A Case Study of Flexible Object Manipulation

John E. Hopcroft
Department of Computer Science
Cornell University
Ithaca, New York 14853

Joseph K. Kearney
Department of Computer Science
University of Iowa
Iowa City, Iowa 52242

Dean B. Krafft
Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

This article describes a project undertaken to explore programming physical operations on complex flexible objects. Uncertainty about the exact state of the object makes it impossible to precisely specify the actions to be performed at the time the program is written. Furthermore, the detailed consequences of manipulations on flexible objects cannot be determined before the action is performed. Lacking precise specifications, the programmer must abstract the essential properties of objects and actions.

In an effort to study manipulation of flexible objects, a system to tie knots in rope with a robot arm was developed. The system includes an extensible graph representation for knots, a vision system that binds the contour of a physical rope to an abstract description, and a knot-tying language based on parametric motion commands. Knots of modest complexity, such as a bowline or figure 8, can be tied in a variety of ropes with minimal constraints on the initial configuration of the rope. The work highlights the importance of software engineering principles and a good programming environment for robot program development.

1. Introduction

Most work in robotics has been directed toward rigid objects with simple, regular shapes. Object geometry is usually very predictable, varying only within small tolerances from one instance of an object to another. Conventional robots and robot programming languages rely on this regularity by imposing a rigid structure on the world. The robot must know the exact form and position of objects in the environment and the precise actions to be performed.

Many objects dealt with every day are irregular, pliant, and have highly variable shapes. Flexible seals, gaskets, clips, and hoses are common components in assemblies. Materials such as rubber, cloth, paper, wire, or rope are difficult to handle and impossible to model completely. Conceptually simple manipulations are difficult to specify, and simple actions can produce complex changes to the objects. In this environment, robots and robot languages that depend on precise information and fine tolerances are difficult to program and extremely error prone.

This article reports on an experiment undertaken to study methods for programming manipulations of flexible objects. The task chosen was tying knots in rope. Knots provide a rich variety of well-defined objects whose interwoven structure is difficult to characterize. Rope handling is difficult to control, because movements can affect the shape and position of the rope in complex and unpredictable ways. As a consequence, robot programs to tie knots cannot depend on the exact structure of the rope and must rely on sensory information (Inoue and Inaba 1984).

Our goal was not to tie a single knot, but to devise a "grammar" of knots sufficient to naturally express the manipulations needed to tie many different knots. The grammar was to be independent of the device used to manipulate the rope, the size of the rope, and the detailed configuration of the rope. We also intended to develop a system to convert programs written in this grammar into robot manipulations on an actual rope.

The International Journal of Robotics Research,
Vol. 10, No. 1, February 1991,
© 1991 Massachusetts Institute of Technology.

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

The language we created is composed of parametric command specifications. The project illustrates the importance of software engineering principles and programming environments for building layers of abstraction in robot programming languages. The experience also emphasizes the interdependence of language, action, and vision for complex manipulation problems.

2. Problem Definition and Solution Requirements

The project was motivated by a desire to study programming for physical objects whose exact form is unknown at programming time. For this reason, we chose to avoid techniques that would limit the rope's freedom of motion. Special-purpose fixtures were ruled out as being too restrictive. The rope must naturally trail and twist as a portion is grasped, carried, and dropped.

Methods were not to depend on metric properties of the rope. Knot-tying programs were to work for a range of sizes, stiffnesses, and initial configurations. From the programmer's perspective the referent need not even be a knot. The language should have descriptive power to characterize the threading of smooth space curves. Road layouts and script handwriting have a woven structure that is similar to knots and could potentially be described in the language.

A single length of rope was used. The rope was placed on a table with no other objects in the work space. The rope was viewed from a camera placed above the table. A PUMA 560 robot arm was used to carry out the manipulations. Figure 1 shows the robot performing an intermediate step in a knot program.

A wide variety of knots can be tied in a single rope with one arm by weaving a piece of the rope over and under sections lying on the table. We envisioned our task as the identification of constructs for specifying the interweaving of rope and the determination of robust methods to achieve them.

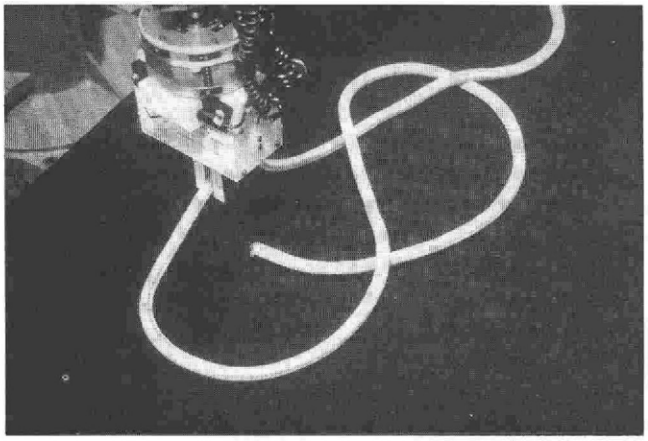


Fig. 1. The robot performing an intermediate step of a knot program. A segment is being lifted over the free end.

3. Object-Level Programming

The approach we pursued was to begin by designing an abstract description of a rope and a set of logical operations to transform the rope from one state to another. Because the exact structure of the rope was unknown, the description could not depend on absolute spatial coordinates. The development process applied the methodology of abstract data types to operations on physical objects (Liskov and Guttag 1986).

An abstract data type presents a programmer with abstract operations on an object. The specific structure of the object and the details of how operations change the object are hidden from view. In typical applications, the data type maintains an internal data structure that represents the state of the data abstraction. Operations have a deterministic effect on the representation. The initial state of the representation is known, and each operation changes it in a fixed and predetermined way.

Indeterminacy in the object state distinguishes the implementation of abstract data types for physical objects from their normal usage. The initial state of the object cannot be known exactly. Furthermore, actions have a nondeterministic effect on the objects. The state of a physical object must be inferred from sensory data. Perceptual and actions systems are needed to map the abstract operations onto a physical instance of the object. The interpretation of sensory input must be guided by the data type. The meaning of an image, for example, depends on the abstractions defined by the data type. The data type may or may not maintain an internal representation of the object. If an internal representation is used, then the data type must provide an interface to the perceptual system to allow creation and modification of the underlying data structures.

The value of programming with high-level constructs is well accepted in the field of robot language design. Robot languages can be characterized by the view of the world they present to the programmer. There are several classification schemes that define hierarchical taxonomies of robot languages (Lieberman and Wesley 1977; Lozano-Perez 1983; Ambler et al. 1987; Volz 1988). These taxonomies can not only be used to classify individual languages, but can also be viewed as multiple layers on which a single system can be constructed. Each successive level is built from the set of primitives provided by levels below.

Commercially available languages require program-

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

ming at the robot level. The programmer must directly specify movements of the robot. Movement primitives give the position, orientation, or compliance of a point on the end effector. Lower levels solve the inverse kinematics necessary to determine joint positions and plan trajectories (Finkel et al. 1975; Paul 1976; Shimano 1979; Taylor et al. 1982; Geschke 1983). Experience has shown that robot-level programs are difficult to write, edit, debug, and modify (Gini and Gini 1985). Part of the difficulty is caused by the unnaturalness of metric definitions of three-dimensional positions and orientations. More fundamentally, the dependence on precise information limits the generality and applicability of robot programs. When objects are flexible, when object shapes are complex, or when motion sequences are intricate, it is difficult to obtain precise information about object location and to enforce fine tolerances on motions.

To remedy the awkwardness and limitations of robot-level programs, researchers have studied methods to elevate the level of abstraction in robot languages. Object-level languages allow the programmer to specify geometric contraints on object positions (Popplestone et al. 1980; Popplestone and Ambler 1983; Latombe et al. 1985). Each statement specifies a partial constraint on the relationship between objects. The command

"**place** *face 1 of object B* **against** *face 2 of object C*"

states that surfaces on two objects should be in contact. If *face* 1 and *face* 2 are planar, then the statement means that the surfaces should be coplanar with surface normals pointing in opposing directions. The compiler solves a system of constraints given by a set of statements using internal object models. Object-level programs are translated into robot-level programs that specify the robot actions necessary to achieve the desired object configurations.

In *task-level* languages, the program specifies the functions to be performed or tasks to be accomplished (Lieberman and Wesley 1977). The ambition is to have language constructs at the level of assembly operations such as:

insert *peg 1* **into** *hole_a.*

Lower levels would devise a grasping strategy, determine error recovery methods, and plan a collision-free path for moving the peg.

Significant research problems must be solved before object- and task-level programming languages are feasible. Recent attention has focused on the study of supporting functions such as grasp selection, error avoidance, error recovery, and motion planning (Brooks 1982; Lozano-Perez et al. 1984, Cutkosky 1985). Although considerable progress has been achieved in understanding these subproblems, little effort has been made to integrate this knowledge into working robot systems. If high-level languages are to be built on robot-level languages, then these languages must provide a suitable foundation for large-scale software development. Unfortunately, robot-level languages have not been designed with this purpose in mind. The tools of software engineering are missing from most of these languages. Mechanisms for user-defined data types, information hiding, and separate compilation are unavailable in most robot-level languages. In many languages, modularization can only be accomplished with macro substitution.

We chose to build our system on the substrate of a general-purpose programming language augmented with operations for controlling the robot and acquiring images. Programs were written in the C language on a Sun color workstation running the Unix operating system. This environment provided us with good editing facilities, graphic and image display capability, debugging tools, and a variety of helpful utilities. Our work relied heavily on these features.

4. A Parametric Language for Knot Programming

The rope is abstractly viewed as a graph. The edges of the graph represent uncrossed segments of rope. The vertices of the graph correspond to a self-crossing or an end of the rope. Four segments are incident on all vertices that arise from a crossing. Because a single strand of rope was used, the edges could be ordered from one end of the rope to the other. Following the convention of knot-tying books, one end of the rope is designated the *free end* and the other the *standing end* (Ashley 1944; Day 1947).

All knot commands address the rope by its graph structure. References are given parametrically. For example, a point is specified by a segment and fraction. When a command is executed, a point that divides the corresponding segment of the physical rope in proportion to the predetermined fraction is located.

A graph representation of the rope is created with a vision system. The contour of the rope, in the image, is attached to the edges of the graph. Positions and orientations on the rope are mapped into robot coordinates when arm motion commands are generated.

Knot commands prescribe a place to grasp the rope and a path along which to move the rope. The grasp location is specified as a proportion of a segment. The destination is defined with respect to some other point on the rope. An intermediate point constrains the path followed to reach the destination point. This point is defined as a deviation from the straight-line path. Orientations, as well as positions, are always defined

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

with respect to the orientation of the rope at some point. Thus all references are bound to the structure of the rope however it happens to lie at the time the command is executed.

The knot data type maps parametric knot operations to an instance of a physical knot and sends motion commands over an asynchronous communications channel to the PUMA controller. The data type also assumes responsibility for determining grasp orientation and wrist motion. Early experiments demonstrated that the rope could unpredictably flip if significantly twisted. To avoid this problem, the orientation of the rope held in the hand must smoothly follow the hand trajectory. The rope was grasped by two opposing fingers. The parallel grippers could be in either of two orientations that differed by 180°. Because the robot hand has a limited range of wrist motion, care had to be taken to grasp the rope with the maximum range of wrist motion in the direction to be taken. If there still was insufficient range of motion to accomplish the wrist rotation, then the rope was set down midway through the motion and the wrist was reversed.

4.1. Knot Operations

The syntax of low-level knot commands is described in this section.

Positions and Frames

A *position* specifies a point on a segment of the rope. *Positions* are defined relative to the structure and shape of the rope at execution time. The segments of the rope are numbered from 0 to *N*, beginning at the standing end. A *segment_position* refers to a point on a segment as a fraction along the segment's length:

<segment_position> ::=*<segment_number><proportion>*

where

$0 \leq$ *<segment_number>***<number of segments**

and

$0 \leq$ *<proportion>* ≤ 1.

Points not on the rope are defined relative to a *segment_position*. A local coordinate frame is placed on a *segment_position*, and a displacement vector is specified in polar coordinates:

<relative_position> ::=*<segment_position><displacement>*

where

<displacement> ::=*<distance>*$<\theta>$.

We assume that $-\pi \leq \theta, \phi \leq \pi$ and *distance* ≥ 0.

Distances are measured in units equal to the length of the segment in the defining *<segment_position>*.

When specifying a desired placement for a section of the rope, both the position and orientation of the rope must be defined. A *frame* combines position and orientation information:

<frame> ::=*<relative_position>*$<\phi>$.

A *<frame>* is used to define a goal position and orientation with respect to a parametrically defined point on the rope.

Paths

A *path* constrains how a section of rope will be moved. Consider a point on the rope with position P_0 and orientation ϕ_0 in an arbitrary coordinate system. Let the goal destination for a move operation be a frame with location P_1 and orientation ϕ_1. The trajectory of the hand from the initial to goal states is determined by a single intermediate point, P_m, through which the rope point must pass. The point P_m is specified by a single signed value, the *excursion*. The excursion specifies the deviation from a straight-line path from the initial to the goal points. It is defined in a coordinate frame that has an origin at the midpoint of P_0P_1, is oriented in the direction of P_0P_1, and has units $\|P_0P_1\|$. A vector of length *excursion*$\cdot\|P_0P_1\|$ is placed at the midpoint of P_0P_1 oriented at $\pi/2$ to P_0P_1 if the sign of the *excursion* is positive and at $-\pi/2$ to P_0P_1 if the sign of the *excursion* is negative. Joint interpolated motion is used to move the robot from P_0 to P_m to the goal destination P_1.

A second parameter is needed to specify whether the orientation of the rope will be rotated in a clockwise or counterclockwise direction from ϕ_0 to ϕ_1. Thus the complete specification of a *path* is given by

<path> ::=*<excursion><direction>*

where

<direction> ::=**clockwise|counterclockwise|min-angle.**

If *<direction>* is **min-angle,** then the rope is rotated through the minimum angle to get from ϕ_0 to ϕ_1.

Operations

The language includes four commands:

>**grasp** *<segment_position>*—Grasp the rope at segment_position.

move *<frame><path>*—Move from the current position and orientation to the position and ori-

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

entation specified by *frame* along a route specified by *path*.

drop—Drop the rope at the current position.

create—Create a new representation of the knot with the vision system.

An example with two commands and the geometric interpretation of the positions and paths defined by the commands is presented in Figure 2.

An Example: The Figure 8 Knot

A portion of a program for creating a figure 8 knot is shown in Figure 3. Before this segment is executed, the rope will have been configured as in Figure 2. This segment forms the shape of an 8. First, the free end is crossed over the base of the rope. The second move statement loops the free end back toward the standing end, where it is dropped. Lastly, the standing end is grasped and carried over the free end.

4.2. The Vision System

The vision system provides a description of the physical rope. This description binds an abstract representation composed of segments and crossings to three-dimensional contours along the skeleton of the rope. An image from the overhead camera is thresholded to create a binary image of rope and background. A knot

	segment place, $\mathbf{p}_0$
grasp	0 .98

	reference place, $\mathbf{p}_r$	*destination place*, $\mathbf{p}_1$ *position*	*destination place*, $\mathbf{p}_1$ *orientation*	*path excursion*	*path direction*
move	0 .35	.10 -135	-90	.25	counterclockwise

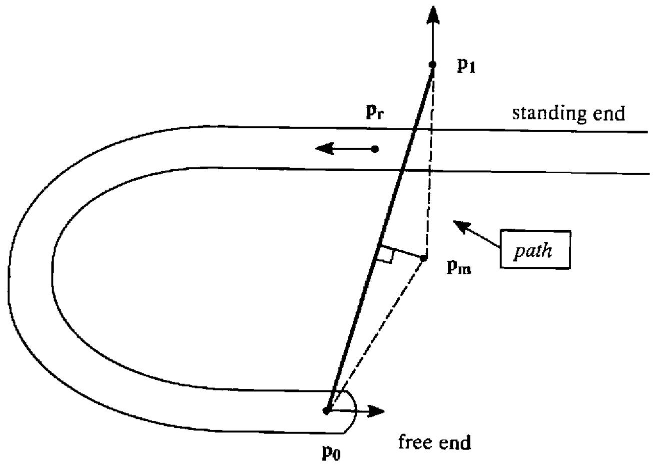


Fig. 2. Two commands from a program to tie a figure 8 knot and their geometric interpretation for an instance of the rope.

```
grasp 0.95
move 0.35 .10 −135 −90 .25 counterclockwise
move 0.1 −.05 90 90 .15 clockwise
drop
create
grasp 0.5
move 0.5 −.28 90 0 0 min-angle
drop
```

Fig. 3. A portion of the program for tying a figure 8 knot.

structure is generated by following the skeleton of the rope from the standing end to the free end. An example of the camera's view of the rope is given in Figure 4.

First, a point on the initial segment is found by searching the border of the image. This can be done because the rope trails off the end of the table and out of view. A point on the opposite side of the rope is located by searching perpendicular to the edge of the rope. The skeleton of a segment is determined by parallel tracking of opposite sides on the rope cross section. Edge points are iteratively advanced along the borders of the rope until the separation between the points is significantly larger than the expected width of the rope. On each iteration, the line segment between the edge points is compared with the direction of the midline contour. The lagging edge point, as determined by the angle between the connecting segment and the contour direction, is always advanced. The effect is that the two points shimmy along the rope boundary, sliding an approximately perpendicular cross section along the rope contour.

The rope crossings present some difficulty. The flexibility of the rope precludes projection of the midline across the crossing; when the crossing segments are nearly parallel the likelihood of traversing the wrong segment is high. Instead, the crossing is mapped

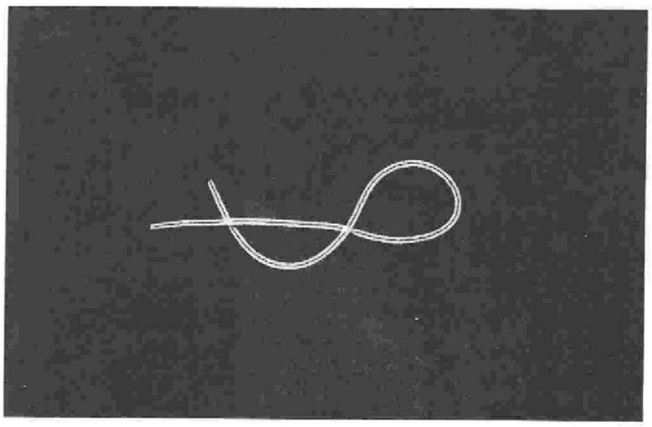


Fig. 4. The contours generated by the vision system are drawn on the rope image.

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

by following the crossing segments. The opposite sides of the two crossing segments are then followed back to the continuation of the crossed segment.

This process is repeated until the end of the rope is found. A knot graph is constructed as the rope is traversed and skeletal contours are attached to graph edges. The countours found by the vision system for an intermediate step in tying the figure 8 knot are shown in Figure 4.

5. Graphic Editing and Debugging

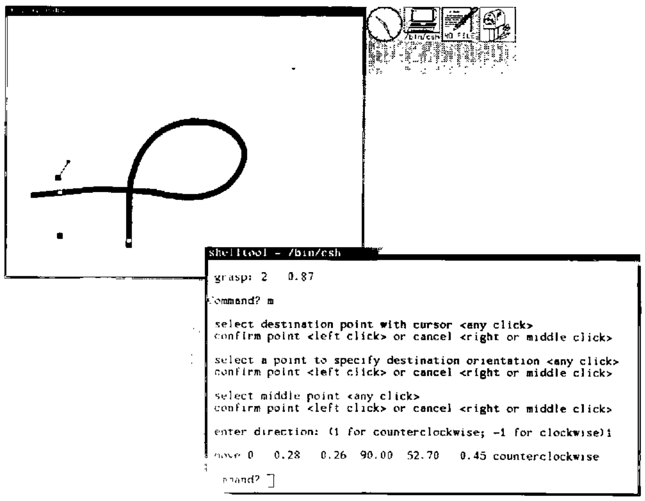
Robot-level programming languages have been frequently criticized for requiring programmers to provide numeric specifications of three-dimensional positions and orientations. These measurements are difficult for programmers to originally derive and are later difficult for programmers to understand and modify. A number of robot-level languages have been augmented with physical positioning devices to simplify coordinate frame definition (Gini and Gini 1985; Latombe et al. 1985).

We found that knot-programming commands were also cumbersome to write and difficult to understand once written. Software was developed to allow knot commands to be graphically defined and displayed.

A graphic tool permits a programmer to interactively step the robot through a series of knot commands specifying robot motions by pointing with a cursor on an image of the rope. The programmer can select one of the four knot commands (**grasp, move, drop,** or **create**) in a text window. When a command requires geometric arguments, the user is prompted to position the cursor on the image and click the mouse button. For example, the **move** command requests in succession a destination point, a second point to specify the desired orientation of the rope at the destination, and a midpoint to determine the path excursion. The user is last asked to choose the direction in which to rotate the rope from the initial to goal orientation.

Following each command specification, the operation is executed. A **create** command causes the image to be updated and a new representation of the knot to be created. For movement commands, the robot executes the prescribed motion by grasping, moving, or dropping the rope. The system also synthesizes a parametric knot command from the absolute positions and orientations chosen by the user. Each parametric command is printed on the screen as it is synthesized. Given an exemplar sequence of rope manipulations, a general program can be assembled.

Figure 5 illustrates the graphic interface after generation of a **grasp** and **move** command sequence. A **create** command caused the system to generate an up-to-date description of the rope. Next, the free end



```
grasp: 2   0.87

Command? m

select destination point with cursor <any click>
confirm point <left click> or cancel <right or middle click>

select a point to specify destination orientation <any click>
confirm point <left click> or cancel <right or middle click>

select middle point <any click>
confirm point <left click> or cancel <right or middle click>

enter direction: (1 for counterclockwise; -1 for clockwise)1

move 0   0.28   0.26  90.00  52.70   0.45 counterclockwise

 mmand? ]
```

Fig. 5. The graphic interface for specifying knot commands.

of the rope was grasped. The user's choice for the grasp point is marked by a filled square near the end of the rope. The system determined the nearest point on the rope contour, shown as an open square. The contour point was translated into a scaled length along the structurally ordered segment to form the parametric **grasp** command appearing in the text window.

The user then directed the system to move across the first segment. The two points entered to determine the goal position and orientation are indicated by a filled square located at the first selection and a line segment connecting the first and second selections. The graphically specified destination and orientation were mapped into a relative position and orientation with respect to a nearby point on a rope segment. The reference point is indicated by an open square. The user selected one additional point to determine the excursion parameter for the move and the direction of rotation. Lastly, the constructed **move** command was printed in the text window.

The graphic display was also very helpful for debugging system software. When robot commands were added or modified, they first were tested by projecting the motions into image space and drawing the result on the display. Gross errors were detected without risk of damage to the robot.

6. Parser and Compiler Generation for Higher Level Languages

A second, more abstract layer of knot-tying operations was built on top of the parametric language. This language raises references from the level of points and paths to the level of segments and segment crossings. A sequence of operations in the parametric language is compiled from a single segment command.

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

The segment-level language was designed to allow symbolic names to be attached to parts of the knot. In the low-level language, all references to a knot are expressed in terms of the structure of the rope. Length and position expressions are bound to the physical rope as it exists the moment the reference is made. The physical meaning of a reference can change as the rope is manipulated. For example, the Nth segment will become the $N + 1$st segment whenever a segment $0..N - 1$ is crossed. Small modifications to the object can have global consequences for the referencing scheme. This is inconvenient, because the programmer must do a great deal of bookkeeping to keep track of a particular piece of rope. By attaching labels to important entities, such as segments, the programmer can persistently refer to a structural element by name. In the segment-level language, names persistently refer to the same logical segment.

The fundamental command for structuring segments is **cross.** A cross statement directs a crossing of the last segment (the segment that includes the free end) over or under another segment. The cross command optionally allows names to be attached to new segments as they are created.

Using the cross command the end of the rope can be interwoven through the knot. This provides a simple means to tie many knots. The syntax for cross is:

> **cross** {**over**|**under**}
> *<seg-name>***from**{**right**|**left**}
> [**curving**[**widely**|**sharply**]{**right**|**left**}]
> [**creating***<endname>*]
> [**splitting***<leftname><rightname>*].

The elements in curly brackets are mandatory—one must be chosen. The elements in square brackets are optional. This command causes the last segment to cross **over** or **under** segment *<seg-name>*. The direction of the crossing is determined by the parameter following the **from** term. The **left**|**right** orientation is defined for someone standing on the rope facing the free end. The optional **curving** element guides the compiler in the selection of the path to the crossing.

The last two elements associate names with the newly created segments. Initially, a single segment named "rope" is defined. Each cross operation can create up to three new named segments. The name *<endname>* in the **creating** element is assigned to the segment that starts at the new crossing and continues to the free end. The **splitting** element assigns names *<leftname>* and *<rightname>* to the left and right segments created as the free end crosses the *<seg-name>*. The names are illustrated in Figure 6. Names continue to be associated with the length of rope between two logical crossings as the structure of the knot is modified by the addition of new crossings. If the segment is split by a subsequent crossing, the name refers to a set of segments between the original crossings.

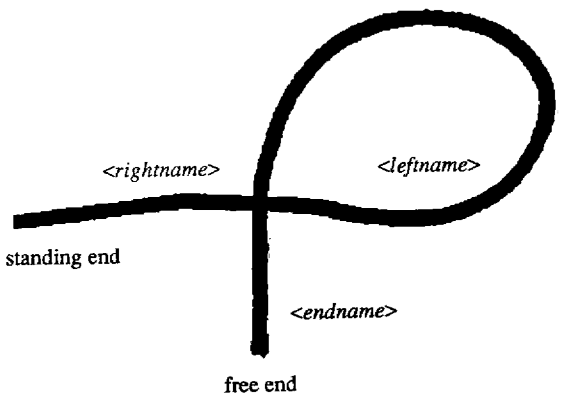


Fig. 6. Naming conventions for the segment-level language.

A second command was added to the segment-level language to allow modification of the geometry of the last segment. The **extend** command specifies simple shaping and enlargement of the last segment necessary to tie certain knots. The syntax is

> **extend** [**a little**|**a lot**]
> [**curving** [**widely**|**sharply**] {**right**|**left**}].

This command causes the last segment to be extended and possibly turned to form a J shape. The modifying parameters **a little, a lot, widely,** and **sharply** provide rough guidelines for the amount of extension and curvature of the turn.

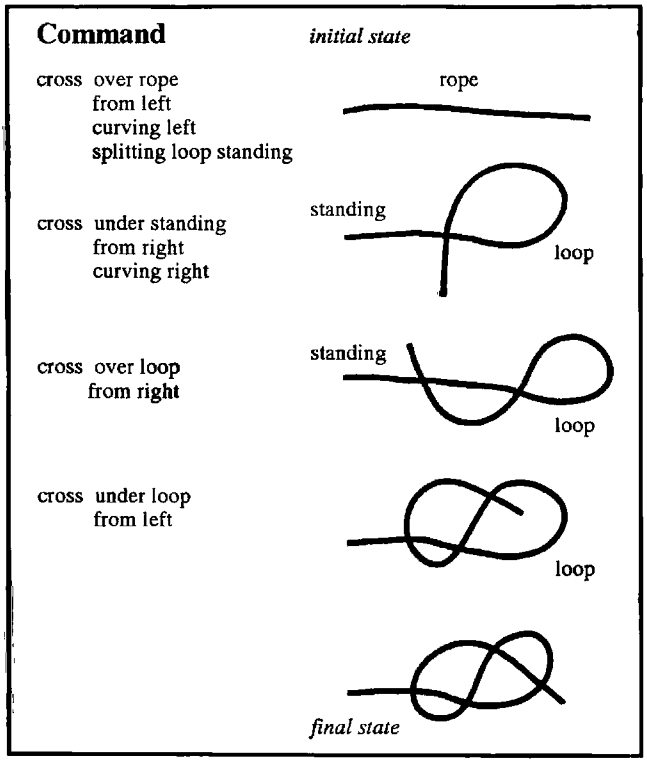
The language is best understood through an example and some illustrations. The program for a figure 8 knot is given in Figure 7, together with the form of the rope and the named segments created by each command.

The segment-level language was implemented using the Unix utilities *yacc* and *lex.* A grammar was defined and then converted into a parser and compiler for the language. The compiler makes two passes over a knot program. In the first pass, a logical knot is created by simulating the execution of the knot program. An abstract representation of the knot is built as the compiler steps through the program. The abstract knot has the topology of the actual knot but is attached to no specific contour. In the second pass, the compiler adjusts segment lengths to assure that when the segment is made it will be of sufficient length to permit later crossings. The compiler determines specific crossing points and segment lengths.

For the sake of concreteness, the following overview of the compilation of a **cross over** command is presented:

1. The parser recognizes the **cross over** tokens,

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012


Command

```
cross  over rope
       from left
       curving left
       splitting loop standing

cross  under standing
       from right
       curving right

cross  over loop
       from right

cross  under loop
       from left
```

Fig. 7. A segment-level program for tying a figure 8 knot with illustrations of the rope state at each step.

parses the command, and returns tokens for each of the potential clauses: *segment, fromdirection, curve, create, and split.*

2. Low-level commands to capture an image and grasp the free end of the rope are assembled to form a skeletal sequence of low-level commands.
3. The move command that the will do the actual work of the crossing is appended to the command skeleton. The parameters of the move are determined from the *segment, fromdirection,* and *curve* clause information.
4. A crossing is created in the abstract knot using the naming and direction information in the clauses.
5. A drop command is added to the skeleton.

The basic structure of the low-level command sequence is determined by these five steps. However, all parameters are not fixed until a second pass is made over the complete knot program. Prior to the second pass, the final abstract knot is inspected, and segments are assigned equal lengths. During the second pass, parametric positions are assigned in the grasp and move commands based on the desired final configuration. At this point, the knot program is complete and ready to execute.

Four low-level commands are assembled to implement a single **cross over** command. A **cross under** requires a series of **grasp, move,** and **drop** sequences. First, the end of the rope is brought near the segment to be crossed and dropped. The segment is then lifted over the free end and finally the end of the rope is pulled under the segment to the destination position. In other respects, compilation of a **cross under** statement proceeds as above.

7. Summary

The knot-tying system was used to construct programs for a variety of common knots. Knot programs were tested with $\frac{1}{4}$- and $\frac{1}{2}$-inch width nylon ropes. Initially the unknotted rope was placed in a random orientation near the center of the work space. Experiments demonstrated that modestly complex knots could be robustly tied. The most complicated knot successfully programmed was a bowline requiring six crossings.

The current implementation has a weak error detection mechanism and no error correction capability. Parametric motions adapt to the rope with indifference to form or arrangement. A command that references the *n*th segment is valid if there are at least *n* segments in the knot. If the referenced segment does not exist, then an error message is printed and the program terminates. Little more error detection can be done with the information provided by parametric commands. The program makes no explicit statements about the desired structure or shape of the rope.

The segment-level language affords a greater opportunity for error analysis. The intended topology is directly expressed in the **cross** commands. Unsuccessful crossing operations, incidental undoing of prior crossings, and unplanned crossings can lead to structural abnormalities. These errors could be detected by periodically comparing the topology of the knot graph obtained with the vision system to the topological structure implied by the series of crossing commands. During compilation, the present system incrementally constructs an ideal knot by simulating the action of segment-level commands. This abstract model has the topological structure of the intended knot but the contours are absent. However, this information is not retained in the compiled program. Error checking could be introduced by interpreting rather than compiling the segment level language. Error correction raises a number of difficult planning problems beyond the scope of the present system.

Neither of the current languages explicitly describe the geometry of the rope. For simple knots, the two-dimensional topology of the rope provides sufficient constraints to accomplish the task. However, for more

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

complicated knots, the geometry of uncrossed segments is also important. Most knots can be tied in a variety of ways. Many common techniques require that segments of the rope be formed into shapes in preparation for future operations (Ashley 1944; Day 1947).

Figure 8 demonstrates two steps in tying a sheepshank. The rope is first formed into an elongated S shape. More global geometric descriptions are needed to characterize some complex knots. For example, it is important that the three loops in Figure 9 be colinear and approximately the same size. An ability to characterize the important geometric properties of segments and relationships among parts of the knot is needed to determine the success or failure of manipulations with intricate knots. The design of shape representations adequate to express the qualitative properties of complex, self-intersecting figures is a challenging problem.

The purpose of this project was to explore programming of physical operations on objects where it is impossible to precisely specify the actions to be performed. The knot-tying study demonstrated the difficulties that arise in constructing robust, nonmetric robot programs. In working on the project, the fundamental importance of two notions, programming on abstractions and the interrelated roles of language, action, and perception in doing so, became apparent. Appropriate abstractions allow the programmer to address the properties of the object, not the actions of the robot. Parametric motion commands are a step toward more qualitative languages. The experience also emphasizes how programming tools can contribute to building robot programming systems.

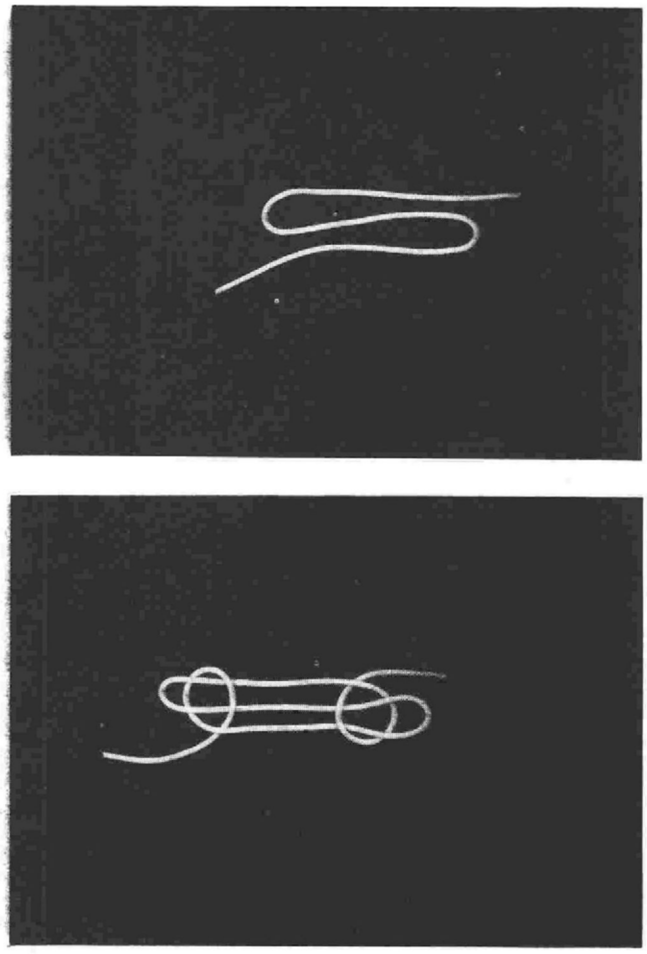


Fig. 8. Two steps in tying a sheepshank knot.

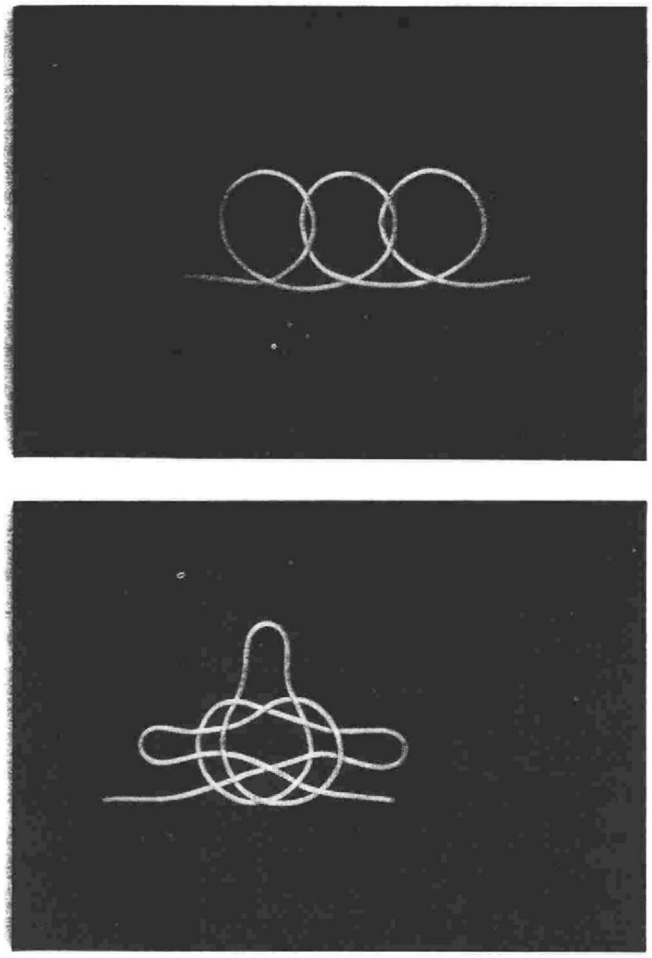


Fig. 9. Two steps in tying a masthead knot.

Acknowledgments

This research was supported in part by National Science Foundation Grants DCR85-02568, DCR86-17355, and IRI-8808896 and Office of Naval Research Grants ONR 00014-86K-0281 and N0014-88-K-0632. We would like to thank Douglas Cambell for his valuable assistance.

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012

References

Ambler, A., Cameron, S. A., and Corner, D. F. 1987. Augmenting the RAPT robot language. In Rembold, U., and Hörmann, K., eds.: *Languages for Sensor-based Control in Robotics.* New York: Springer-Verlag, pp. 305–316.

Ashley, C. W. 1944. *The Ashley Book of Knots.* Garden City, N.Y.: Doubleday, Doran and Co.

Brooks, R. 1982. Symbolic error analysis and robot planning. *Int. J. Robot. Res.* 1(4):29–68.

Cutkosky, M. R. 1985. *Robot Grasping and Fine Manipulation.* Boston, Mass.: Kluwer Academic Publishers.

Day, C. L. 1947. *The Art of Knotting and Splicing.* New York: Dodd, Mead and Company.

Finkel, R., Taylor, R., Bolles, R., Paul, R. P., and Feldman, J. 1975. An overview of AL: A programming system for automation. *Fourth International Joint Conference on Artificial Intelligence,* pp. 758–765.

Geschke, C. C. 1983. A system for programming and controlling sensor-based robot manipulators. *IEEE Trans. Pattern Anal. Machine Intell.* PAMI-5(1):1–7.

Gini, G., and Gini, M. 1985. Dealing with world-model-based programs. *ACM Trans. Program. Lang. Sys.* 7(2):334–347.

Inoue, H., and Inaba, H. 1984. Hand-eye coordination in rope handling. In Paul, R., ed.: *Robotics Research: The First International Symposium,* Cambridge, Mass.: MIT Press, pp. 163–174.

Latombe, J. C., Laugier, C., Lefebvre, J. M., Mazer, E., and Miribel, J. F. 1985. The LM robot programming system. In Inoue, H., ed.: *Robotics Research: The Second International Symposium.* Cambridge, Mass.: MIT Press, pp. 377–391.

Lieberman, L. I., and Wesley, M. A. 1977. AUTOPASS: An automatic programming system for computer controlled mechanical assembly. *IBM J. Res. Develop.* 21(4):321–333.

Liskov, B., and Guttag, J. 1986. *Abstraction and Specification in Program Development.* Cambridge, Mass.: MIT Press.

Lozano-Pérez, T. 1983. Robot Programming. *Proc. IEEE* 71(7):821–841.

Lozano-Pérez, T., Mason, M., and Taylor, R. 1984. Automatic synthesis of fine-motion strategies for robots. *Int. J. Robot. Res.* 3(1):3–24.

Paul, R. 1976. WAVE: A model-based language for manipulator control. Society of Manufacturing Engineers, Technical paper MR 76-615, pp. 10–17.

Poplestone, R., and Ambler, A. A language for specifying robot manipulations. In Pugh, A., ed.: *Robotic Technology.* London: Peter Peregrinus, pp. 125–141.

Poplestone, R., Ambler, A., and Bellos, I., An interpreter for a language for describing assemblies. *Art. Intell.* 14:79–107.

Shimano, B. 1979. VAL: A versatile robot programming and control system. *COMPSAC 79 Conference Proceedings,* pp. 878–883.

Taylor, R., Summers, P., and Meyer, J. 1982. AML: A manufacturing language. *Int. J. Robot. Res.* 1(3):19–41.

Volz, R. 1988. Report of the robot programming language working group: NATO workshop on robot programming languages. *IEEE J. Robot. Automat.* 4(1):86–90.

Downloaded from ijr.sagepub.com at SAGE Publications on December 6, 2012