (parse "John throws her the ball."))
⇒ (STATEMENT THROW
     (TENSE :PRESENT)
     (INDIRECT-OBJECT (PRONOUN SHE))
     (OBJECT (MEMBER-OF BALLS)
             (DEFINITE THE))
     (AGENT (PERSON "John")))

 (parse "While he walks to the window, Mary picks up the
book."))
⇒ (PARALLEL
     (STATEMENT WALK
       (TENSE :PRESENT)
       (TO (MEMBER-OF WINDOWS)
           (DEFINITE THE))
       (AGENT (PRONOUN HE)))
     (STATEMENT PICK-UP
       (TENSE :PRESENT)
       (OBJECT (MEMBER-OF BOOKS)
               (DEFINITE THE))
       (AUXPREP UP)
       (AGENT (PERSON "Mary"))))
```

Simple statements, like the first example, are passed to a function called INSTANCE-ACTION-FROM-PARSE which creates a new action and adds it to the scene. Compound statements, like the second, handled in much the same way. They are given to INSTANCE-COMBO-ACTION-FROM-PARSE, which instances each of the components, and then adds the results to the timeline.

A new instance of an action, such as THROW, is created using Arlotje's NEW-UNIT mechanism. This creates a new unit with a unique identifier, such as THROW.3, and adds it to the MEMBERS slot of the action class THROW. Other slots which are initialized include ACTION-PARSE, which contains the parse itself, and ACTION-SENTENCE, which contains the original text string of the sentence as typed by the user. In the case of compound actions, each action gets a copy of the original parse and sentence. The action's ACTION-SCENE slot is initialized to the current scene.

After a statement is parsed, it is examined for clues to determine where it should be inserted into the scene. Phrases which do not have any explicit timing information are assumed to occur in serial order. The actions are added to the scene's *timeline* (see page 43 for more information about how timelines are represented).

An example of a temporal clue can be found in the sentence "In this scene, John gets angry." The phrase "in this scene" indicates that this occurs over the course of the entire scene, so all other actions ought to occur in parallel to this.

Other phrases, like "while...", "as...", and "meanwhile..." indicate events which occur in parallel with other events. Events happen in serial order when they contain phrases like "then..." or "afterwards...". More sophisticated techniques could be probably used to extract temporal clues from phrases like "Between the two explosions, he managed to duck behind the sofa," but I did not attempt to go this far.

There is placeholder indicating the current insertion point into a scene, called the INSERTION-GROUP of a scene. As successive statements from the user are parsed, the corresponding actions are inserted wherever indicated (if explicit temporal constraints can be deduced from the sentence), or appended to the end of the insertion group by default.

Of course, series of actions do not necessarily follow immediately one after another, and actions in parallel do not necessarily begin or end at exactly the same moment. I allow for the insertion of dummy actions, which are

instances of the "wait" action class. These can be used to fill in those places were a little slack is needed. Thus, if `ACTION.2` were to begin just a little bit after `ACTION.1`, you could represent that with an expression like this:

```
(PARALLEL ACT.1 (SERIAL WAIT.1 ACT.2))
```

Note that a certain subtlety of timing is overlooked in the above example where John offers the book. A sentence of the form "*X* does something, but *Y* does something else", in general is taken to mean that *X*'s actions and *Y*'s actions occur in parallel.

This means that the statement "He offers the book to Mary rudely, but she refuses it." is turned into a timing block like this:

```
(PARALLEL OFFER.1 REFUSE.1)
```

which unfortunately misses the notion that people don't really refuse things until after the offer takes place.

One solution to this would be to make the timing extractor just a little bit smarter. The difference in timing to be extracted from these two sentences can be deduced by special knowledge about the causal relationships among the actions:

- "John walks to the perfume counter but Mary heads for the toy department."
- "John offers the book but Mary refuses it."

In the first example, two acts of walking are implied, so they are assumed to occur in parallel. In the second example, since the refusal is *caused* by the offer, the refusal should be delayed by a certain amount. With more knowledge about which sorts of actions are causally related, the system could make better decisions about the relative timings. Work on story understanding, such as that by Schank, Abelson, and Haase, promise to be able to provide just that sort of knowledge. As it is, since the system has no such knowledge, so it does the best it can. As we shall see in the next chapter, a special rule can be written to detect such occurrences using a pattern-matcher on the timeline and rearranging the events and their timings as necessary.

As the user types in more sentences, the timeline is filled out accordingly with a high-level description of the events in the scene. Finally, the description is complete, and it's time to actually generate an animation from it. The next chapter describes how this is done.

7 *Refinement*

Once a scene has been described as a timeline of high-level actions to be performed, it must be *refined* into progressively more specific actions. Ultimately, the timeline is reduced to a collection of motor skills.

Refinement is basically doing the same thing as what a compiler does when it compiles a program written in a high-level language into machine code. The motor skills can be thought of as machine instructions, which will only execute properly if invoked in the proper sequence and with the proper parameters. The refinement process described in this chapter is just one of many possible ways to do this compilation. Other algorithms can probably be found which are more efficient, but I just chose this one because it was simple and relatively straightforward to implement, while still allowing for some flexible control over the refinement process.

Refining vs. planning

The term *planning* is often used in the AI literature to describe this process, but I chose *refinement* because I want to emphasize that most of the plan is in the mind of the director (the user), not the actors. Divadlo's characters are not so autonomous that they will spontaneously improvise goals and plots - that remains a task for another system, such as Talespin [Meehan]. A command like "Barbara goes to the fax machine and waits patiently," is the sort of thing Divadlo can handle. But a command like "Colin tries to impress Cheryl" is not, unless a lot more knowledge were added about what sorts of things making a good impression entails.

Planning systems are usually concerned with complex ordering of tasks, where each task is typically an abstract operation, in many cases without any parameters to modify its execution. By contrast, the temporal order of a scene's tasks are pretty much well-specified by the input, and the problem is more one of making decisions about what values to give to each action's many parameters.

Scripts

The refinement of an action is governed by a *script* [Schank] associated with each action, which indicates how it can be decomposed into more specific actions. Scripts are actually written for each class of actions, and are inherited by specific instances of each action. For example, one script exists for the action class `SIT`, and each time a character sits down (e.g. `SIT.1`, `SIT.2`, etc.), a copy of that script is used to refine that action.

This script has two parts; the *resources* which indicate parameters which must be known before any further reasoning can be done, and the *cases*, which describe various ways to carry out the action. For example, here is a typical script describing the action `SIT`:

```
(defscript sit
    ((agent (find-plausible-agent this-action))
     (chair (find-plausible-chair this-action))
     (current-occupant (get-value (my chair)
                                  'occupant)))

 (already-sitting
   :doc ("No need to SIT, - ~a is already sitting down."
         (my agent))
   :test (equalp (my agent) (my current-occupant))
   :script (delete-action this-action))

  (already-occupied
   :doc ("~a currently occupied by ~a, evacuate it"
         (my chair) (my current-occupant))
   :test (and (not (null (my current-occupant)))
              (not (equal (my current-occupant)(my agent))))
   :effects '((occupant ,(my chair) ,(my agent))
```

```
            (posture  ,(my current-occupant) standing)
            (posture  ,(my agent) sitting))
 :script (create-subtasks this-action
          '(serial
            (stand (agent ,(my current-occupant)))
            (go
             (agent ,(my current-occupant))
                 (to ,(make-suggestion '(pick-random-place-nearby
                                         ,(my chair)))))
            (sit (agent ,(my agent)) (chair ,(my chair))))))
(far-away
 :doc ("~a is too far away from ~a, must move closer"
       (my agent) (my chair))
 :test (not (ready-to-sit? (my agent) (my chair)))
 :effects '((occupant ,(my chair) ,(my agent))
            (position ,(my agent)
                      ,(get-value (my chair) 'position))
            (posture  ,(my agent) (sitting ,(my chair))))
 :script (create-subtasks this-action
          '(serial
            (go (agent ,(my agent))
            (where ,(place-in-front-of (my chair))))
            (sit (chair ,(my chair))))))
(normal-case
 :doc ("~a sits on ~a" (my agent) (my chair))
 :test t
 :effects '((occupant ,(my chair) ,(my agent))
            (posture  ,(my agent) (sitting ,(my chair))))
 :script (create-subtasks this-action
           '(sit-ms (agent ,(my agent)) (chair ,(my chair))))))
```

The first element of the DEFSCRIPT form is a list of *resources*, which in this case are AGENT, CHAIR, and CURRENT-OCCUPANT. These describe, respectively, the agent who's doing the sitting, the chair he's supposed to sit on, and the current occupant of that chair.

The other forms describe specific *cases* of sitting, such as having to walk over to a distant chair, or waiting for the current occupant to vacate it.

Resources

Each instance of an action attempts to bind specific values for these resources. For example, if the sentence "John sits on the red chair" creates the action `SIT.3`, the resources can be computed as follows:

```
AGENT ⇒ JOHN
CHAIR ⇒ DESK-CHAIR.1
CURRENT-OCCUPANT ⇒ NIL
```

Presuming that nobody is currently sitting on `DESK-CHAIR.1`, we'll say that the value of `CURRENT-OCCUPANT` is `NIL`.

Two special notations are valid within the `DEFSCRIPT` form. The symbol `THIS-ACTION` is interpreted to be the current action, so in this case it stands for `SIT.3` wherever it appears. Also, the macro form `(MY` *`symbol`*`)` refers to the resource named *symbol* within this action. For example, `(MY AGENT)` refers to `JOHN`.

The syntax of `DEFSCRIPT` for declaring the resources is similar to the the syntax of the `LET` form in Lisp. Each declaration is of the form `(`*`varname form`*`)`, which declares a resource named *varname*. Unlike `LET`, however, the *form* is only evaluated as needed to compute the resource's value. As we shall see, resources may be explicitly specified when an action is created, so there's no need to evaluate this form in order to try computing a value.

For convenience, functions like FIND-PLAUSIBLE-AGENT and FIND-PLAUSIBLE-TARGET, etc. exist which try to identify the AGENT or TARGET, etc. of an action. These follow a few simple heuristics, such as looking for the object and subject of the sentence, respectively. They also perform a little type-checking, for example, to make sure the AGENT is an animate object, etc.

Deferred choices

In some cases, the value of a resource may not be computed with the currently available information. For example, the choice of `CHAIR` for `SIT.3` might be dependent on the choice of `CHAIR` for `SIT.2`, and if that hasn't been chosen yet, no decision for the former should be made. This feature was inspired by a similar feature in ANI [Kahn], which allowed the system to gather evidence for making judgments about parameter values.

In this case, the form which is evaluated to compute a resource may opt to return a special kind of object called a deferred-choice, or *choice* for short.

A *choice* object consists of the following fields:

- `name` – the name of this choice
- `action` – the action in which this choice is made
- `resource` – the resource for which this choice is made
- `suggestions` – suggestions for making this choice
- `depends-on` – other values this choice depends on
- `conflicts` – mutually exclusive choices
- `ok-test` – a predicate indicating whether this choice is ready to be made
- `doit` – a form for determining a value for this choice
- `args` – any additional relevant arguments which should be supplied to the `ok-test` and `doit` forms

The purpose of a choice object is to capture all the information necessary to defer making a choice, to know when the choice cannot be put off any longer, and to choose a particular value when called upon to do so.

The `action` and `resource` fields refer to the particular resource in the particular action which decides to defer the choice by making the choice object.

The `suggestions`, `depends-on`, `conflicts`, and `args` fields are lists of arbitrary lisp forms, but are usually lists of symbols or units referring to other

resources, actions, or units.They are used by the `ok-test` and `doit` forms in their computations.

When a choice is deferred, it is placed on a global queue called `*DEFERRED-CHOICES*`. A choice can only be deferred if there are no reasons for it *not* to be deferred. A choice may not be deferred if doing so would get the system stuck, so a couple of mechanisms are used to keep this from happening. For example, the `depends-on` relation between choices form a directed graph, in which a cycle among deferred choices could cause the system to wait forever for one of the choices to be made. So, before a choice is deferred, this condition is checked. Similarly, the `conflicts` field explicitly indicates which choices cannot be deferred along with a given choice. Deferral of a choice is denied of one of these conflicting choices has already been deferred.

If deferral of a choice has been denied, that means the system cannot afford to put off making that choice any longer; it must be made *now*. A choice can also be made if its `OK-TEST` evaluates to *true*. The `OK-TEST` is a function which, when passed `ARGS`, determines whether sufficient information is currently available to compute a value for the choice. If it fails, the choice remains deferred.

When a choice must be made, either because `OK-TEST` succeeded or because deferral was denied, the function stored in `DOIT` is invoked with arguments `ARGS`. This returns a value, which becomes the new value of the resource named `RESOURCE` in the action `ACTION`.

The cases of a script

After the resources of an action are determined, a particular *case* is selected to carry out the action. The script contains one or more cases, describing different possible conditions which may affect the performance of that action. A case consists of the following parts:

- `name` – the name of this case
- `documentation` – a form used to describe this case
- `test` – a form which indicates if this case is applicable
- `effects` – a series of assertions summarizing the consequences of performing this case.
- `script` – a form which implements this case by refining a portion of the time-line.

To illustrate this, here is a typical case from the script for `SIT`:

```
(far-away
  :doc ("~a is too far away from ~a, must move closer"
        (my agent) (my chair))
  :test (not (ready-to-sit? (my agent) (my chair)))
  :effects '((occupant ,(my chair) ,(my agent))
             (position ,(my agent)
                       ,(get-value (my chair) 'position))
             (posture  ,(my agent) (sitting ,(my chair))))
  :script (create-subtasks this-action
            '(serial
              (go (agent ,(my agent))
                  (where ,(place-in-front-of (my chair))))
              (sit (chair ,(my chair))))))
```

`FAR-AWAY` is the name of this case. It's convenient to have a name to refer to this case for debugging, and as we shall see later, for detecting infinite loops when the same case is invoked more than once for the same action.

`:DOC` is the documentation form. The value is a list which is given to `FORMAT`, a standard Lisp function for computing strings. The resulting string is used for debugging and is printed when tracing the intermediate steps of refinement.

`:TEST` indicates whether this case is applicable, i.e. whether `AGENT` is in fact too far away from `CHAIR` to sit down directly on it. It calls the function `READY-TO-SIT?` to ascertain whether `AGENT`'s location is directly in front of `CHAIR`, facing away from it. Each of the cases' tests are evaluated in turn, like a Lisp `COND` form. The first case whose `TEST` is true is the one that is chosen.

:EFFECTS is a list of assertions indicating the effects of performing this case. This is handy for reasoning about the consequences of actions, much like the add-lists and delete-lists of traditional planners.

:SCRIPT is a form that actually refines the action. One common script function is DELETE-ACTION, which is used when the current action is deemed superfluous, and is simply eliminated. This happens when, for example, the action calls for an agent to sit down on a given chair when that agent is *already* sitting in that chair.

Another common, and important, script function is CREATE-SUBTASKS. It causes the action to be replaced by one or more *subtasks*. These are organized in the same serial/parallel structure as the original timeline itself (see page 45). Each subtask is specified by a unit indicating an action class, optionally followed by resources.

The subtask is created by making a new instance of the action class, and binding the appropriate resource values if supplied. For example, in the script shown above, two new subtasks are created, one instance of GO, and another of SIT. A unique number is assigned to each instance upon creation, so SIT.3 might have subtasks named GO.1 and SIT.4.

An action and its subtasks are linked by mutual pointers: the SUBTASKS slot of SIT.3 is set to (GO.1 SIT.4), and conversely, the SUBTASK-OF slot for each of the subtasks is set to SIT.3.

The original action, SIT.3, is removed from the timeline, and is no longer considered to be one of the actions in the current scene. In its place, the sequence (SERIAL GO.1 SIT.4) is spliced in. In this way, the timeline for the scene always indicates the current state of the refinement, without being cluttered up by the earlier, higher-level action. But these higher-level actions are still accessible, via the SUBTASK-OF links.

If the scripts are written properly, the new action, like SIT.4, will in some sense be more refined, or closer to achieving the goal, than the original action, SIT.3. In this case, SIT.4 is simply the act of sitting down,

whereas the higher-level action SIT.3 includes both walking over to the chair and sitting down.Each act of refinement is arguably making progress toward the goal of reducing the actions to motor skills.

However, there is always the chance that a bug in the way the scripts are written can give rise to an infinite recursion. For example, if SIT.4 were no different from SIT.3, and for some reason, SIT.4 also required *its* subtasks to be GO.2 and SIT.5, ad infinitum. It turns out to be pretty easy to detect this sort of infinite recursion. Whenever a case is selected which results in the creation of a new subtask, the ancestry of that action is checked by walking up the SUBTASK-OF links for an identical ancestor. An action is considered identical to an ancestor if the resources have the same values, and the same case was chosen for refinement. If this is happened, a "too much recursion" error is signalled, and creation of the new subtask is skipped.

Global optimizations

One way to think of refining an action is that it is the application of rules to rewrite that action. Usually, as described above, this occurs when one action is the focus of consideration.

However, certain patterns may arise in a scene where two or more actions may be optimized as a group by applying similar rewrite rules. These so-called global rules consist of two parts:

- a pattern which serves to detect cases when the rule is applicable
- the consequences of the rule

For simplicity, I implemented both parts of global optimization rules as arbitrary lisp functions to be called on the the timeline. Typical optimizations may involve cases like these:

- A sequence of actions may be replaced by a more specific action. For example, one might wish to replace a sequence of RUN and KICK-BALL with a specialized action called RUN-WHILE-KICKING-BALL.

- The ordering of actions might be rearranged because of interactions among them. For example (as shown in the example in chapter 8, page 105), the parallel occurence of two causally related actions, like OFFER and REFUSE, might be modified by delaying the REFUSE action. This is most easily done by inserting a new WAIT action which causes nothing to happen for an appropriate interval.
- The resources of some action might be modified because of some other element of the scene. For example, if some actor is told to perform the same action three times in a row, he might choose to perform it differently somehow on the third try, just to avoid monotony. Changing some resource of the third SIT action would cause this difference.
- Opportunistic behavior could call for a rewrite of the actions if some opportunity arose during the scene. For example, an actor who WALKs past a drinking fountain might choose to stop for a drink.

All the global optimization rules are periodically given a chance to examine the timeline to see if any of them are applicable, and if so, they are immediately applied. As with CREATE-SUBTASKS, the possibility of infinite recursion is avoided by annotating actions when they are rearranged or created by such a rule. These annotations are examined to see if some sort of infinite loop threatens to arise.

Perhaps the most interesting kind of refinement comes from introducing new actions for other reasons than subtasking. The character's personality or mood may indicate that some actions ought to be inserted or modified, for purely aesthetic reasons. For example, a character with a cold may perform a SNEEZE action during the scene, even though it was not explicitly called for. A GRUMPY character may thump the table as he passes it (and so on for DOPEY, SLEEPY, etc.)

The refinement process

In order to refine the actions in the scene, several processes occur:

- values for the resources of each action must be determined
- deferred choices must be acted on when possible
- the applicable case for each action's script must be determined
- the action must be refined by applying that case
- global optimizations must be applied where needed

It is not clear what is the best ordering for all of these process to occur, especially for a serial implementation. Some sort of time-sharing mechanism is needed to run each in turn, to give them all a fair chance.

I chose not to do anything too exotic here; a simple algorithm seems to work sufficiently well.

1. First, resolve all the "obvious" resources
2. Apply global optimizations, if any
3. Refine the next unrefined action
   a. Attempt to bind all its resources.
   b. If any resources are deferrable choices, defer considering this action
   c. Otherwise, apply the script
4. Attempt to resolve any deferred choices
5. Repeat until done

The *next* action in step 3 is found by simply walking the timeline tree and finding the first action that needs refining.

If an unresolvable error occurs, it is reported to the user, along with an explanation. Typically this means the system is incapable of complying with all the commands given by the user, who has to retract or modify something before trying again.

The refinement process is *done*, and rendering can begin, when

- All remaining actions are actually motor skills.
- All required parameters (resources) for these units have specific values.

No choices are pending on the deferral list.

8 *An Example*

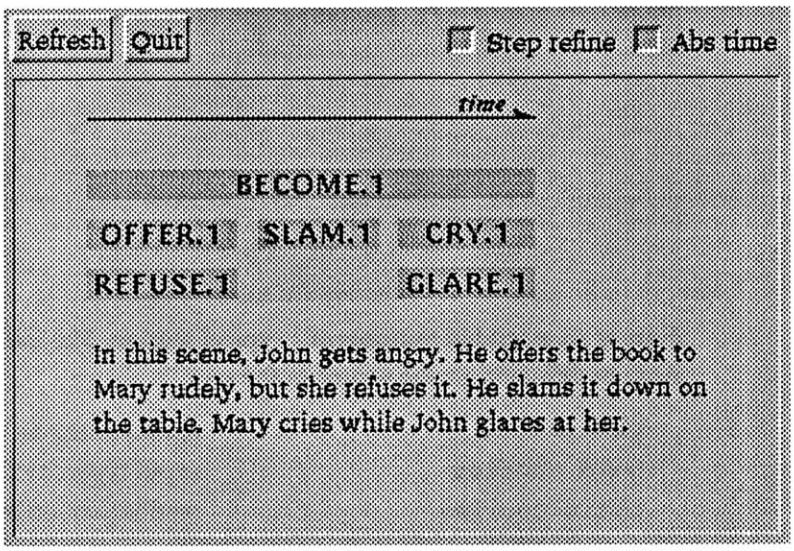
Here is an extended example of a typical session. The following fonts are used to differentiate the source of the text:

plain palatino	commentary on the example
`bold courier`	input typed by the user
`plain courier`	results printed by the system

In this example I will explain how Divadlo sets up and generates a brief animation from the following input:

> *New scene. The characters in this scene are John and Mary. John's shirt is green. He's left-handed. There's a desk at stage left. It's facing center stage. There's a NeXT on the desk. There's a swivel chair in front of the desk. The chair's facing center stage. Mary is sitting on it. There's a table upstage right. There's a dark red desk chair in front of the table. The upstage chair's facing Mary. John is sitting on the red chair. There's a purple book on the table.*
>
> *In this scene John gets angry. He offers the book to Mary rudely, but she refuses it. He slams it down on the table. Mary cries while John glares at her.*

As Divadlo processes this, the state of the timeline is normally displayed in a separate graphics window, as shown below. This "timeline" display is updated throughout the refinement process, but it would take up too much space to include all those screens here. Instead, I have reproduced the content of the timeline display below as a small diagram.



The first step is to run the function `TOP-LEVEL-LOOP`, which enters the parser.

```
(top-level-loop)
p> New scene.
 Creating a new scene: SCENE.1
```

The characters are defined. The words "John" and "Mary" are entered into the lexicon as male and female first names, so the shapes of male and female characters will be loaded from a file.

```
p> The characters in this scene are John and Mary.
Loading structure for MEN...
```

The shapes are read into lisp from the shape file library. This takes about a minute for all the shapes in a complex figure like a person, or a few seconds for something simple like a desk lamp. The name of each part is printed to indicate the progress.

```
 Reading guy
 Reading torso
 Reading neck
```

```
Reading head
Reading face
Reading hair
Reading Reye
Reading Leye
Reading nose
Reading lips
Reading Ruarm
Reading Rlarm
Reading Rhand
Reading Rthumb1
Reading Rthumb2
Reading Rthumb3
Reading Rindex1
Reading Rindex2
Reading Rmiddle1
Reading Rmiddle2
Reading Rring1
Reading Rring2
Reading Rpink1
Reading Rpink2
Reading Luarm
Reading Llarm
Reading Lhand
Reading Lthumb1
Reading Lthumb2
Reading Lthumb3
Reading Lindex1
Reading Lindex2
Reading Lmiddle1
Reading Lmiddle2
Reading Lring1
Reading Lring2
Reading Lpink1
Reading Lpink2
Reading Rthigh
Reading Rshin
Reading Rfoot
Reading Lthigh
Reading Lshin
Reading Lfoot
```

```
Loading structure for WOMEN...
 Reading gal
 Reading torso
 Reading neck
 Reading head
 Reading face
 Reading hair
 Reading Reye
 Reading Leye
 Reading nose
 Reading lips
 Reading Ruarm
 Reading Rlarm
 Reading Rhand
 Reading Rthumb1
 Reading Rthumb2
 Reading Rthumb3
 Reading Rindex1
 Reading Rindex2
 Reading Rmiddle1
 Reading Rmiddle2
 Reading Rring1
 Reading Rring2
 Reading Rpink1
 Reading Rpink2
 Reading Luarm
 Reading Llarm
 Reading Lhand
 Reading Lthumb1
 Reading Lthumb2
 Reading Lthumb3
 Reading Lindex1
 Reading Lindex2
 Reading Lmiddle1
 Reading Lmiddle2
 Reading Lring1
 Reading Lring2
 Reading Lpink1
 Reading Lpink2
 Reading Rthigh
 Reading Rshin
 Reading Rfoot
```

```
 Reading Lthigh
 Reading Lshin
 Reading Lfoot
 Adding actors to the scene: (JOHN MARY)
p> John's shirt is green.
 OK.
```

Nothing has been displayed yet, so this is changed only in the internal representation for now.

```
p> He's left-handed.
 Assuming 'HE' is JOHN.
 OK.
```

Notice the use of a pronoun and the contracted form of the verb "is". This makes the commands seems less stilted and easier to type.

```
p> There's a desk at stage left.
 Introducing a new object: DESK.1
```

This statement serves two purposes. It calls for the creation of a new desk object, which must be loaded from the shape library. It also indicates the desk's location within the scene. The general class of objects is `DESKS`, and a specific instance, called `DESK.1`, is created.

```
Loading structure for DESKS...
 Reading desk
 Reading R
 Reading RLfront
 Reading RLpull
 Reading RUfront
 Reading RUpull
 Reading Rleg4
 Reading Rleg3
 Reading Rleg2
 Reading Rleg1
 Reading L
 Reading LLfront
 Reading LLpull
 Reading LUfront
 Reading LUpull
```

```
Reading Lleg4
Reading Lleg3
Reading Lleg2
Reading Lleg1
Adding an object to the scene: DESK.1
```

Once the desk has been loaded, it must be moved to stage left. Since geometric information is stored redundantly in both the Lisp system and the 3d display subsystem (Rendermatic), this information is passed along to Rendermatic so its records will remain consistent.

However, since the system notices that Rendermatic has not yet been initialized, it automatically starts up a new Rendermatic process. Once it's up and running, the desk's shape definition is downloaded as a necessary prerequisite for moving it to stage left.

```
One minute while I start up Rendermatic...
Loading DESK.1 into rendermatic.
desk.1...
L...
Lleg1...
Lleg2...
Lleg3...
Lleg4...
LUfront...
LUpull...
LLfront...
LLpull...
R...
Rleg1...
Rleg2...
Rleg3...
Rleg4...
RUfront...
RUpull...
RLfront...
RLpull...
OK.
p> It's facing center stage.
Assuming 'IT' is DESK.1.
OK.
```

The desk is now rotated from its default orientation.

```
p> There's a NeXT on the desk.
```

Like the desk, the NeXT computer is instanced, loaded, and relocated. The prepositional phrase "on the desk" indicates where it should be placed. By default, the bounding box of the NeXT and the bounding box of the desk would be compared, and the two objects would be stacked like simple blocks.

However, we take advantage of the ability to treat certain objects specially by overriding the default mechanism. In this case, we place the monitor (and keyboard and mouse) on the desk using the normal bounding-box algorithm, but an auxiliary piece of code teleports the cpu cube to the floor beside the object in the prepositional phrase (in this case, the desk).

```
Introducing a new object: NEXT.1
Loading structure for NEXT-COMPUTERS...
Reading NeXT
Reading Rfoot
Reading Lfoot
Reading Object25
Reading stand
Reading tilt
Reading moncase
Reading tube
Reading grid
Reading bezel
Reading screen
Reading kbd
Reading fkey
Reading querty
Reading num
Reading mouse
Reading cube
Adding an object to the scene: NEXT.1
Loading NEXT.1 into rendermatic.
next.1...
cube...
kbd...
```

```
 mouse...
 num...
 querty...
 fkey...
 Object25...
 tilt...
 moncase...
 tube...
 bezel...
 screen...
 grid...
 stand...
 Lfoot...
 Rfoot...
 OK.
p> There's a swivel chair in front of the desk.
 Introducing a new object: SWIVEL-CHAIR.1
```

There's actually three different shapes of chairs in the shape library, so the shape for "swivel chairs" is loaded.

```
Loading structure for SWIVEL-CHAIRS...
 Reading chair2
 Reading leg1
 Reading Object15
 Reading castor1
 Reading Object17
 Reading leg2
 Reading Object19
 Reading castor2
 Reading Object21
 Reading leg3
 Reading Object2
 Reading castor3
 Reading Object4
 Reading leg4
 Reading Object7
 Reading castor4
 Reading Object9
 Reading leg5
 Reading Object
 Reading castor5
```

```
Reading Object
Reading stem
Reading seat
Reading Object
Reading Object
Reading back
Reading Object
Adding an object to the scene: SWIVEL-CHAIR.1
Loading SWIVEL-CHAIR.1 into rendermatic.
swivel-chair.1...
stem...
seat...
Object...
Object...
back...
Object...
leg5...
Object...
castor5...
Object...
leg4...
Object7...
castor4...
Object9...
leg3...
Object2...
castor3...
Object4...
leg2...
Object19...
castor2...
Object21...
leg1...
Object15...
castor1...
Object17...
OK.
p> The chair's facing center stage.
OK.
p> Mary is sitting on it.
Assuming 'IT' is SWIVEL-CHAIR.1.
```

Now that we have specified the position and posture for Mary, we have to load her shape into Rendermatic.

```
Loading MARY into rendermatic.
mary...
Lthigh...
Lshin...
Lfoot...
Rthigh...
Rshin...
Rfoot...
torso...
Luarm...
Llarm...
Lhand...
Lpink1...
Lpink2...
Lring1...
Lring2...
Lmiddle1...
Lmiddle2...
Lindex1...
Lindex2...
Lthumb1...
Lthumb2...
Lthumb3...
Ruarm...
Rlarm...
Rhand...
Rpink1...
Rpink2...
Rring1...
Rring2...
Rmiddle1...
Rmiddle2...
Rindex1...
Rindex2...
Rthumb1...
Rthumb2...
Rthumb3...
neck...
head...
```

```
 face...
 lips...
 nose...
 Leye...
 Reye...
 hair...
 OK.
p> There's a table upstage right.
 Introducing a new object: TABLE.1
Loading structure for TABLES...
 Reading table
 Adding an object to the scene: TABLE.1
 Loading TABLE.1 into rendermatic.
 table.1...
 OK.
p> There's a dark red desk chair in front of the table.
```

Desk chairs are blue by default, but by asking for a red one, we get a red one.

```
 Introducing a new object: DESK-CHAIR.1
 Loading structure for DESK-CHAIRS...
 Reading chair3
 Reading fr
 Reading back
 Reading Larm1
 Reading Larm2
 Reading Rarm1
 Reading Rarm2
 Reading LRleg
 Reading LFleg
 Reading RRleg
 Reading RFleg
 Adding an object to the scene: DESK-CHAIR.1
 Loading DESK-CHAIR.1 into rendermatic.
 desk-chair.1...
 fr...
 RFleg...
 RRleg...
 LFleg...
 LRleg...
```

```
 Rarm1...
 Rarm2...
 Larm1...
 Larm2...
 back...
 OK.
p> The upstage chair's facing Mary.
 OK.
```

Note that in addition to using pronouns, we can refer to an object by its category ("desk chair"), its color ("the red chair"), or its location ("the upstage chair"). This helps makes referencing objects more convenient and natural.

```
p> John is sitting on the red chair.
 Loading JOHN into rendermatic.
 john...
 Lthigh...
 Lshin...
 Lfoot...
 Rthigh...
 Rshin...
 Rfoot...
 torso...
 Luarm...
 Llarm...
 Lhand...
 Lpink1...
 Lpink2...
 Lring1...
 Lring2...
 Lmiddle1...
 Lmiddle2...
 Lindex1...
 Lindex2...
 Lthumb1...
 Lthumb2...
 Lthumb3...
 Ruarm...
 Rlarm...
 Rhand...
 Rpink1...
 Rpink2...
```

```
 Rring1...
 Rring2...
 Rmiddle1...
 Rmiddle2...
 Rindex1...
 Rindex2...
 Rthumb1...
 Rthumb2...
 Rthumb3...
 neck...
 head...
 face...
 lips...
 nose...
 Leye...
 Reye...
 hair...
 OK.
p> There's a purple book on the table.
 Introducing a new object: BOOK.1
Loading structure for BOOKS...
 Reading book
 Adding an object to the scene: BOOK.1
 Loading BOOK.1 into rendermatic.
 book.1...
 OK.
p> Render.
```

This actually renders the scene so we can take a look at it. Any objects in the scene that still need to be loaded at this point are loaded automatically. The floor is defined to be in every scene, even though the director didn't explicitly call for one, so it has to be loaded at this point.

```
 Loading structure for FLOORS...
 Reading floor
 Loading FLOOR into rendermatic.
 floor...
 OK.
```

Now that we have defined the initial conditions for the scene, it's time to describe the action that will take place in it.

```
p> In this scene, John gets angry.
 New action: BECOME.1
```

BECOME.1 is the action of "getting angry". Its duration is defined to occupy the entire duration of the scene, because of the phrase "in this scene." Since we don't know how long that is yet, exactly, we add all the other actions in parallel to BECOME.1.

```
p> He offers the book to Mary rudely, but she refuses it.
 New actions: (OFFER.1 REFUSE.1)
```

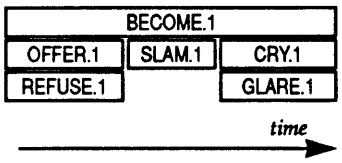
Two new actions, OFFER.1 and REFUSE.1 are added to the scene. Because of the way the sentence is structured (*A* does *X*, but *B* does *Y*), the two actions occur in parallel.

```
p> He slams it down on the table.
 New action: SLAM.1
```

We assume that the new action, SLAM.1, takes place after the previous two actions.

```
p> Mary cries while John glares at her.
 New actions: (CRY.1 GLARE.1)
```

That's all the action for this scene. At this point, we can look at the timeline display, which shows us something like this:



We will now start to refine the scene, as described in chapter 7. First off, we try to nail down all the resources whose values are obvious, or trivially derived from the input. For example, it's easy to compute the AGENT for most of the actions, since that's explicitly given by the parse.

```
> (refine-scene)
 Resource AGENT of BECOME.1 set to: JOHN
 Resource AGENT of OFFER.1 set to: JOHN
```

```
Resource OBJECT of OFFER.1 set to: BOOK.1
Assuming 'SHE' is MARY.
Resource AGENT of REFUSE.1 set to: MARY
Assuming 'IT' is BOOK.1.
Resource OBJECT of REFUSE.1 set to: BOOK.1
Resource AGENT of CRY.1 set to: MARY
```

Now we go back and do the more difficult resources. The MOOD of BECOME.1 can be computed, but the DURATION cannot be determined yet. We will *defer* this choice until more information becomes available, and process some other actions in the meantime.

```
Resource MOOD of BECOME.1 set to: ((MOOD ANGRY))
Resource DURATION of BECOME.1 set to: #<CHOICE
  CHOOSE-BECOME-DURATION 1082796142>
```

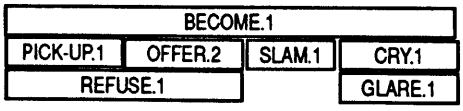
The next action to be processed is OFFER.1.

```
Resource TARGET of OFFER.1 set to: MARY
Resource MOOD of OFFER.1 set to: ((RUDE))
Refining OFFER.1: JOHN is not holding BOOK.1. Going
  to pick it up.
```

Since all the resources for OFFER.1 are known, we can now select a script to carry it out. Since John isn't holding the book, he has to go pick it up first.

```
Replacing OFFER.1 with PICK-UP.1 and OFFER.2.
```

Notice that OFFER.1 has been replaced with two new skills, PICK-UP.1 and OFFER.2. This leaves us with a timeline that looks like this:



We now proceed with refining these.

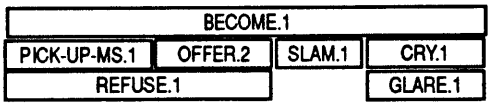
```
Resource AGENT of PICK-UP.1 set to: JOHN
Resource OBJECT of PICK-UP.1 set to: BOOK.1
Resource AGENT of OFFER.2 set to: JOHN
```

```
Resource OBJECT of OFFER.2 set to: BOOK.1
```

One of the resources that PICK-UP needs to know is which hand to use. Several potential sources of information are considered but eliminated. Finally, we are left with the clue that John is left-handed, so that's the value we use.

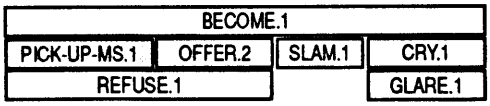
```
John's a lefty, so we'll use his left hand.
Resource HAND of PICK-UP.1 set to: LEFT
Refining PICK-UP.1: JOHN picks up BOOK.1.
```

PICK-UP.1 is now ready for refinement. It is replaced with the motor skill which actually moves the arm, PICK-UP-MS.1.




```
Resource AGENT of PICK-UP-MS.1 set to: JOHN
Resource OBJECT of PICK-UP-MS.1 set to: BOOK.1
Resource HAND of PICK-UP-MS.1 set to: LEFT
Replacing PICK-UP.1 with PICK-UP-MS.1.
Refining PICK-UP-MS.1: Pick-up motor skill - JOHN grabs
BOOK.1 with his left hand.
Resource MOOD of PICK-UP-MS.1 set to: NIL
Resource DURATION of PICK-UP-MS.1 set to: 1
```

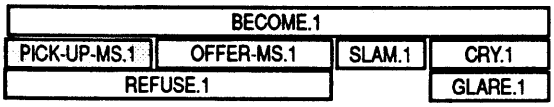
Finally, all the resources of PICK-UP-MS.1 are computed. Since it is a motor skill, and its resources are known, we no longer need to consider it for any further refinement. It is shaded to indicate that it is completely refined.



Now we consider OFFER.2. It too can be reduced to a motor skill, OFFER-MS.1.

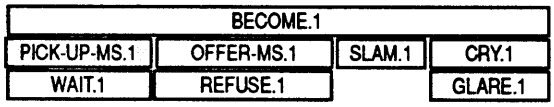
```
Resource TARGET of OFFER.2 set to: MARY
Resource MOOD of OFFER.2 set to: ((RUDE))
Refining OFFER.2: JOHN offers BOOK.1
```

```
Replacing OFFER.2 with OFFER-MS.1.
```

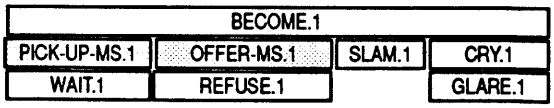


There is a global optimization rule (see page 102) which is applicable in this situation. It states that whenever a refusal occurs in parallel with an offer-motor-skill, the refusal is delayed until after the offer starts. The test for this rule has finally become true, so it is applied before any further refinement is done.

The effect is to insert a `WAIT` action (which does nothing), before `REFUSE.1`. Choosing the duration of `WAIT.1` is deferred, until more information can be gathered from `OFFER-MS.1`.



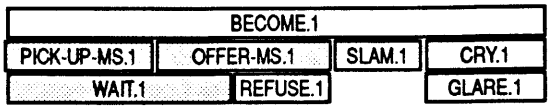

```
Refining OFFER-MS.1: Offer motor skill - JOHN
    offers BOOK.1
Resource AGENT of OFFER-MS.1 set to: JOHN
Resource OBJECT of OFFER-MS.1 set to: BOOK.1
Resource TARGET of OFFER-MS.1 set to: MARY
Resource MOOD of OFFER-MS.1 set to: ((RUDE))
Resource DELAYED-BY of REFUSE.1 set to: WAIT.1
Resource HAND of OFFER-MS.1 set to: LEFT
Resource DURATION of OFFER-MS.1 set to: 1
```



Now that `OFFER-MS.1` is refined, we can refine `WAIT.1` too.

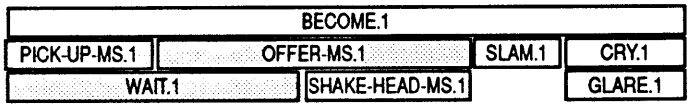
```
Refining WAIT.1: Just waiting.
```

```
Resource DURATION of WAIT.1 set to: 1.5
```

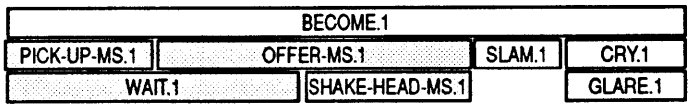


Now we refine `REFUSE.1`. We don't know anything about Mary's mood, either from the story, or from the description of Mary. The script for `REFUSE` says that the default way to refuse something is to shake one's head.

```
Resource MOOD of REFUSE.1 set to: NIL
Refining REFUSE.1: MARY just says no.
Replacing REFUSE.1 with SHAKE-HEAD-MS.1.
```

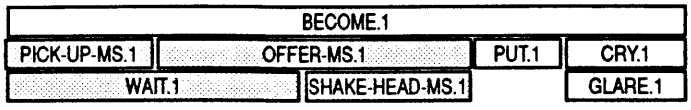



```
Refining SHAKE-HEAD-MS.1: Shake-Head motor skill
- MARY shakes her head.
Resource AGENT of SHAKE-HEAD-MS.1 set to: MARY
Resource OBJECT of SHAKE-HEAD-MS.1 set to: BOOK.1
Resource MOOD of SHAKE-HEAD-MS.1 set to: NIL
Resource AMOUNT of SHAKE-HEAD-MS.1 set to: 45
Resource DURATION of SHAKE-HEAD-MS.1 set to: 1
```

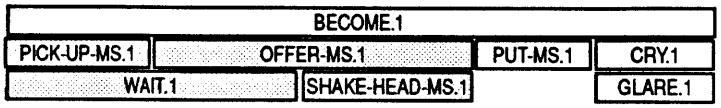



```
Refining SLAM.1
Replacing SLAM.1 with PUT.1.
Resource MOOD of PUT.1 set to: ((ANGRY VERY))
```

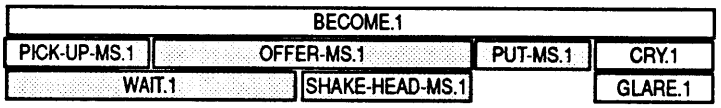
`SLAM` is actually a special case of `PUT`, only the `MOOD` is set to `VERY ANGRY`. So we replace `SLAM.1` with `PUT.1`.



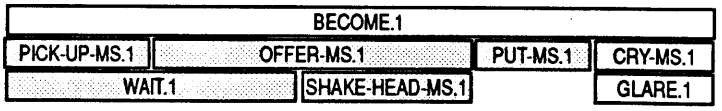
```
Refining PUT.1: JOHN puts BOOK.1 down onto TABLE.1
Assuming 'HE' is JOHN.
Resource AGENT of PUT.1 set to: JOHN
Assuming 'IT' is BOOK.1.
Resource OBJECT of PUT.1 set to: BOOK.1
Resource WHERE of PUT.1 set to: TABLE.1
Replacing PUT.1 with PUT-MS.1.
```




```
Refining PUT-MS.1: Put motor skill - JOHN puts
   BOOK.1 down on TABLE.1
Resource AGENT of PUT-MS.1 set to: JOHN
Resource OBJECT of PUT-MS.1 set to: BOOK.1
Resource WHERE of PUT-MS.1 set to: TABLE.1
Resource MOOD of PUT-MS.1 set to: ((ANGRY VERY))
Resource HAND of PUT-MS.1 set to: LEFT
Resource DURATION of PUT-MS.1 set to: 0.5
```

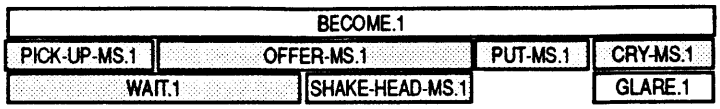



```
Refining CRY.1: MARY breaks out crying. I hope you're
satisfied.
Resource AGENT of CRY-MS.1 set to: MARY
Replacing CRY.1 with CRY-MS.1.
```




```
Refining CRY-MS.1: Cry motor skill - MARY
   starts sobbing.
```

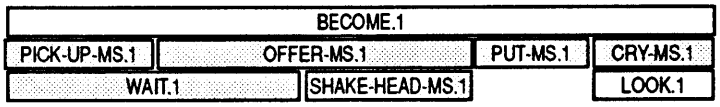
```
Resource DURATION of CRY-MS.1 set to: 1
```




```
Refining GLARE.1
```

Like SLAM, GLARE is just an ANGRY version of LOOK.

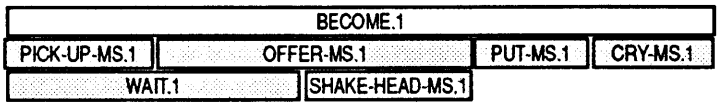
```
Replacing GLARE.1 with LOOK.1.
```




```
Resource MOOD of LOOK.1 set to: ((ANGRY VERY))
Resource AGENT of LOOK.1 set to: JOHN
Assuming 'SHE' is MARY.
Resource TARGET of LOOK.1 set to: MARY
Resource ANGLE of LOOK.1 set to: 0.0
```

Since John is already looking at Mary, there's no need for LOOK.1 to do anything, so it is deleted from the script.

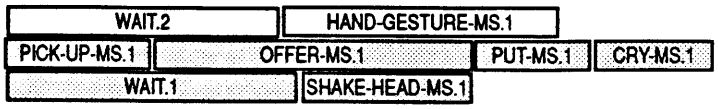
```
Deleting LOOK.1.
```



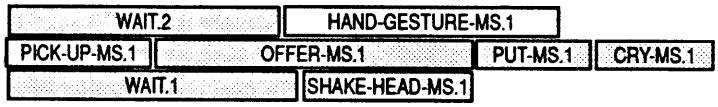
Now the duration of BECOME.1 can be computed at last, because the timing block it belongs to (the actions in PARALLEL with it) has a specific duration.

```
Refining BECOME.1: JOHN gets madder.
Resource DURATION of BECOME.1 set to: 4.0
Replacing BECOME.1 with WAIT.2 and HAND-GESTURE-MS.1.
```

The way someone can get mad is to wait one third of the time, and then clench one's fist for a third of the time.

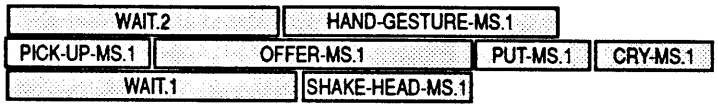



```
Refining WAIT.2: Just waiting.
Resource DURATION of WAIT.2 set to: 1.3333333333333333
```




```
Resource AGENT of HAND-GESTURE-MS.1 set to: JOHN
Resource GESTURE of HAND-GESTURE-MS.1 set to: FIST
Resource DURATION of HAND-GESTURE-MS.1 set to:
    1.3333333333333333
Resource HAND of HAND-GESTURE-MS.1 set to: RIGHT
```

Since John only has one free hand, he clenches his RIGHT fist. This completes the last motor skill.



Now that the ordering and durations of all the motor skills is complete, we walk the timeline tree one last time. The START and END times are assigned to each motor skill, in first seconds, then in frames (here we are assuming 30 frames per second).

```
Resource START of WAIT.2 set to: 0
Resource START-FRAME of WAIT.2 set to: 0
Resource END of WAIT.2 set to: 1.3333333333333333
Resource END-FRAME of WAIT.2 set to: 39
Resource START of HAND-GESTURE-MS.1 set to:
1.3333333333333333
Resource START-FRAME of HAND-GESTURE-MS.1 set to: 40
```

```
Resource END of HAND-GESTURE-MS.1 set to: 2.6666666666666665
Resource END-FRAME of HAND-GESTURE-MS.1 set to: 79
Resource START of PICK-UP-MS.1 set to: 0
Resource START-FRAME of PICK-UP-MS.1 set to: 0
Resource END of PICK-UP-MS.1 set to: 1
Resource END-FRAME of PICK-UP-MS.1 set to: 29
Resource START of OFFER-MS.1 set to: 1
Resource START-FRAME of OFFER-MS.1 set to: 30
Resource END of OFFER-MS.1 set to: 2
Resource END-FRAME of OFFER-MS.1 set to: 59
Resource START of WAIT.1 set to: 0
Resource START-FRAME of WAIT.1 set to: 0
Resource END of WAIT.1 set to: 1.5
Resource END-FRAME of WAIT.1 set to: 44
Resource START of SHAKE-HEAD-MS.1 set to: 1.5
Resource START-FRAME of SHAKE-HEAD-MS.1 set to: 45
Resource END of SHAKE-HEAD-MS.1 set to: 2.5
Resource END-FRAME of SHAKE-HEAD-MS.1 set to: 74
Resource START of PUT-MS.1 set to: 2.5
Resource START-FRAME of PUT-MS.1 set to: 75
Resource END of PUT-MS.1 set to: 3.0
Resource END-FRAME of PUT-MS.1 set to: 89
Resource START of CRY-MS.1 set to: 3.0
Resource START-FRAME of CRY-MS.1 set to: 90
Resource END of CRY-MS.1 set to: 4.0
Resource END-FRAME of CRY-MS.1 set to: 119
SCENE.1 refined and ready to record.
```

9 *Conclusion*

I hope to have demonstrated that it is possible to teach an animation system enough about emotion and expressive qualities to take over the "acting" part of a desktop theater system, leaving the "directing" part to the user. The simulated actors have the discretion to interpret goals and constraints presented in plain English, choosing their mannerisms and actions accordingly, liberating the user from having to specify their behavior in excrutiating detail.

Animated characters can express their internal moods and intentions by selecting from a palette of *traits*. By changing, augmenting, or suppressing actions as they go about their business, they indicate their motivations to the audience, instead of merely performing a logical sequence of actions that simply accomplish a task in the most functionally optimum way.

In order for an animation system to do these things, it has to represent them somehow. Unlike a conventional animation system, which is basically a CAD system for specifying concrete positions and trajectories of objects, an "intelligent" animation environment can collaborate with the user if it knows more about what's actually going on in the scene. It does this by representing the scene symbolically as well as numerically, keeping track of qualitative relationships as well as the quantitative ones.

In order to describe a scene, the user types commands and receives responses in plain English. This dialog serves to describe the initial conditions and constraints on the objects and characters, as well as describe the action which occurs in the scene. Several features of the parser serve to

make this process convenient, such as the abilities to deal with pronouns, contractions, parallel clauses, etc. The user can also ask questions about the state of various objects, and get answers in English.

Objects and their actions are stored in a library, though extending this library is a task for programmers, not the casual user. Objects may be translated from a variety of popular 3D shape file formats, and may be articulated, i.e. represented as a tree of rigid shapes with flexible joints. Characteristic poses of these objects are themselves stored in a pose library, and combinations of partial poses are supported so that various parts of a figure can be managed independently. This allows, for example, any of several hand gestures to optionally override a default hand pose suggested by another processes which might control the upper body and arm.

The joints themselves are represented in any of several notations, acknowledging the fact that different notations are appropriate for different kinds of joints. Where possible, these notations are interconvertable so that various motion operators, such as interpolation for key-frame animation, or Jacobian computation for inverse kinematics, can be performed on as many parts of various body forms as possible.

Actions themselves are described in the forms of scripts, which represent knowledge about the various ways in which a given high-level action might conceivably be decomposed into more concrete actions. I have chosen to term this process *refinement*, to distinguish it from planning, since the characters do not attempt to solve complex problems on their own. Rather, their task is to interpret and carry out the user's plans.

A refined scene is a script of the sort acceptable to traditional animation software, namely, a timeline of highly specific actions to be performed, with all detailed parameters provided.

What does it mean?

This kind of system is not going to put Disney animators out of a job any time in the conceivable future. Feature films have been, and will perhaps always be made the traditional way, with hundreds of animators carefully guiding every nuance of every gesture for hundreds of thousands of frames. As long as people don't mind spending tens of millions of dollars on a piece of film 100 minutes long that can be shown over and over again for decades, there'll be cheap labor in third-world countries to help make them.

My goal is somewhat different, perhaps best expressed by Brenda Kay Laurel in her dissertation entitled "Toward the Design of a Computer-Based Interactive Fantasy System" [Laurel]. Although she didn't actually build one, she analyzed the issues surrounding systems that would someday engage the user in a simulation of a vivid, dramatic, interactive performance. One of the necessary premises is that such a system be capable of generating expressive animation on the fly, automatically, in response to the user's interactions with the simulated characters. It's interesting to note that her perspective as a theater student is markedly different from that of the authors of traditional animation software, who by and large come from a computer science background.

One of my objectives, therefore, was to do what I could to help bring these systems a little closer to realization. The way to do this, I think, is to augment an animation system with some of the sorts of knowledge and reasoning abilities that work in AI has espoused. This work is in fact being done at the University of Pennsylvania [Badler].

There is a good deal of work in the AI community on story understanding [Schank] and story generation [Meehan], but with one exception [Takashima], these have not as yet been connected up with any graphics. Thus the stories remain as paragraphs of text and symbolic networks, rather than visually engaging performances. I expect that work which merges story-

understanding and story-animation software will be enormously useful to those who are attempting to research either of these fields separately.

The return of the semi-autonomous agent

Perhaps the hottest trend in user interface research lately is the proliferation of devices for direct manipulation. Datagloves from VPL and Mattel, position sensors from Polhemus and Bird, head-mounted displays, force-feedback joysticks, and so forth — all of these promise to open up a much more intimate, direct style for manipulation and perception. Accompanying all these i/o devices is the demand for bigger and faster computers, to process the signals which are entering and leaving the computer with increasingly staggering bandwidths.

The objects in these "virtual reality" worlds are purposefully deterministic. Doors can be opened or closed, balls can be tossed and fall due to simulated gravity, and clouds of elastic springs are connected into networks forming wiggling gelatin-like meshes. These are not autonomous agents, but playthings for users suited up with the appropriate gear to fling and sculpt and inspect. "Behavior" for these objects is therefore the mechanistic simulation of physical law.

On the other hand, there's a thriving interest in fully autonomous agents. The field sometimes goes by the name"artificial life[1]." By investigating genetic algorithms, cellular automata, neural networks, self-organizing systems, etc., researchers are hoping to find emergent properties from systems with minimal initial programming. The results so far have produced fascinating simulations of complex behavior like that exhibited by ant swarms and bacterial colonies.

Between the realistic furniture of Virtual Reality and the ant swarms of Artificial Life, there lies a middle ground of autonomy. This is typified by the semi-autonomous (but fictional) robots of the 50's and 60's; a character like Robbie the Robot who was pictured as a benign servant who just does what you tell it. A semi-autonomous agent has enough independence to do what you command without

1. *Artificial Life*, Conference Proceedings, edited by Chris Langton, Addison-Wesley, 1989

specifying the tedious details, but not so much independence as to act without your motivation.

This sort of agent has become unfashionable lately, which is perhaps one reason why I chose to study it. Unlike Artificial Life, you wouldn't want to be surprised by its behavior, and unlike the objects in Virtual Reality, you wouldn't want to directly manipulate it with a dataglove.

Future work

Divadlo's movies are silent, but it is clear that sound is a critical element to drama. I would have liked to equip the real-time playback mechanism with a MIDI-controlled sampling synthesizer to augment the animation with synchronized cues such as door slams, foot steps, gasps of surprise, and perhaps even real dialogue.

Real directors of course do not direct real actors with simply verbal instructions, but they often augement these instructions with gestures and diagrams to indicate how they want actors to behave. I felt a natural language interface was a bold move on my part, to get away from the direct manipulation mentality espoused by conventional animation tools. But an interesting research topic would be to examine how to integrate gestures and diagrams to give the user to more flexibility when indicating spatial information to the system.[1]

It's enormously frustrating to try building something which touches on so many diverse problems, especially those of modelling the complexities of human behavior. Simply listing all the ways this work could be improved would probably take reams of paper.

1. The SDMS (Spatial Database Management System, 1980) research project at the MIT Architecture Machine Group was informally known as "Put that there," due to its then-uniqe integration of verbal and gestural input. I would dub the gestural enhancement of a system like my own as *"Perform* that there."

Suffice it to say that a system that purports to be a general-purpose tool for generating animation should theoretically accommodate all the complexities of human interaction and drama in its design. I could not hope to do this, but still I would not be satisfied with a system that simply omitted a few components like the knowledge base, or the planner, or the renderer. I see this thesis as a vertical slice through the entire animation process, showing how all those components might fit together. This slice is perhaps a tenuous one, and at every point could be fleshed out with improvements in all directions.

Although I provided a mechanism for building up a library of actions, the library itself is not that well-stocked. For example, I deeply regret that due to time limitations I was unable to implement a decent walking motor skill. I expect this paucity could be resolved with the cooperation of a magnanimous animation studio, or also by incorporating others' work on dynamics-based task models, such as [Isaacs & Cohen], [Zeltzer], [Badler], [Girard], [Wilhelms], etc. But clearly this would be a bit of work as well.

Part of the reason why such work is still so hard is that these systems are standalone efforts. No attempt has been made to implement them in any way that would promote modular compatibity with any other aspects of an automatic animation environment, nor has there been any good reason for or understanding how to do so. Perhaps Badler's group at Univ. of Penn. is the closest to formulating an open, modular environment for task-level animation. If this kind of system were to be coupled with one that understood drama, in the sense described by [Laurel], we could finally make our own custom expressive animations on our desktops.

Bibliography

Allen, James, *An Interval-Based Representation of Temporal Knowledge,* IJCAI 1981.

Badler, Norman, et al., *Animation from Instructions,* chap. 3 of *Making Them Move,* ed. by Badler, Barsky, & Zeltzer; Morgan Kaufmann, San Mateo, CA, 1991.

Chen, David, *Rendermatic,* rendering software, MIT Media Lab, 1990.

Finin, Timothy, *BUP - A Bottom-Up Parser for Augmented Phrase Structured Grammars,* Franz lisp software, Univ. of Pennsylvania, 1985

Fishwick, Paul, *Hierarchical Reasoning: Simulating Complex Processes over Multiple Levels of Abstraction,* Technical Report TR-86-6, University of Florida, 1986

Forbus, Kenneth, A *Study of Qualitative and Geometric Knowledge in Reasoning about Motion,* TR-615, MIT AI Lab, February 1981.

Forbus, Kenneth, *Qualitative Process Theory,* PhD Thesis, MIT AI Lab, July, 1984.

Gangel, Jeffrey Scott, *A Motion Verb Interface to a Task Animation System,* MS thesis, Univ. of Pennsylvania, MS-CIS-85-35, 1985.

Goldberg, Adele and Robson, D., *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, 1983.

Haase, Ken, *ARLOtje Internals Manual*, MIT Media Lab, 1990.

Kahn, Kenneth Michael, *Creation of Computer Animation from Story Descriptions*, PhD thesis, AI TR-540, MIT AI Lab, 1979.

Laurel, Brenda Kay, *Toward the Design of a Computer-Based Interactive Fantasy System*, PhD thesis, Ohio State University, 1986

Lenat, Doug, and Guha, R., *Building Large Knowledge-Based Systems*, Addison-Wesley, 1990.

N. Magnenat-Thalmann and D. Thalmann, *The Direction of Synthetic Actors in the film "Rendezvous à Montreal,"* IEEE Computer Graphics and Applications, vol 12, pp.9-19, 1987.

Meehan, J., *The Metanovel: Writing Stories by Computer*, Yale University, Department of Computer ScienceResearch Report #74, September 1976.

Minsky, Marvin, *Society of Mind*, Simon and Schuster, New York, 1985.

Paul, Richard, *Robot Manipulators*, MIT Press, Cambridge, 1981.

Risdale, G., Hewitt, S., and Calvert, T.W., *The Interactive Specification of Human Animation*, Proc. Graphics Interface-86 Conference, Vancouver, 1986.

Schank, R.C. and Abelson, R.P. *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum Press, Hillsdale, NJ, 1977.

Sims, Karl, *Locomotion of Jointed Figures over Complex Terrain*, MS Thesis, MIT Media Lab, June 1987.

Takashima, Y., Shimazu, H., and Tomono, M., *Story Driven Animation*, CHI+GI 1987 Proceedings, ACM SIGCHI, pp.149-153, 1987.

V.P.L. Research Inc., *RB2 Virtual Environment System*, Product Report, VPL Research, Redwood City, CA, USA, 1989.

Winograd, Terry, *Understanding Natural Language*, Cognitive Psychology Vol. 3, (1) 1972.

Winston, Patrick, *Artificial Intelligence*, 2nd Edition, Addison-Wesley, 1984.

Zeltzer, David, *Motor Problem Solving for Three Dimensional Computer Animation*, Proceedings of L'Imaginaire Numerique, May 14-16 1987.

Appendix – Grammar rules

The grammar rules

The 94 grammar rules used by Divadlo are given below. A grammar rule in BUP has the following components:

1. A left-hand side (LHS). This is a single token indicating the syntactical meaning of the phrase, such as NOUN-PHRASE.

2. A right-hand side (RHS). These are the components which make up a phrase. A rule is applicable when the pattern of the RHS matches the current parse. A special exception is when the RHS is :OTHER, which always matches anything. :OTHER is used in conjunction with :TEST (see below) to match odd kinds of tokens that are not in the lexicon, such as real numbers.

3. The semantics of the result (optional). This defaults to simply a list of the semantics of each component of the RHS. Most of the rules however, as you can see below, use a form instead.

4. :test - When provided, the test is a lisp expression that must hold true for the rule to be applicable. This is only performed if the phrase matches the rule's RHS. Providing extra TESTs allows finer control over which expressions are accepted. One thing I did a lot was to use the FEATURE function to examine the features of one or more components of the RHS, preventing certain illegal combinations from being accepted.

5. `:features` - This specifies the features of the phrase, if this rule applies. Normally, this defaults to the concatenation of all the features of the components, but I used this to add and delete features in some cases as they are discovered.

6. `:weight` - A weight indicates the "goodness" of this particular parse, it defaults to the normalized sum of the weights of the RHS. When a given sentence can result in several parses, these parses are sorted by weight, and the one with the highest score is used. This is very handy sometimes in giving priority to one solution in ambiguous cases.

A brief explanation of quote-macro notation in lisp

Many of the forms making up these rules will look rather odd even to readers with some Lisp experience. In particular, the use of commas (`,`) atsigns (`@`), and other things (`#>2`) are not commonly used in lisp, so a bit of explanation is in order.

One can use a forward quote character to write a list, such as `'(1 2 3)`, which prevents the insides of the list from being evaluated. A special feature of lisp allows one to use a backquote instead of a forward quote, and use a comma to selectively evaluate arguments[1]. Thus,
`` `(1 2 ,(sqrt 9)) `` would give the same result as `'(1 2 3)`.

The expression "`,@`" allows one to unravel and insert a list within a backquoted list. The following examples illustrate this point briefly

```
(setq x '(1 2 3))
⇒ (1 2 3)
'(A B X D)
⇒ (A B X D)
`(A B ,X D)
⇒ (A B (1 2 3) D)
`(A B ,@X D)
```

1. For more info about backquote and other aspects of lisp syntax, see *Common Lisp: the Language*, Guy Steele, Digital Press, 1990

```
 ⇒ (A B 1 2 3 D)
```

Another thing that needs explaining is that "#>" is a macro to simplify reference to the RHS of the rule. "#>2" means "the second element of the RHS of this rule. This notation is not standard to lisp, it is unique to BUP.

```
R. S ← (IGNORE S)
 :semantics #>2
 :example noise words like "uh"
R. S ← (S IGNORE)
 :semantics #>1
R. S ← COMMAND
 :example [Get the ball]
R. S ← QUERY
 :example [Where is the ball?]
R. S ← STATEMENT
 :example [The ball is in the bagel-stretcher.]
R. S ← QUIT
 :semantics '(quit ,#>1)
 :example [Quit]
R. COMMAND ← VP
 :semantics '(command ,@#>1)
 :example [Get the ball]
:test (not (eq 'be (car #>1)))
R. COMMAND ← (FIRST-NAME VP)
 :example [Bob,][get the ball]
 :semantics'(command (agent ((person ,@(get-value #>1
                                        'name-string))))
            ,@#>2)
 :test (not (or (feature 2 'third-singular)
                (feature 2 'past)
                (feature 2 'future)))
 :weight 2
R. STATEMENT ← (NP VP)
 :example [The duck][bites albatrosses]
 :semantics '(statement ,@#>2 (agent ,@#>1)))
R. STATEMENT ← (HOW NP VP)
 :example [In this scene][the duck][eats ham]
 :semantics '(statement ,@#>3 (agent ,@#>2) ,@#>1))
R. STATEMENT ← (NP AUX-BE ADJ1)
```

```
 :example [The duck] [is] [blue]
 :semantics '(statement has-property ,#>1 ,#>3))
R. STATEMENT ← (NP AUX-BE PPHRASE)
 :example [The duck] [is] [in the oven]
 :semantics '(statement has-property ,#>1 ,#>3))
R. STATEMENT ← ("while" STATEMENT STATEMENT)
 :example [While][Rick sleeps][Greykell watches TV]
 :semantics '(parallel ,#>2 ,#>3))
R. STATEMENT ← (STATEMENT "while" STATEMENT)
 :example [David gets a sandwich][while][the ads continue]
 :semantics '(parallel ,#>1 ,#>3))
R. STATEMENT ← ("as" STATEMENT STATEMENT)
 :example [As][Bob eats,][Lisa laughs]
 :semantics '(parallel ,#>2 ,#>3))
R. STATEMENT ← (STATEMENT "as" STATEMENT)
 :example [David screams][as][Diane tries to paint]
 :semantics '(parallel ,#>1 ,#>3))
R. STATEMENT ← (STATEMENT "and" STATEMENT)
 :example [Bob eats][and][Lisa laughs]
 :semantics '(parallel ,#>1 ,#>3))
R. STATEMENT ← (STATEMENT "but" STATEMENT)
 :example [Bob eats][but][Lisa laughs]
 :semantics '(parallel ,#>1 ,#>3)
R. STATEMENT ← ("after" STATEMENT STATEMENT)
 :example [After][Bob eats][Lisa laughs]
 :semantics '(sequence ,#>2 ,#>3)
R. STATEMENT ← (STATEMENT "then" STATEMENT)
 :example [Bob eats][then][Lisa laughs]
 :semantics '(sequence ,#>1 ,#>3)
R. STATEMENT ← ("then" STATEMENT)
 :example [then][he killed her]
 :semantics '(sequence ,#>2)
R. STATEMENT ← ("after" "that" STATEMENT)
 :example [after][that][he killed her]
 :semantics '(sequence ,#>3)
R. QUERY ← ("where" AUX-BE NP)
 :example [Where] [are] [the large cats]
 :semantics '(query location? ,#>3)
R. QUERY ← ("what" VERB NP VERB)
 :example [What] [did] [you] [do]
 :semantics '(query doing-what? (agent ,@#>3))
 :example [What] [are] [you] [doing]
```

```
 :test (and (member #>2 '(do be))
            (eq #>4 'do))
R. QUERY ← ("who" VP)
 :example [Who] [is eating the fish]
 :semantics '(query who-is-doing? ,@#>2))
R. QUERY ← ("who" AUX-BE PPHRASE)
 :example [Who] [is] [in this scene]
 :semantics '(query who-is ,#>3))
R. QUERY ← ("what" AUX-BE NP VP)
 :example [What] [are] [you] [holding]
 :semantics '(query object? ,@#>4 (agent ,@#>3))
 :test (and (feature 4 'transitive)
            (feature 4 'participle)))
R. QUERY ← (AUX-BE NP VP)
 :example [Are] [you] [sitting]
 :semantics '(query doing-yes/no? ,@#>3 (agent ,@#>2))
 :test (and (feature 3 'participle)))
R. QUERY ← (AUX-BE NP ADJ)
 :example [Is] [the door] [closed]
 :semantics '(query apt-description? ,#>2 ,#>3))
R. QUERY ← (AUX-BE NP PPHRASE)
 :example [Is] [the door] [in the kitchen]
 :semantics '(query apt-description? ,#>2 ,#>3))
R. QUERY ← (AUX-BE NP NP)
 :example [Is] [the blue door] [the kitchen door]
 :semantics  '(query same-object? ,#>2 ,#>3))

;; Verb Phrases
R. VP ← VP1
 :example [singing]
R. VP ← (VP1 HOW)
 :semantics (append #>1 #>2)
 :example [singing] [in the tub]
R. VP ← (HOW VP1)
 :semantics  (append #>2 #>1)
 :example [loudly] [singing]
R. VP ← (HOW VP1 HOW)
 :semantics (append #>2 #>1 #>3))
 :example [loudly] [singing] [in the tub]

 ;; Auxiliary BE verb
```

```
R. AUX-BE ← VERB
 :semantics  #>1
 :example [am]
 :test (and (eq #>1 'be)
            (not (feature 1 'participle)))
R. PARTICIPLE ← VERB
 :semantics  #>1
 :test (feature 1 'participle))
R. V ← (AUX-BE PARTICIPLE)
 :example [is][running]
 :semantics '(,#>2 (tense :present :participle))
 :test (feature 1 'present)
 :features (remove-duplicates (remove 'participle
                                      (feature))))
R. V ← (AUX-BE AMOUNT PARTICIPLE)
 :example [is][not][crying]
 :semantics '(,#>3 (tense :present :participle)
               (amount ,#>2))
:test (feature 1 'present) :features
       (remove-duplicates (remove 'participle (feature))))
R. V ← (AUX-BE PARTICIPLE)
 :example [was][running]
 :semantics  '(,#>2 (tense :past :participle))
 :test (feature 1 'past)
 :features (remove-duplicates
             (remove 'participle (feature))))
R. V ← ("will" "be" PARTICIPLE)
 :example [will be][running]
 :semantics '(,#>3 (tense :future :participle))
 :features '(future ,@(remove 'participle (feature))))
R. V ← VERB
 :semantics '(,#>1 (tense :present))
 :example [sleeps]
 :test (feature 1 'present))
R. V ← VERB
 :semantics '(,#>1 (tense :past))
 :example [slept]
 :test (feature 1 'past))
R. VP1 ← (V NP)
 :example [get] [the ball]
 :semantics '(,@#>1 (object ,@#>2))
 :test (feature 1 'transitive))
```

```
R. VP1 ← (V NP NP)
 :example [give] [me] [the ball]
 :semantics '(,@#>1 (indirect-object ,@#>2)
                    (object ,@#>3))
 :test (feature 1 'indir-obj))
R. VP1 ← V
 :semantics #>1
 :example [sit]
 :test (feature 1 'intransitive))
R. VP1 ← (V AUXPREP NP)
 :example [pick] [up] [the ball]
 :semantics '(,@#>1 (object ,@#>3) (auxprep ,#>2))
 :test (and (feature 1 'transitive) (feature 1 #>2))
 :weight 2
R. VP1 ← (V NP AUXPREP)
 :example [pick] [the ball] [up]
 :semantics  '(,@#>1 (object ,@#>2) (auxprep ,#>3))
 :test (and (feature 1 'transitive) (feature 1 #>3))
 :weight 2
R. VP1 ← (V AUXPREP)
 :semantics  '(,@#>1 (auxprep ,#>2))
 :example [sit][down]
 :test (and (feature 1 'intransitive) (feature 1 #>2))
 :weight 2
R. VP1 ← (V ADJ)
 :semantics  '(become ,@(cdr #>1) (becomes ,@#>2))
 :example [becomes] [angry]
 :test (feature 1 'become))

 ;; Noun Phrases
R. NP ← FIRST-NAME
 :example [Ralph]
 :semantics  '((person ,@(get-value #>1 'name-string)))
R. NP ← PRONOUN
 :semantics  '((pronoun ,#>1)))
 :example [you]
R. NP ← STRING
 :semantics  '((string-constant ,#>1)))
 :example ["quoted string"]
R. NP ← SYMBOL
 :semantics  '((symbol ,#>1)))
```

```
 :example ['SYMBOL]
R. NP ← LOCATION
 :example [upstage]
R. NP ← (DEF-ARTICLE LOCATION)
 :semantics  #>2
 :example [the][left of the chair]
R. NP ← NP1
R. NP ← (DEF-ARTICLE NP1)
 :example [the] [big cat]
 :semantics  '(,@#>2 (definite ,#>1))
R. NP ← (INDEF-ARTICLE NP1)
 :example [a] [big cat]
 :semantics '(,@#>2 (indefinite ,#>1)))
R. NP ← (NP "*possessive*" NP1)
 :example [Steve's] [chocolate]
 :semantics '(,@#>3 (belongs-to ,@#>1))
R. NP ← (POSSESS-PRONOUN NP1)
 :example [His] [chocolate]
 :semantics '(,@#>2 (belongs-to (pronoun ,#>1)))
R. NP1 ← NOUN
 :semantics  '((member-of ,#>1)))
 :example [cat]
R. NP1 ← (NOUN NOUN)
 :example [beer] [can]
 :semantics '((member-of ,#>2) (containing ,#>1))
 :test (find 'containers (get-value #>2 'supersets)))
R. NP1 ← (NOUN "of" NOUN)
 :example [can] [of] [beer]
 :semantics '((member-of ,#>1) (containing ,#>3))
 :test (find 'containers (get-value #>1 'supersets)))
R. NP1 ← (NOUN NOUN)
 :example [kitchen] [door]
 :semantics '((member-of ,#>2) (belongs-to ,#>1))
 :test (find 'rooms (get-value #>1 'supersets))
 :weight 2
R. NP1 ← (ADJ NP1)
 :semantics  (append #>2 #>1))
 :example [big brown] [cats]
R. NP1 ← (QUANTITY UNIT)
 :semantics  '(units ,#>1 ,#>2))
 :example [500] [volts]
R. NP ← (NP CONJ NP)
```

```
 :example [the cat] [and] [the sofa]
 :semantics  (list #>2 #>1 #>3)
 :features '(plural ,@(feature)))
; add plural to existing features
R. NP ← (NP NP CONJ NP)
 :example [the mouse][the cat][and][the sofa]
 :semantics  (list #>3 #>1 #>2 #>4)
 :features '(plural ,@(feature))

;; Adjectives
R. ADJ ← ADJ1
 :semantics (list #>1)
 :example [brown]
R. ADJ ← (ADJ ADJ1)
 :semantics (cons #>2 #>1)
 :example [crazy big] [brown]
R. ADJ1 ← ADJECTIVE
 :example [brown]
 :semantics '(,(get-value #>1 'adjective-form-of) ,#>1))
R. ADJ1 ← (ADJECTIVE PPHRASE)
 :example [angry][at bob]
 :semantics '(mood ,#>1 ,#>2)
 :test (eq 'mood (get-value #>1 'adjective-form-of)))
R. ADJ1 ← QUANTITY
 :example [a couple of]
R. ADJ1 ← (AMOUNT ADJ1)
 :example [very][happy]
 :semantics  '(,@#>2 ,#>1)
R. AMOUNT ← ADVERB
 :semantics  #>1
 :example [very]
 :test (eq 'amount (get-value #>1 'adverb-form-of))
R. QUANTITY ← NUMBER
 :semantics  '(quantity ,#>1)
 :example [three]
 :features (if (< 1 #>1) '(plural)))
R. QUANTITY :other
 :example ; a printed number like [149.5]
 :semantics '(quantity ,(read-from-string #>1))
 :test (numberp (read-from-string #>1))
 :features (if (< 1 (read-from-string #>1)) '(plural)))
```

```
;; Prepositional phrases
R. PPHRASE ← (PREP NP)
  :semantics '(,#>1 ,@#>2))
  :example [into] [the oven]
R. PPHRASE ← PRONOUN
 :example [here]
 ;;some pronouns imply a direction; an implicit "to"
 :semantics  '(to (pronoun ,#>1))
 :test (feature 1 'place)
 :weight 2
R. PPHRASE ← LOCATION
 :example [upstage]
 ;;; (implicit "to")
 :semantics '(to ,@#>1))

;; Positions are adjectives, locations are nouns
R. ADJ1 ← POSITION
R. POS-ADJ ← ADJECTIVE
 :semantics #>1
 :test (eq 'position (get-value #>1 'adjective-form-of)))
R. POSITION ← POS-ADJ
 :example [upstage]
 :semantics '(position ,#>1)
 :weight 1.5
 :features '(position)
R. POSITION ← (POS-ADJ POS-ADJ)
 :example [upstage] [right]
 :semantics '(position ,#>1 ,#>2)
 :weight 2
 :features '(position)

 ;; Locations
R. LOCATION ← POSITION
  :semantics (list #>1)
 :example [upstage]
R. LOCATION ← ("the" POSITION "of" NP)
 :example [the] [left] [of] [the cat]
 :semantics '(,#>2 (of ,@#>4))
R. LOCATION ← (POSITION "of" NP)
 :example [left] [of] [the cat]
 :semantics '(,#>1 (of ,@#>3))
```

```
R. HOW ← (HOW1 HOW)
 :semantics  (cons #>1 #>2)
 ;; HOW is a list of adverbial clauses
 :weight (if (and (feature 1 'position)
                  (feature 2 'position))
             0 1))
 ; Try to merge positions like "upstage left"
R. HOW ← HOW1
  :semantics (list #>1)
;; HOW1 is a single HOW clause
R. HOW1 ← ADVERB
 :semantics '(,(get-value #>1 'adverb-form-of) ,#>1))
R. HOW1 ← PPHRASE

 ;; Other
R. STRING ← :other
 :example ; a quoted string like "splunge"
 :semantics (read-from-string #>1)
 :test (stringp (read-from-string #>1))
R. SYMBOL ← :other
 :example ; a quoted symbol like 'CHAIR
 :semantics (read-from-string (subseq #>1 1))
 :test (char= (elt #>1 0) #\')))
```

Acknowledgements

You find the best folk in the most dangerous places. I learned bravery and a lot more in the Jungle – thanks to Pie, rob, Chip, Jamy, NewT, Eddie, Colin, Steven, Teller, Isawake, Dr. Stockdale, and all the others.

The guys in the Snake Pit also have my undying thanks for sharing their knowledge and their toys with me – David Chen, Steve Drucker, Tinsley Galyean, Mikey M^{c}Kenna, Stevie Pieper, Dave Sturman, and Fran Taylor. Extra-super thanks to Michael Johnson and Marg Minsky – without their extraordinary friendship, help, and willingness to calm me down and feed me at the right times, I would've been carted off long ago.

Thanks to Marvin Minsky, Nicholas Negroponte, Jerome Wiesner, and everyone at the Media Lab for believing in me and giving me the exquisite opportunity to pursue the art of science, and the science of art. It's safe to say there's practically nobody at the Lab who *didn't* help me at some point, so I'll just have to come around and thank you all individually.

Thanks to the folk who taught me the meaning of interactive entertainment: David Shaw, Diane Martin, Greykell & Rick Dutton, Bob & Lisa Montgomery, John O'Neil, Mike Massimilla, Walt Freitag, Barbara Lanza, Harley & Yuriko Rosnow, and a cast of hundreds.

A special thanks to Gilberte, an infinite source of warmth and understanding.

And lastly, to my mom and dad, and my wonderful siblings, what else can I say, but thanks for making it all possible.