View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)
Department of Computer & Information Science

April 1991

Action Composition for the Animation of Natural Language Instructions

Libby Levison
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Libby Levison, "Action Composition for the Animation of Natural Language Instructions", . April 1991.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-28.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/383
For more information, please contact repository@pobox.upenn.edu.

Action Composition for the Animation of Natural Language Instructions

Abstract

This research project investigates the relationship between computer animation and language; specifically, developing utilities to generate animation from natural language instructions. Methods for specifying simulations at a task-level rather than at the level of individual motions are discussed. We envision a system which would allow engineers or technical staff who currently write instruction manuals to instead generate animations which illustrate the task. However it is unlikely that these engineers would have sufficient knowledge of animation techniques. For this reason, such a system must provide high-level tools to permit the engineer to animate a task without becoming entangled in low-level animation issues.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-28.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/383

Action Composition For The Animation Of Natural Language Instructions

MS-CIS-91-28
GRAPHICS LAB 40
LINC LAB 200

Libby Levison

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

April 1991

1 Introduction

This research project investigates the relationship between computer animation and language; specifically, developing utilities to generate animation from natural language instructions. Methods for specifying simulations at a task-level rather than at the level of individual motions are discussed. We envision a system which would allow engineers or technical staff who currently write instruction manuals to instead generate animations which illustrate the task. However it is unlikely that these engineers would have sufficient knowledge of animation techniques. For this reason, such a system must provide high-level tools to permit the engineer to animate a task without becoming entangled in low-level animation issues.

In addition to its potential as an instructional or human factors analysis tool, the animation environment provides a means of visually corroborating the interpretation of an instruction set. This is useful both for Computational Linguists (interested in a means to check implementations which analyze discourse) and for engineers who write instruction manuals (the animation will provide a way for instructors to disambiguate their instructions). Animation has benefits over live video in that the animation scene abstracts away from extraneous information and focuses on those details relevant to the task. Furthermore, the task might take place in a hostile or dangerous environment in which it would be impossible to create a film due to the dangers to the actors, or in a developmental environment which simply does not exist.

The research described here was carried out in conjunction with the **Anim**ation and **N**atural **L**anguage Project (AnimNL) – a joint effort between the Computer Graphics Research Lab and the Language, Information and Computation Lab at the University of Pennsylvania. We believe that we are in a unique position to investigate instruction understanding, the automatic generation of human-like motion and their relationship.

I begin (Section 2) with a brief discussion of the AnimNL project and this research task. Section 3 describes the software used in the implementation. Section 4 explains the composition and hierarchical structure of the **task-actions**, and discusses such contingencies as timing conditions. Finally Section 5 elaborates issues that became apparent while carrying out this project, and Section 6 presents some ideas for future work.

2 The AnimNL Project

The Graphics Lab's interest in human factors analysis suggested the domain of repair and maintenance tasks involving human agents. This domain, goal-oriented but animatable, provides a rich source of instructional texts. Our initial strategy is to analyze a set of instructions, produce a linguistic interpretation and analysis of the text, and use this representation to generate an animation. The goal of the project is:

> *Given a task specified in Natural Language instructions, automatically generate a (narrated) simulation of an agent executing the task.*

One interest of the AnimNL project is the relation between natural-language instructions and animated simulations specified at a task-level.

In discussing these animations certain assumptions are made. One assumption is that the engineer who currently writes the instruction manuals will "write" the animations – they will not be created by a professional animator. However current animation systems do not provide for

someone who is not an animator. A method which permits an engineer to specify an animation is needed. One goal of this research project was to test the feasibility of generating animations from a task-level, as opposed to a lower level; i.e. allow the engineer to specify high-level goals rather than each simulation movement. This is the third level of Zeltzer's animation taxonomy, what he referred to as his "task-level" [12]. However our motivation differs from his in that our intention is to design a system not for animators, but for engineers with little knowledge of task-level motor-control and related issues.

To illustrate some of these issues, consider the instruction:

Move the cup to the table

given in a scene with an animated agent, a table, and a cup on a shelf next to the table. The animator-engineer should not need to specify the number of degrees to swing the shoulder joint or straighten the elbow joint for each frame of the animation to move the agent's hand to the cup. Rather, the animator should be able to specify the action at a much higher level – by utilizing a set of high-level **task-actions**. If, instead of specifying joint movements, the engineer could specify:

reach-action (*hand cup*)
move-action (*hand table-top*)

he would be describing the action at a task-level. Using combinations of task-actions, an engineer could "program" an animation. The AnimNL Project is interested in using these same task-actions in generating animations from instructions.

2.1 This Research

While the AnimNL group plans on beginning with natural-language instructions and ending with animation, the goal of this research project was to test the robustness of the interplay between software packages available at the University of Pennsylvania which will be used in building the animations. In addition, we wanted to test the feasibility of specifying high-level task definitions in this environment. As non-animators, we were interested in developing a set of high-level animation definitions that would enable the animation of a set of instructions. The instructions chosen for the exercise describe the removal of a Fuel Control Valve (FCV) from an aircraft fuselage. The results of this research illustrate issues the AnimNL project will encounter in future attempts to generate animation from natural language instructions. They also demonstrate that it is indeed possible to define a set of high-level animation definitions which provides a means of describing an animation at a task-level.

3 Programming Environment

The research described herein uses software developed in the Computer Graphics Research Lab at the University of Pennsylvania. The environment is the *Jack*™ modeling system which runs on Silicon Graphics Iris workstations and provides three-dimensional modeling capabilities as well as extensive human factors and anthropometric analysis tools. Built on a powerful representation for articulated figures composed of joints and segments with boundary geometry,

† *Jack* is a trademark of the University of Pennsylvania.

Jack provides an interactive interface into a 3D articulated world. *Jack* provides low-level animation support through real-time inverse-kinematics and constraint satisfaction, in addition to collision detection and strength analysis, among other features.

The scene for the animation was created in *Jack*, the work area, tools and parts modeled as specified in the instructions. Just as the engineer who currently writes the instruction manuals has knowledge of the task and knows, for example, that a Phillips head screw driver is required, it is assumed that the engineer-animator will have the knowledge required to lay out the scene of the animation. It is also assumed that a skilled engineer is already trained in analyzing tasks and developing instruction sets for the domain. This project simply provides a different medium in which the engineer can explain the task.

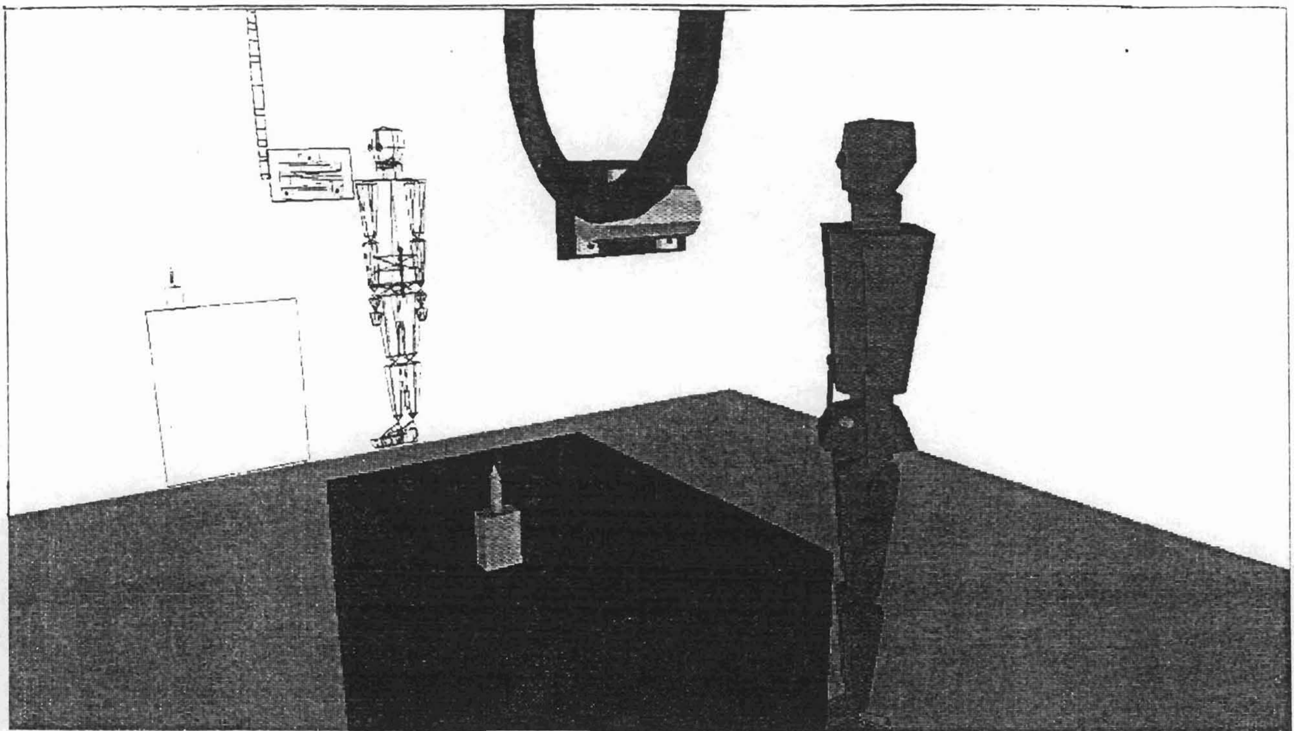

Figure 1: *Beginning scene from the animation, Fuel Valve installed. The animated agent is in the center. His projection to the yz-plane at the far left gives a 3D effect to the picture. An oil bottle is on the worktable. The circle in the upper center of the picture is an abstract representation of the fuselage; the FCV is the cylindrical object mounted to the attached plate.*

The animation was programmed in **Yaps**, a symbolic task simulator [4]. **Yaps** provides *animation-directives* which access *Jack*'s low-level animation commands. These animation-directives are not only ordered and sequenced via **Yaps**'s temporal and conditional relationships, but can also be composed to produce parameterized simulation procedures. These procedures, called **task-actions**, are defined for a number of parameters (agent, object, location, etc). The same task-action can thus be used any number of times with different parameters to

create different animation segments. The possibility of defining and reusing these procedures greatly simplifies the animation programming problem for the engineer. By extending these procedural compositions far enough, high-level procedures could be generated such that the mapping from the instructions to these procedures would be straightforward.

KB[3] is a frame-based, object-oriented knowledge system which establishes symbolic references to *Jack*'s geometric data. While *Jack* maintains and manipulates the geometric model of the world, **KB** maintains the symbolic information. **Yaps** uses **KB**'s symbolic representation to manipulate the geometric model. (These symbolic **KB** representations are passed to the **Yaps task-actions** as parameters.) This frees **Yaps** from "knowing" the specific world coordinates of an object or the object's geometric representation in *Jack*. For instance, if *Jack* contains a model of a cup, **KB** would have an entry which identified *cup* as that particular *Jack* entity. **Yaps** has no knowledge of the object's location; **KB**'s mapping from symbolic to geometric representation will resolve any ambiguity. Thus the animator need not talk about *the-cup-on-the-table-at-world-coordinates-(x,y,z)*, but can reference the symbolic entity, *cup*. Should the *cup* move during the course of the action, **KB** resolves the problem of the *cup*'s exact location.

4 Task-actions

At the time of this research, **Yaps** provided only three low-level animation-directives with which to access *Jack*'s animation commands. These are *generate-motion*, *create-constraint* and *delete-constraint*. *Generate-motion* causes an object (not necessarily animate) to move from its current location to another. (*Jack* has no notion of how motion is generated. It uses inverse kinematics and joint constraint information to analyze how a motion is propagated through a figure.) *Create-constraint* establishes a physical link between two (not necessarily adjacent) objects. If two objects are linked together and one of the objects is moved, the second object moves along with it. The physical constraint (relation) between the objects is maintained. *Create-constraint* can be further specified to use *positional* and/or *orientational* alignments. *Delete-constraint* removes the specified constraint between two objects.

Yaps provides a mechanism for building animation templates by combining or composing the above animation-directives. Using different combinations of *generate-motion*, *create-constraint*, and *delete-constraint*, and varying the agents and the objects of these *animation directives* as well as their temporal and causal relations, it is possible to build a set of task-actions. Task-actions can themselves be composed into more complex task-actions. As the procedures acquire more specification, the task-actions approach task-level descriptions. It is important to note, however, that task-actions simply define templates; an animation is realized by instantiating the task-actions, supplying parameters as well as timing constraints and other conditions. The composability of the task-actions allows for the definition of some abstract and high-level concepts. It is these high-level animation descriptions which will allow the engineer to program an animation at a task-level.

4.1 Defining task-actions

4.1.1 Motivating Some task-actions

The first templates to be defined were simply encapsulations of the entry points into the *Jack* animation-directives. These task-actions are **reach-action**(*agent object*), **hold-action**(*agent object*) and **free-object-action** (*object*); they correspond to *generate-motion*, *create-constraint* and *delete-constraint* respectively.[1] In the following, the use of *agent* and *object* is simply for readability; for example, a **hold-action** can be applied between two objects (*e.g.*, **hold-action** (*wrench-head 5-8th-socket*)).

Consider trying to describe the actions inherent in the example:

Move the cup to the table

assuming that the agent is not currently holding the cup. The agent must first hold the cup before he can move it. How is this animation specified? Explicitly stating the sequence of actions:

reach-action (*agent cup*)
hold-action (*agent cup*)

to cause the agent to reach his hand to the location of the cup and to constrain the cup to his hand seems awkward. The composability of the task-actions allows a new task-action to be defined: **grasp-action**:

(deftemplate **grasp-action** (*agent object*)
    reach-action (*agent object*)
    hold-action (*agent object*)).

Grasp-action is a sequence of instantiations of two primitive task-actions.

Now that the agent can grasp the cup, how can he move the cup? A second action, **position-action**, is defined to relocate the cup to a new location:

(deftemplate **position-action** (*object1 location*)
    reach-action (*object1 location*)
    hold-action (*object1 location*)).

If a previous action had left an object (the cup) in the agent's hand, this task-action could be used to move the object to a new location (**position-action** *cup table*). (In this instruction set, the only time the instruction "move something that is already being held" was used required that the object be constrained to the new location. This is the justification of the **hold-action** in this definition.) Note here that *location* could be the location of *object2*.

Thus, to animate the instruction:

Move the cup to the table

the *animation-script* could be:

[1] Although the names chosen for the task-actions do make some attempt to elicit their definition, there was no attempt to come up with definitive definitions of these actions in this segment of the research project.

grasp-action (*agent-right-hand cup*)
position-action (*cup table-top*).

In animating *move*, it is still necessary to specify a list of commands. No high-level task-action has been defined for *move* and therefore the action must be described in increments. However, it is possible to encapsulate the elements of the primitive list at a still higher task-level. **Move-action** could be defined as:

(deftemplate **move-action** (*agent object1 location*)
grasp-action (*agent object1*)
position-action (*object1 location*)).

The actual definition is:

(deftemplate **move-action** (*agent object1 location*)
grasp-action (*agent object1*)
reach-action (*object1 location*));

in other words, reach to *object1*, create a constraint, and move *object1* to *location* (where *location* might be the location of some *object2*). In the *Move the cup* example, the instantiation required to achieve the desired animation would be:

move-action (*agent-right-hand cup table-top*).

This conciseness is one benefit of task-action composition.

Once the *cup* is actually on the *table*, it can be "un-grasped" by using:

free-object-action (*cup*)

which breaks the constraint between the *hand* and the *cup*. If the *hand* is later moved, the *cup* will no longer move with it.

4.1.2 Domain-specific task-actions

The *Move the cup to the table* example motivated a few fundamental task-action definitions. Some of these are actions common to many instructional tasks and milieus; this set of **task-actions** is also usable in the instruction set describing the FCV removal. However, it was also necessary to return to the instruction set and develop **Yaps** definitions for actions specific to the domain in question. These task-actions can be either primitive (see **turn-action** below) or compositional (see **ratchet-action**). The first new task-action, **attach-action**, is defined as:

(deftemplate **attach-action** (*agent object1 object2*)
move-action (*agent object1 object2*)
hold-action (*object1 object2*)).

This allows the agent to grasp *object1*, move it to the location of *object2*, and establish a constraint between *object1* and *object2*. The expansion of this task-action is the command string:

reach-action, hold-action, reach-action, hold-action.

Attach-action could have been equivalently defined as:

(deftemplate **attach-action** (*agent object1 object2*)
  grasp-action (*agent object1*)
  position-action (*object1 object2*))

which would expand to exactly the same *Jack* animation-directive command string as above. The task-action definitions are associative; this provides flexibility and power to the system, and increases the feasibility of defining a minimal set of task-actions to be used throughout the domain.

The FCV removal instructions also require: **turn-action** (*object degrees*). **Turn-action** causes the *object* to rotate by the specified number of *degrees*. The geometric definition of the object includes information as to its degrees of freedom; for example, around which axis a bolt will be allowed to rotate. At the time that this research was done, the system did not have a feedback tool to monitor *Jack* entities; instead of testing for an ending condition on an action (a bolt being free of its hole), actions had to be specified iteratively (the number of times to turn a bolt). **Turn-action** is actually a support routine, used in the final task-action needed to animate the FCV instructions: **ratchet-action**. This is defined as:

(deftemplate **ratchet-action** (*object degrees iterations*)
  turn-action (*object degrees*)
  turn-action (*object –degrees*)
  ratchet-action (*object degrees iterations-1*)).

Ratchet-action is used to animate of a socketwrench ratcheting back and forth.[2]

The complete set of task-actions is listed below. With this set of only nine task-actions, it was possible to program the entire animation script from the natural language instructions (see Appendix B for an excerpt of the final animation script).

- **reach-action** (*agent object*)
- **hold-action** (*agent object*)
- **free-object-action** (*object*)
- **grasp-action** (*agent object*)
- **move-action** (*agent object location*)
- **attach-action** (*agent object1 object2*)
- **position-action** (*object1 object2*)
- **turn-action** (*object degrees*)
- **ratchet-action** (*object degrees iterations*)

[2]Having to explicitly state a number of degrees is not an elegant programming solution; it would have been preferable to take advantage of *Jack*'s collision detection algorithms to determine the range of the ratchet movement. However processing considerations at the time the work was done required this rather rough implementation.

4.2 Instruction Translation

The set of task-actions now provides a language for describing the instructions. If this set is robust enough in the domain, mapping from the instructions to the animation script will be straightforward for the engineer/animator. There is still quite a bit of room for interpretation, and the nature of the task-actions dictates that there will be more than one way to describe an animation. Returning to the *Move the cup to the table* example, the engineer can now equally well animate:

Move the cup to the table.

as:

grasp-action (*agent-right-hand cup*)
position-action (*cup table-top*)
free-object-action (*cup*)

or:

move-action (*agent-right-hand cup table-top*)
free-object-action (*cup*).

In both cases, the engineer has decided to release the constraint between the *agent* and the *cup* as soon as the *cup* is on the *table-top*. But the engineer has also described the required animation at the task-level.

4.3 Sequencing Sub-tasks

Yaps is a simultaneous language; that is, all task-action instantiations are resolved concurrently. To sequence the actions and force them to occur in a specific order, the engineer/animator must use the *timing-constraints* option provided by **Yaps**. This construct allows the user to specify starting, ending and duration conditions for the instantiation of each action. It is possible to achieve the ordering needed to create a sequential animation by predicating the starting condition of instruction-2 on the ending condition of instruction-1. However a task-action template, which is defined as a series of other task-actions, has the sequencing automatically built in via the instantiation process. If this were not the case, defining **grasp-action**, for example, would be impossible because achieving and completing the **reach-action** before starting the **hold-action** could not be guaranteed.

The actions do not need to be performed discretely. Other **Yaps** timing constructs allow the actions to be overlapped and delayed by specifying *(start (after 5 min))* or *(start now)*, for example. Nor is defining a discrete linear order on the sub-tasks the only possibility. The simultaneous nature of **Yaps** is used to animate actions (such as moving an object with both hands) by simultaneously animating:

move-action (*agent-left-hand box*)
move-action (*agent-right-hand box*).

The **Yaps** timing constraints provide a powerful mechanism for specifying the relationships among the task-actions in the animation. Timing is one of the most critical issues involved in generating realistic animations; the power that **Yaps** provides in resolving timing issues greatly enhances the potential of the *Jack* animation system.

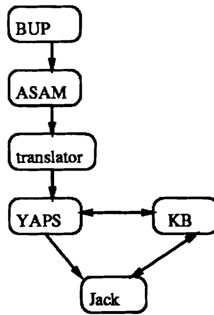

Figure 2: *Converting natural language instructions to animation.*

4.4 Task Duration

A formula developed by Fitts[6] was used to approximate the duration of a task-action. This equation estimates the time required for a human to "hit" a target (e.g., for an agent to press a button or push a panel). The calculation is based on the dimensions of the target, the performance level of the agent and the agent's distance from the target. Fitts' Law was used to calculate the duration of all **reach-action** instantiations. Thus time requirements were cumulative (i.e., the sum of the sub- task-action times). **Create-constraint** uses a small default constant time to estimate sub-task length.

4.5 Comparison with Earlier Work

Previous work by Kalita [8] took the output of a parser (BUP)[5], built representations following a verbal analysis (ASAM), translated these representations into **Yaps**, and generated a short animation. The current research did not attempt to start with a language parser; rather, the focus was to test the robustness of **Yaps**, **KB** and their interaction with *Jack* (see Figure 2).

Kalita [8] took a different approach to the problem of action description. He did not build a set of task-actions to be used in programming his animation. Instead, Kalita defined each sub-task as a sequence of animation-directives. His final animation script was completely "flat", with no hierarchical definitions. Not only does this imply that he did not benefit from using **Yaps** as a programming language (and defining reusable animation procedures), but that his animation required the animator to repeatedly decompose each verb to the level of animation-directives. He did not make use of the concept of hierarchical task-actions. This composability is the powerful tool which allows us to create animations at a task-level.

5 Discussion

This section presents a brief discussion of issues encountered in generating this animation, and how the information gained from this research has and will effect future research.

5.1 Flexibility of This System

- Timing

  The **Yaps** timing constraints provide a powerful mechanism for specifying the interrelationships among the task-actions in the animation script. Timing is one of the most critical issues involved in generating realistic animations. It is not sufficient to simply list all the actions; they must be sequenced and connected temporally. The power that **Yaps** provides in resolving timing issues greatly enhances the potential of the *Jack* animation system. An excerpt of the output from **Yaps** (a list of *Jack* commands) appears in Appendix C, which illustrates the complexities of the timing conditions required to produce the animation.

- Domain

  Although this report only discusses animated agents, the abstract task-action definitions are equally applicable for other simulated agents, be they animated or robotic. This investigation into the semantics of these instruction verbs has wide applicability. The ability to use hierarchical action descriptions to define a set of commands for a robot arm, or to describe the actions of an animated agent in a dangerous work environment, should be of interest in many different domains.

- Agent Ability

  One question which arises in defining the task-actions is whether or not the instructions need to be concerned with how the task-action varies depending on the agent and his abilities. Is it necessary to describe different task-actions for a five-foot agent, as opposed to a six-foot five-inch agent?

  Because this work is embedded in *Jack*, variations in agent ability at the animation specification level are not a concern. As long as the animation is within the agent's capabilities (and thus the animation is "solvable"), substituting different agents gives different evaluations of the tasks. By testing different agents with varying abilities, one can analyze the task requirements and gather information on human factors issues. Similarly, it is possible to vary workplace geometry, tools, and agent placement (location).

  There is a comparison here between innate and planned action. In reaching to grab a cup, we do not think about how to control the muscles in the forearm, however, we do consider the goal of getting our hand to the same location as the cup. This distinction between cognizant motion and action is internal in this animation; *Jack* manages the motor skills. The same distinction is found in the level of detail of the instructions. One does not tell someone:

  > *Reach your hand to the cup, constrain your hand to the cup, begin ...*

  Rather, we give them a goal, and allow that goal to lend information as to how it will be accomplished. The hierarchy of the task-actions captures some of this understood knowledge.

  The task-actions have been defined such that they are not concerned with the abilities of a specific agent, but allow for interpretation based on each agent's capabilities. Not only

does this allow the same animation script to be used for different agents (thereby generating different analyses), it also means that the primitives used to define the elementary task-actions are themselves defined in terms of this level of innate action.

- CPU Considerations

  The result of the research is a short film depicting an animated agent removing a part from an airplane. The generation of each frame requires a significant amount of computation; the frames were saved as they were generated and collected into a short film. The resulting animation can be played back whenever and as often as necessary.

5.2 Features Needing Improvement

- Timing

  At the time of this research, **Yaps** provided only limited timing capabilities; therefore animating conditional actions was impossible. However extensions to **Yaps** now provide access to *monitors* which allow for conditional constraints. One can create a monitor which continually checks certain conditions in the world. Upon detection of a given condition, the monitor can trigger (instantiate) a new task-action or terminate an old one. A simple example is to establish a gravity monitor and attach it to an object in the scene. If the monitor ever discovers that the object is unsupported, it would invoke **fall-action** (object).

  An additional and far more complex example would be to replace **ratchet-action** with a task-action with an inherent monitor which would turn the object (a bolt) until the object was free. This would be an improvement over ratcheting the bolt for a specified number of iterations, as was necessary in this preliminary implementation.

- **Creating-constraints**

  To avoid circularities, *Jack* must define a source and destination for each **create-constraint**. While the source object controls what is constrained to it, the destination does not. That is, if the *hand* (source) is constrained to the *cup* (destination), moving the *cup* will result in the *hand* also moving. However, moving the *hand* will not necessarily preserve the physical distance between it and the *cup*. This is clearly unintuitive and awkward, although it was possible to program the animation despite this problem. One requires a *create-constraint* (**hold-action**) which bypasses the constraint mechanism. This functionality is implemented in *Jack*, but is not currently accessible via **Yaps**.

- Posture Planning

  Work by Jung [7] promises to eliminate problems that were encountered in generating certain movements. In determining animations, there must be a certain level of *postural planning*, to prevent the agent's hand from colliding with his leg or to determine that the agent is currently too far away from the *table* to move the *cup* to it. These are unspecified in instructional dialogue. Jung's proposal is to plan the postures needed in the course of satisfying an animation-directive. This functionality, when combined with the investigation into understanding the influence the task has on the interpretation of each instruction, should greatly enhance the power of our system.

- Fitts' Law

  Fitts' Law was used to estimate the time required for many of the movements in the animation; however, the results are not exactly correct. Fitts' Law was not intended to calculate the time required for an agent to move his hand from one location to another, but rather depends crucially on the target of each motion. It also gives the *fastest* time requirement. Any use of Fitts' Law should be scaled by some motivation factor. Other algorithms need to be investigated that will give a set of durations for each task. Although Fitts' Law is not directly applicable for this domain, the algorithm does give a reasonable estimate for the durations of the task-actions. Relative to one another, the sub-task times make sense. Although the length of each **task-action** might not be correct, the animation does appear to be temporally coherent.

  Options which exist for obtaining accurate sub-task times include:

  - using existing databases for task times – these are found in the human factors domain; and
  - using a strength model to predict the minimum task completion time from the maximum rate of motion.

6 Future Directions

This is an ongoing research project. Other areas for study are briefly discussed below.

- Timing

  One useful project would be to test and illustrate the power of the **Yaps** monitors, using different starting and ending conditions and sub-task durations. A set of these conditionals would greatly enhance the task-action library. Preliminary tests in defining action templates using *monitors* include:

  start (action2) when-start (action1)
  do (action2) until-finish (action1).

  These task-actions work quite well and provide considerable programming power.

- Minimal Task-actions Set

  A subject not discussed here is the usefulness of the set of **task-actions** defined for the purposes of this animation. Investigating the applicability of these task-actions to other task domains, and discovering how their definitions would need to be changed is another area for further research. Additionally, one could try to establish which task-actions might constitute a minimal set, or if such a set even exists. Limiting this discussion to the world of animatable instructions might make it feasible.

  Since this project was completed, the set of animation-directives provided by *Jack* has been greatly expanded [10]. In fact, whether or not these commands should still be called "primitives" is questionable. *Jack* now handles much more sophisticated actions, such as *move left foot*; the implication is that the task-action descriptions will also move up the task-level hierarchy.

- The Object-oriented Approach

  One can view the work to date as using the verb in the natural language instructions as the means of selecting a particular task-action, and the verb's arguments (subject, object, etc.) as parameterizing that selection. However, a verb such as **remove** has significant variations in behavior when applied to objects as diverse as jar lids, labels, or bolts. Because of this, the appropriateness of treating the different actions simply as parameterizations is questionable. One possible course for future research is to investigate an "object-centered" approach in generating animation.

  KB is an object-oriented language. Objects could store domain specific information, for example, their degrees of freedom. A bolt might be constrained by its geometry such that, when turned, it "knows" that it must turn around a certain axis. Work has begun in defining a set of task-action which will act as object-oriented procedures. One current thought is to build a hybrid system of underspecified definitions of the task-actions in conjunction with an object-centered knowledge base. Each class of objects will have enough geometric and other supplementary inherited information that when the procedure is applied to the object, the object can provide the missing information and the task-action can be animated. There might be a core task-action definition for **open**, which will behave differently depending on whether one is animating *open the door* or *open the umbrella*. The intent is to capture the idea that action specifications describe the *goals* of the action – not the behaviors.

  As a further extension of this approach, it would be interesting to investigate what it means to instantiate, for example, **remove-action** on a hierarchical assembly. In the animation described here, **remove-action** was applied to each bolt. It would also make sense to program either **remove-action** (*the-bolts*) or **remove-action** (*FCV*). One question is whether it is correct to simply apply **remove** to each sub-part in order, or if more information is needed in the data structure of the object and what that information entails.

7 Conclusions

This work, and the work of the AnimNL group, has application in many different fields.

- Computational Linguistics

  Not only will a system which allows the generation of animation from natural language instructions provide a forum for testing algorithms which build interpretations of language, but it will also allow study of instructions as discourse. Additionally, a task-level description of an animation will allow us to capture and utilize the duality inherent in instructions that they specify both a *goal to be achieved* and the *manner to achieve that goal* [11]. This knowledge is crucial in generating animation, specifically animations of low-level motions.

- Maintenance and Human Factors Analysis

  The possibilities for using these animations from instructions as a test bed for human factors analyses of different tasks are extensive. The resulting animations can be used

to analyze whether or not an agent will be able to perform a task in an environment before the environment is actually built. They will be equally useful in designing a part for use in an environment which dictates limitations on number of agents or access to a part. As a utility which allows an engineer to "walk" an agent through a proposed design of a factory plant, or test different strategies for repairing a faulty part (to find a work strategy which offers minimal stress to the worker), this animation facility should also be of interest, and can be a crucial step in the design and test process for many industries.

- Lexical Semantics

  This project also provides an interesting means of studying taxonomies of actions and objects as well as the difference in how humans think about actions and how we specify them. Given the limited set of animation-directives in *Jack*, is it realistic to believe that one could derive a set of high-level task-actions which would allow someone to describe any set of instructions? Does such a set exist in the general case, or even in a limited environment?

As a result of this research project, a short animation of the agent removing the FCV from the fuselage of an aircraft was produced. This animation was created at a task-level by this author, who had minimal knowledge of animation. Excerpts of the final animation script can be found in Appendix B. We are continuing our investigation of a high-level language for specifying animations at a task-level.

8 Acknowledgements

This research is partially supported by Air Force HRL/LR ILIR-40-02 and F33615-88-C-0004 (SEI), Lockheed Engineering and Management Services (NASA Johnson Space Center), NASA Ames Grant NAG-2-426, NASA Goddard through University of Iowa UICR, FMC Corporation, Martin-Marietta Denver Aerospace, Deere and Company, Siemens Research, NSF CISE Grant CDA88-22719, and ARO Grant DAAL03-89-C-0031 including participation by the U.S. Army Human Engineering Laboratory and the U.S. Army Natick Laboratory.

This work is part of the AnimNL project at the University of Pennsylvania, and has benefited greatly from weekly meetings and discussions with, among others: Norm Badler, Breck Baldwin, Jeff Esakov, Barbara Di Eugenio, Moon Jung, Mike Moore, Charlie Ortiz, Mark Steedman, Bonnie Webber and Mike White. Thanks also to Dawn Griesbach, Owen Rambow and Phil Resnik.

A Fuel Control Valve Removal Instructions

1) With right hand, remove socket wrench from tool belt, move to front of body. With left hand, reach to tool belt pocket, remove 5/8'' socket, move to wrench, engage. Adjust ratchet for removal.

2) Move wrench to left hand bottom hole, apply pressure to turn in a loosening motion, repeat approximately 7 times to loosen threaded hole.

3) Move wrench away from bolt, with left hand reach to bolt and remove bolt and washer from assembly, move left hand to belt pouch place bolt and washer in pouch.

4) Move wrench to bottom right hand bolt, apply pressure to turn in a loosening motion, repeat approximately 7 times to loosen threaded hole.

5) Repeat operation 4.

6) Move wrench to top bolt, apply pressure to turn in a loosening motion, repeat approximately 6 times to loosen threaded bolt. Move left hand to grasp assembly, loosen the bolt the final turn. Move wrench to tool belt, release. With right hand reach to bolt, remove bolt and washer, place in pouch. Return right hand to assembly, with both hands move Flow Control to movable cart and release.

B Final Animation Script (Instr. 1)

```
;;;  No. 1
;;;   With right hand, remove socket wrench from tool belt,
;;;   move to front of body.  With left hand, reach to tool belt
;;;   pocket, remove 5/8" socket, move to wrench, engage.
;;;   Adjust ratchet for removal.
;
; with the right hand, grasp the wrench from the tool belt,
; and move it to site-front-body
;
(instantiate move-action
      (fred-rh wrench-handle fred-front-body-site planar)

   :instancename "r0-wrench-to-front"

   :time-constraints '((start now)
                       (duration
                          (eval(+(fitts fred-rh wrench-handle)
                                 (fitts wrench-handle
                                 fred-front-body-site)))))

; with the left hand, attach socket to wrench handle.
;  an attach entails, reaching for the socket, grasping
;  it and moving it to the wrench head.
;  if successful, free the left hand from the socket.
;
(instantiate attach-action
      (fred-lh 5-8th-socket wrench-head
                attach-socket-time planar oriented)

   :instancename "r5-attach-socket"

   :time-constraints '((start (end "r0-wrench-to-front"))
                       (duration (eval
                           (+ (fitts fred-lh 5-8th-socket)
                              (fitts fred-left-pocket
                                     fred-front-body-site)
                              attach-socket-time))))

   :on-success '(progn
                   (free-object-action fred-lh)
                   (free-object-action 5-8th-socket)
                   (hold-action wrench-head 5-8th-socket
                         :orientation-type '("orientation")))
```

C Yaps Output of *Jack* Commands

```
advance_clock_to_time(0);
create_motion("yaps00",0,"linear","end effector","site",
  "pbmale95.right_fingers.distal", "none","position", "pbmale95.right_shoulder");
create_site_colocation("world","yaps01","site", "pbmale95.right_fingers.distal");
add_motion_keyframe("yaps00","0","site","world.yaps01");
add_motion_keyframe("yaps00","50.0","site","socketwrench.socketwrench.handle");
create_motion("yaps02",0,"linear","end effector","site",
  "pbmale95.right_upper_arm.distal","none","position","pbmale95.right_shoulder");
create_site_colocation("world","yaps03","site",
  "pbmale95.right_upper_arm.distal");
add_motion_keyframe("yaps02","0","site","world.yaps03");
add_motion_keyframe("yaps02","50.0","site","pbmale95.center_torso.right_elbow_reach");
advance_clock_to_time(50);
delay_command(50);
create_motion("yaps05",50,"linear","end effector","site",
  "socketwrench.socketwrench.handle","none","position",0);
create_site_colocation("world","yaps06","site", "socketwrench.socketwrench.handle");
add_motion_keyframe("yaps05","0","site","world.yaps06");
add_motion_keyframe("yaps05","7.0","site","pbmale95.center_torso.work");
advance_clock_to_time(55);
create_named_constraint("yaps04","site","socketwrench.socketwrench.handle",
  "site","pbmale95.right_fingers.distal","none","position",
  "pbmale95.right_shoulder",1.00);
advance_clock_to_time(57);
create_motion("yaps07",57,"linear","end effector","site",
  "pbmale95.left_fingers.distal","none","position","pbmale95.left_shoulder");
create_site_colocation("world","yaps08","site","pbmale95.left_fingers.distal");
add_motion_keyframe("yaps07","0","site","world.yaps08");
add_motion_keyframe("yaps07","35.0","site","socket.socket.base");
create_motion("yaps09",57,"linear","end effector","site",
  "pbmale95.left_upper_arm.distal","none","position","pbmale95.left_shoulder");
create_site_colocation("world","yaps10","site", "pbmale95.left_upper_arm.distal");
add_motion_keyframe("yaps09","0","site","world.yaps10");
add_motion_keyframe("yaps09","35.0","site","pbmale95.center_torso.left_elbow_reach");
advance_clock_to_time(92);
advance_clock_to_time(95);
create_named_constraint("yaps13","site","socket.socket.base","site",
  "pbmale95.left_fingers.distal","none","position","pbmale95.left_shoulder", 1.00);
delay_command(95);
advance_clock_to_time(97);
create_named_constraint("yaps14","site","socketwrench.socketwrench.head",
  "site","socket.socket.top","orientation","position",0.50,0,1.00);
```

References

[1] Norman Badler, Bonnie Webber, Jugal Kalita, and Jeffrey Esakov. Animation From Instructions. In N. Badler, B. Barsky, and D. Zeltzer, editors, *Making Them Move: Mechanics, Control and Animation of Articulated Figures*, pages 51–93. Morgan-Kaufmann, 1990.

[2] Barbara Di Eugenio. Representing Action Descriptions Found in Natural Language Instructions. University of Pennsylvania, 1991.

[3] Jeffrey Esakov. KB. Technical Report MS-CIS-90-03, University of Pennsylvania, 1990.

[4] Jeffrey Esakov and Norman Badler. An Architecture for High Level Task Animation Control. In P.A.Fishwick and R.S.Modjeski, editors, *Knowledge-Based Systems: Methodolody and Applications.* Springer Verlag, 1989.

[5] T. Finin. BUP – A Bottom-up Parser for Augmented Phrase Structured Grammars. A Franz Lisp Program. Technical Report University of Pennsylvania, 1985.

[6] P. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47:381–391, 1954.

[7] Moon Jung. *Posture Planning for Human Task Animation in Workspaces.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1992, expected.

[8] Jugal Kalita. *Analysis of Action Verbs and Synthesis of Underlying Tasks.* PhD thesis, University of Pennsylvania, 1990.

[9] Jugal Kalita and Norman I. Badler. Semantic Analysis of Action Verbs Based On Physical Primitives. In *Cognitive Science Society 12th Annual Conference*, 1990.

[10] Cary B. Phillips and Norman I. Badler. Interactive Behaviors for Bipedal Articulated Figures. *Computer Graphics*, 25(4), July, 1991.

[11] Bonnie Lynn Webber and Barbara Di Eugenio. Free Adjunts in Natural Language Instructions. In *COLING90: Proc. 13th International Conference on Computational Linguistics, Helsinki*, pages 395–400, 1990.

[12] David Zeltzer. Towards an Integrated View of 3-D Computer Animation. *Visual Computer*, 1(4):249–259, 1985.