1-1-1992

An interactive graphical approach to off-line programming

James J. Troy
Iowa State University

An interactive graphical approach to off-line programming

by

James J. Troy

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Major: Mechanical Engineering

Iowa State University
Ames, Iowa
1992

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

In order to clarify the mathematical notation used in this thesis, a brief explanation may be useful. Coordinate frame transformations will be specified by an upper case T with leading superscripts and subscripts in the form ${}^{B}_{A}T$, which describes frame A relative to frame B. Upper case characters inside brackets, $\{A\}$ for example, refer to a specific coordinate system. An upper case P with leading superscript and following subscript denotes a 3-D vector, ${}^{B}P_{A}$ for example, which relates the origin of reference frame A to the origin of reference frame B. The word "frame" is sometimes used in place of "coordinate system", and should not be confused with the frames (images) of an animation.

ACKNOWLEDGMENTS

I would like to thank my major professor and friend Martin Vanderploeg, as well as the members of by committee, Professors James Bernard and John Jackman.

In addition, I would also like to thank fellow graduate students (and former students) in the Visualization Lab: Terran Boylan, Jim Lynch, Afshin Mikaili, Jay Shannan, Todd Teske, and Jeff Trom, for their comments, advice, and diversions during the course of by research.

And finally, I thank my parents for their encouragement throughout my many years in college.

1. INTRODUCTION

Off-line programming is the creation of a set of instructions to control robot manipulators and other programmable devices of a workcell without actually using the equipment. The addition of a computer graphics interface greatly enhances the usefulness of off-line programming. The advantages of programming a robot in this manner include:

- The ability to visualize the arrangement of a workcell layout before equipment is purchased
- Creating and testing device control code without taking equipment out of production
- The convenience of being able to program different types of robots using a common graphics based instruction set
- Faster program modification
- Increased safety

This thesis describes the initial stages in the development of interactive graphical software for off-line programming. The approach taken here uses "solid" models to represent all components of the robot's workcell, and displays the simulated movement of these components through animated computer graphics. The main topics covered in this thesis will be forward and inverse kinematics, path generation, and the creation of an interactive graphical interface. The application of these topics has led to off-line programming software that was tested by simulating a five degree of freedom articulated robot. Simulation data was then translated to the robot's device control code and tested on the actual robot.

Kinematic position generation will be used to generate all motion (i.e., the dynamic properties of mass and inertia will not be taken into account). The reason for this approach is that most production robots in use today are driven by electric servo motors with high gear reduction ratios. Generating motion in this manner creates high frequency vibrations which have quick settling times. Robots of this type are considered "stiff" position control devices and do not usually require dynamic models to obtain adequate simulations.

Software of the type presented here is already sold commercially but is quite expensive and source code is usually not available. The availability of the source code is necessary to make modifications and enhancements, and to have complete control over the user interface. The main objective of the research presented in this thesis is the development of graphical off-line programming software for a specific robot, that can be modified to allow additional robot models (and other programmable devices) to be added later.

The off-line programming software (named RS, for ***Robot Simulator***) developed during this research is written in C and makes extensive use of the graphics routines of Silicon Graphics' Graphics Library (GL). Testing of the device control code written out by RS was performed on a five degree of freedom Mitsubishi RV-M1 robot. In order to obtain feedback on the use of this software, testing was also performed by undergraduate students with varying degrees of computer graphics and robotics experience. Results include comments about the usefulness of this software, as well as modifications needed to improve its functionality.

2. LITERATURE REVIEW

This literature review gives an overview of research in off-line programming, starting with the historical evolution of robot programming followed by a summary of the theoretical background of robot simulation and off-line programming.

2.1 Evolution of Robot Programming

In order to get a better understanding of why graphical off-line programming has become an important aspect in robotic workcell design, it is useful to discuss the evolution of robot programming.

When robots were initially introduced, the only method of programming was teaching manipulator positions on-line. On-line programming involves directing the robot to the desired goal position using a teach pendant and then recording the position directly into the memory of the machine controller. A variety of proprietary languages, like AML and VAL II, are now available that can be used with the teach pendant to give better on-line programming control [1].

Early off-line programming evolved from advances in computer numerical control (CNC). Some off-line programming software used high level languages like BASIC [2], but lacked capabilities necessary to graphically simulate the program before transferring it to the machine controller.

Derby [3], and Patt and Derby [4] developed PC based software that used wireframe computer graphics to simulate robot workcells. Due to computer hardware limitations, these programs lacked the ability to be dynamically interactive (which is the ability to calculate and graphically display changes as an input variable is being changed from one state to another). This type of interaction is necessary to give the robot programmer a more complete understanding of the robot's motion and to create a more efficient off-line programming environment.

More recent software developments have been designed for use on more powerful graphics workstations. These software packages are capable of simulating and animating solid models at relatively high update rates. Packages like *World Modeler*, developed by Mirolo and Pagello [5] and *Jack* by Phillips et al. [6], have many advanced functions, but most research of this type has focused only on graphical simulation. Since these types of programs lack the ability to translate graphical simulation data into the device control code needed to drive the robot, they cannot be used for off-line programming.

Interactive computer graphics software with off-line programming capabilities are commercially available such as, *CimStation* by Silma and *IGRIP* by Deneb (see [7] [8] [9]). These programs have many advanced features: collision detection, signature models, and dynamic simulation, as well as the ability to write device control code. But these programs often cost more than the hardware on which they are run. With the recent introduction of fast and relatively inexpensive graphics workstations, this cost differential becomes more significant.

2.2 Theoretical Background

One of the most important aspects of off-line programming is that of the underlying kinematic and dynamic equations that control the position and movement of the robot. Most approaches to kinematic simulation are based on notation developed by Denavit and Hartenberg [10], which describes a method of defining coordinate frames attached to

moving links. These frame descriptions are used to develop 4×4 transformation matrices which give the relative positions of one link to another. Ho and Sriwattanathamma [11] present a symbolic matrix manipulation program to automate the derivation of the link transformations from the Denavit-Hartenberg parameters

Several approaches exist for deriving inverse kinematics. These can be broken down into two general categories: numerical and closed form solutions. Stone [12] presents numerical solution methods based on Newton-Raphson and Jacobi iterative algorithms. Phillips et al. [6] present a more computationally efficient numerical approach than traditional numerical solutions, but these still suffer from numerical convergence problems. Fu et al. [13] discuss techniques to derive closed form solutions to the inverse kinematics problem that are much more efficient than numerical methods. Among these are algebraic, geometric, and quaternion based methods.

Although dynamics will not be dealt with explicitly in this thesis, much work in robotics deals with this topic. Fu et al. [13] discusses various closed form and numerical solutions for forward and inverse dynamics. Nikravesh [14] discusses methods of formulating equations for multi-body dynamics, which can be solved by numerical methods. Isaacs and Cohen [15] present methods of producing dynamic simulation systems for computer animation. Davis [16] presents experimental results from a modal analysis of a "stiff" position control robot.

Path generation algorithms which are used to control trajectory in three dimensions are based on either joint space interpolation or cartesian space interpolation. Fu et al. [13] and Craig [17] describe joint space methods which are based on fitting splined curves through a set of predefined control points. Fu et al. [13] describes cartesian space methods (to trace straight lines in 3-D space) using homogeneous transformation matrix and dual quaternion approaches.

Another area of robotics research that is important in developing efficient off-line programming systems is the development of robot signature models. Signatures are used to covert the ideal positions developed in a robot simulation to a set of corresponding positions which compensate for manufacturing and calibration variations of an individual robot. Stone [12] develops a signature modeling technique in which the actual kinematic parameters of an individual robot can be identified and used to create correction functions for that robot.

In order to write more "intelligent" device control code, off-line programming software can be used in conjunction with task level programming. Latombe [18] makes extensive use of computer graphics simulation in his work with robot motion planning to analyze and explain various artificial intelligence and spacial reasoning algorithms.

3. KINEMATICS AND PATH GENERATION

The basis for all graphical simulation involves defining the position of objects in three dimensions, and describing how the position of these objects change with time. In off-line programming, the positioning problem involves locating the manipulator (hand or tool) at certain precision points within the robot's workspace. The following sections will focus on position and motion generation for a five degree of freedom articulated robot, specifically dealing with coordinate systems, forward and inverse kinematics, and path generation.

3.1 Coordinate Systems

The coordinate system used in this thesis is based on Denavit-Hartenberg [10] notation for lower-pair mechanisms. This method identifies link parameters that describe the position of each link relative to an adjacent link. Each link is described by two angles: α_{i-1} and θ_i, and two linear offsets: a_{i-1} and d_{i-1}. For any lower pair joint (revolute or translational) three of the four Denavit-Hartenberg (D-H) parameters are fixed and one is variable. For a revolute joint the variable parameter will be θ_i, for translational joints it is the link offset d_{i-1}. Figures 3.1a and 3.1b show these parameters as specified for rotational and translational joints respectively.

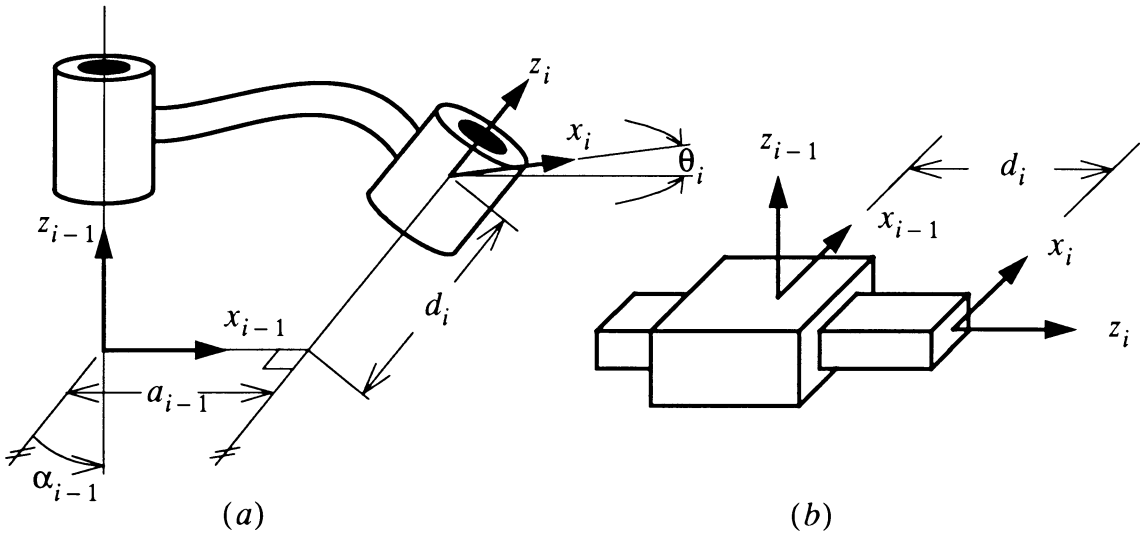


Figure 3.1 Frame definitions for a rotational joint (a) and translational joint (b)

The parameters are specified as follows:

- a_{i-1} = distance from z_{i-1} to z_i relative to axis x_{i-1}
- α_{i-1} = angle from z_{i-1} to z_i relative to axis x_{i-1}
- d_i = distance from x_{i-1} to x_i relative to axis z_i
- θ_i = angle from x_{i-1} to x_i relative to axis z_i

When assigning these parameters, it is useful to make a table listing the four values for each link. It is possible to have different parameter values depending on how the local coordinate frames of each link are assigned. However, any rotation or translation of the variable parameter takes place about or along the local z-axis.

For the five degree of freedom articulated robot, which will be analyzed throughout this thesis, the frame assignments and D-H parameter table are shown in Figure 3.2 and Table 3.1 respectively.

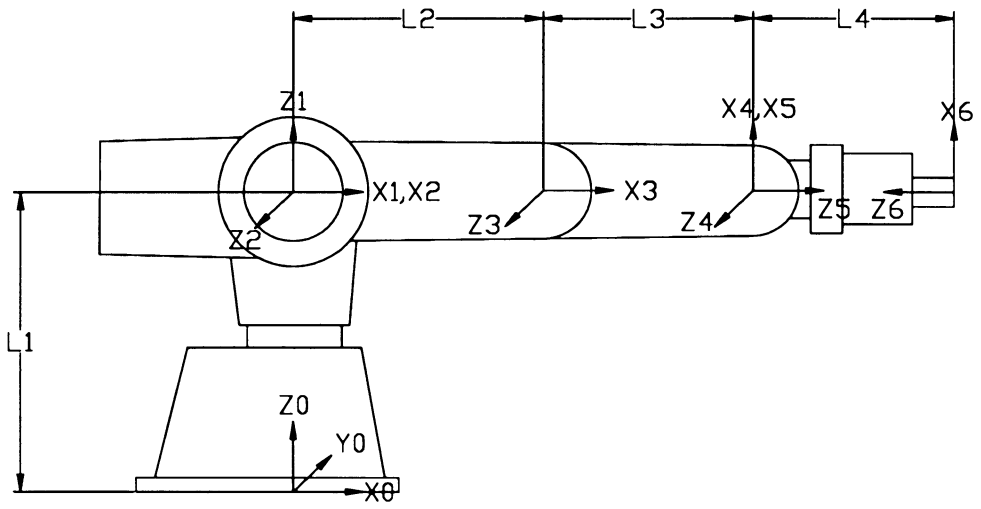


Figure 3.2 Frame assignments for a 5-DOF articulated robot

Table 3.1 D-H parameter table for the 5-DOF robot of Figure 3.2

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0°	0	L_1	θ_1
2	90°	0	0	θ_2
3	0°	L_2	0	θ_3
4	0°	L_3	0	θ_4
5	−90°	0	0	θ_5
6	180°	0	$-L_4$	0

Now that the D-H parameters have been assigned, they will be used to determine the link transformations using 4×4 matrices. The general equation for this transformation is given by Equation 3.1.

$$
{}^{i-1}_{i}T = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i\cos\alpha_{i-1} & \cos\theta_i\cos\alpha_{i-1} & -\sin\alpha_{i-1} & -\sin\alpha_{i-1}d_i \\ \sin\theta_i\sin\alpha_{i-1} & \cos\theta_i\sin\alpha_{i-1} & \cos\alpha_{i-1} & \cos\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}
$$

Once all the link transformation matrices have been defined, they can be multiplied together (in the correct order) to get the transformation matrix relating the coordinate system of any one link to any other link in the system. For example, the transformation matrix describing the position of link 3 relative to link 0 can be calculated as,

$$
{}^{0}_{3}T = {}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T
$$

In general, the transformation matrix of reference frame M relative to frame N is

$$
{}^{N}_{M}T = {}^{N}_{N+1}T\,{}^{N+1}_{N+2}T \cdots {}^{M-2}_{M-1}T\,{}^{M-1}_{M}T \tag{3.2}
$$

The transformations of each link of the 5-DOF robot, based on the D-H parameters of Figure 3.2, are given in Equations 3.3 to 3.8.

$$
{}^{0}_{1}T = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}
$$

$$
{}^{1}_{2}T = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}
$$

$$ {}^{2}_{3}T = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & L_2 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5} $$

$$ {}^{3}_{4}T = \begin{bmatrix} \cos\theta_4 & -\sin\theta_4 & 0 & L_3 \\ \sin\theta_4 & \cos\theta_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6} $$

$$ {}^{4}_{5}T = \begin{bmatrix} \cos\theta_5 & -\sin\theta_5 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin\theta_5 & \cos\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.7} $$

$$ {}^{5}_{6}T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & L_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.8} $$

These transformations will be used later in this chapter to derive the inverse kinematics of the robot.

In order to graphically display the link positions, the transformations of each link (as well as moving objects in the workcell) must be calculated relative to a stationary coordinate system. This involves premultiplying the transformations described in the base coordinate system by the base frame relative to the stationary frame transformation, ${}^{S}_{B}T$, as shown by Equation 3.9.

$$ {}^{S}_{T}T = {}^{S}_{B}T \; {}^{B}_{W}T \; {}^{W}_{T}T \tag{3.9} $$

where the indices S, B, W, and T refer to the stationary, base, wrist, and tool frames respectively, as shown by the frame diagram in Figure 3.3.

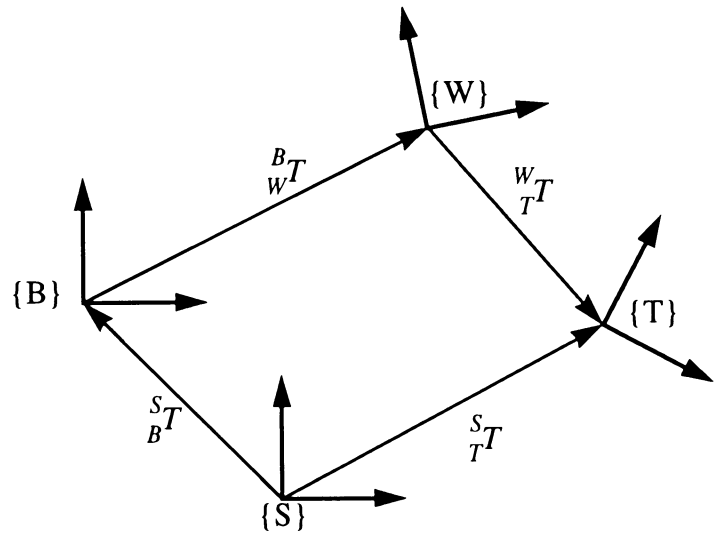


Figure 3.3 Generalized frame diagram

For the 5-DOF robot the $^{S}_{B}T$ and $^{B}_{W}T$ transformations are specified by the user, and the $^{B}_{W}T$ transformation is given by Equation 3.10.

$$^{B}_{W}T = {}^{0}_{5}T = {}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T\,{}^{3}_{4}T\,{}^{4}_{5}T \tag{3.10}$$

When deriving the inverse kinematics, it is sometimes necessary to invert a transform before premultiplying, such as,

$$^{4}_{2}T = ({}^{3}_{4}T^{-1})\,{}^{3}_{2}T$$

This can be accomplished using Equation 3.11,

$$ {}^{B}_{A}T = ({}^{A}_{B}T^{-1}) = \begin{bmatrix} \begin{bmatrix} {}^{A}_{B}R^{T} \end{bmatrix} & \begin{bmatrix} -{}^{A}_{B}R^{T}{}^{A}P_{B} \end{bmatrix} \\ 0\ 0\ 0 & 1 \end{bmatrix} \tag{3.11} $$

where R is the 3×3 rotation matrix and ${}^{A}P_{B}$ is the column vector of x, y, and z positions of frame B relative to frame A. Now that the transformations have been defined, it is possible to formulate the forward and inverse kinematic equations.

3.2 Forward and Inverse Kinematics

Positioning the links of a robot involves finding a set of joint variables and link transformations for a particular manipulator goal point, this can be accomplished in two ways: forward (or direct) kinematics, and inverse kinematics. Forward kinematics involves calculating the position of the robot's manipulator in cartesian coordinates, as a function of the joint variables. This is done by premultiplying the wrist transformation matrix by all previous link transformations until the position of the wrist relative to the base is found (as shown by Equation 3.10). Positioning the manipulator using only forward kinematics involves moving each joint independently in order to reach the desired cartesian goal point. Using this method, it is often very difficult and time consuming to get accurate manipulator placement.

Since the position of the manipulator is the desired quantity, and not the joint angles, it is more intuitive to specify the manipulator goal position in cartesian space and then compute the required joint angles. This is the basic concept behind inverse kinematics. In general, finding inverse kinematic equations is a much more difficult problem than deriving the forward kinematics; forward kinematics involves straightforward matrix multiplication, while inverse kinematics usually involves solving non-linear systems to obtain individual joint variables.

The solution to the inverse kinematics problem can be found by using one of two methods, a numerical approach or a closed form (analytical) approach. The numerical solution formulation is easier to develop and can be applied in a logical manner to any general mechanism (meaning the computer can generate and solve the equations). This generality has a major drawback; since numerical methods are based on iterative techniques, they require significant computational effort. At current levels of computing capabilities, an accurate solution cannot be obtained fast enough for realistic interactive operation for systems with many degrees of freedom. Higher efficiencies can be obtained from numerical solution techniques if larger tolerances on the solution are specified, and/or the solution to the current position is relatively close to the previous solution (see [12]). Unfortunately, large tolerances in the accuracy of kinematic solutions are usually not acceptable in robotic programming. Furthermore, requiring the solutions to be relatively close to each other means that the programmer would have to move the mouse (or other input device) slowly, since fast movements would cause large positioning displacements which would slow down the iterative convergence of the numerical solution algorithm.

The closed form approach offers a substantial increase in computational efficiency over the numerical approach. This type of solution can be calculated fast enough to allow systems with many degrees of freedom to be dynamically interactive, i.e., graphically displaying position changes at high update rates. The drawback of the closed form approach is in the development of the equations which explicitly solve for the joint variables. There are no completely general methods for deriving closed form inverse kinematic solutions (meaning a computer can't derive the equations on its own). In fact, closed form solutions are not possible for some types of mechanisms. Special conditions must be met for a closed form solution to exist: joints must be parallel or at right angles to each other. Fortunately, almost all robots are designed to meet this criteria. One additional advantage that a closed form approach has over a numerical one is the ability to easily find all solu-

tions to a positioning problem. Due to the limitations of the numerical method for inverse kinematic positioning, closed form approaches will be used in this thesis. The techniques for deriving closed form solutions that will be discussed include the algebraic and geometric methods, along with a hybrid method involving both algebraic and geometrical solutions.

The Algebraic Method

The algebraic solution technique involves symbolically multiplying the transformation matrices that make up the ${}^{B}_{W}T$ transformation of Equation 3.10. Trigonometric identities are then used to reduce combinations of certain elements of this transformation to equations involving a single variable. This method of pulling the individual joint variables out of the transformation matrix can be very difficult (or impossible) for some of the variables in systems having many degrees of freedom.

The Geometric Method

Geometric solution techniques involve breaking the system down into a plane for each pair of links and geometrically solving for the variables in that plane. This method is usually less complex than the algebraic approach.

Geometric/Algebraic Solution for a 5-DOF Robot

A method combining the geometric approach with the algebraic method worked best for the 5-DOF robot of analyzed in this thesis. The geometric method was used to solve for angles θ_2 and θ_3, and the algebraic method was used to solve for angles θ_1, θ_4, and θ_5. These equations will be the basis for inverse kinematic positioning, and will be connected to the user interface controls explained in the next chapter.

Manipulator Positioning. Starting with the transformation relating the wrist frame of the robot to the base frame, as in Equation 3.10, results in Equation 3.12.

$$
{}^{B}_{W}T = {}^{5}_{0}T = \begin{bmatrix} c_1c_{234}c_5 + s_1s_5 & -c_1c_{234}s_5 + s_1c_5 & c_1s_{234} & c_1(c_{23}L_3 + c_2L_2) \\ s_1c_{234}c_5 - c_1s_5 & -s_1c_{234}s_5 - c_1c_5 & s_1s_{234} & s_1(c_{23}L_3 + c_2L_2) \\ s_{234}c_5 & -s_{234}s_5 & -c_{234} & L_1 + s_2L_2 + s_{23}L_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.12}
$$

(In order to make better use of space, $\cos(\theta_1)$ has been shortened to c_1 and $\cos(\theta_2 + \theta_3)$ is now c_{23}, etc.). This matrix is completely known since it is derived from the position variables entered by the user. For later reference, matrix of Equation 3.12 will described in terms of its elements as,

$$
{}^{B}_{W}T = {}^{0}_{5}T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.13}
$$

Since the elements of this matrix are not all independent, the user can enter data describing the position and orientation of ${}^{B}_{W}T$ by specifying three position and three rotation variables. One method of specifying these variables is to assign x, y, and z cartesian positions and roll, pitch, and yaw (γ, β, α) rotations based in the stationary coordinate system. ${}^{B}_{W}T$ can then be represented as,

$$
{}^{B}_{W}T = {}^{0}_{5}T = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma & x \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma & y \\ -s\beta & c\beta s\gamma & c\beta c\gamma & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}
$$

For the 5-DOF robot, only the cartesian position variables (x, y, and z) will be assigned by the user (γ, β, and α will be defined by the joint space variables θ_4 and θ_5).

Changing position variables can be performed either directly (by changing x, y, and z independently), or in terms of the manipulator coordinate system. These functions will be applied in terms of a *world* (stationary) mode and a *tool* (manipulator) mode as shown in Figure 3.4.

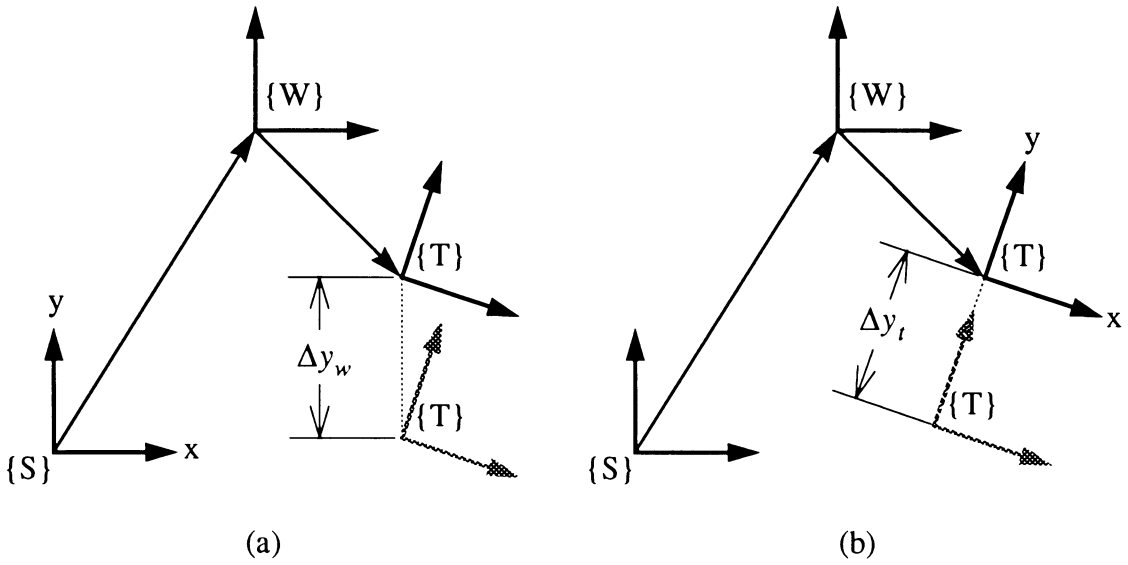


Figure 3.4 Positioning modes: (a) world mode, (b) tool mode

Deriving Joint Angles. From Equation 3.12 and 3.13, joint variable θ_1 can be obtained from the arctangent of elements r_{14} and r_{24},

$$\theta_1 = \text{atan}\left(\frac{r_{24}}{r_{14}}\right) \tag{3.15}$$

Note that a solution to this equation is not possible if $\theta_2 = -90°$ and $\theta_3 = 0°$, but due to the design of this robot, this condition will never arise.

Breaking down the robot into planes, as shown in Figures 3.5, gives the second joint variable, as shown in the following set of equations.

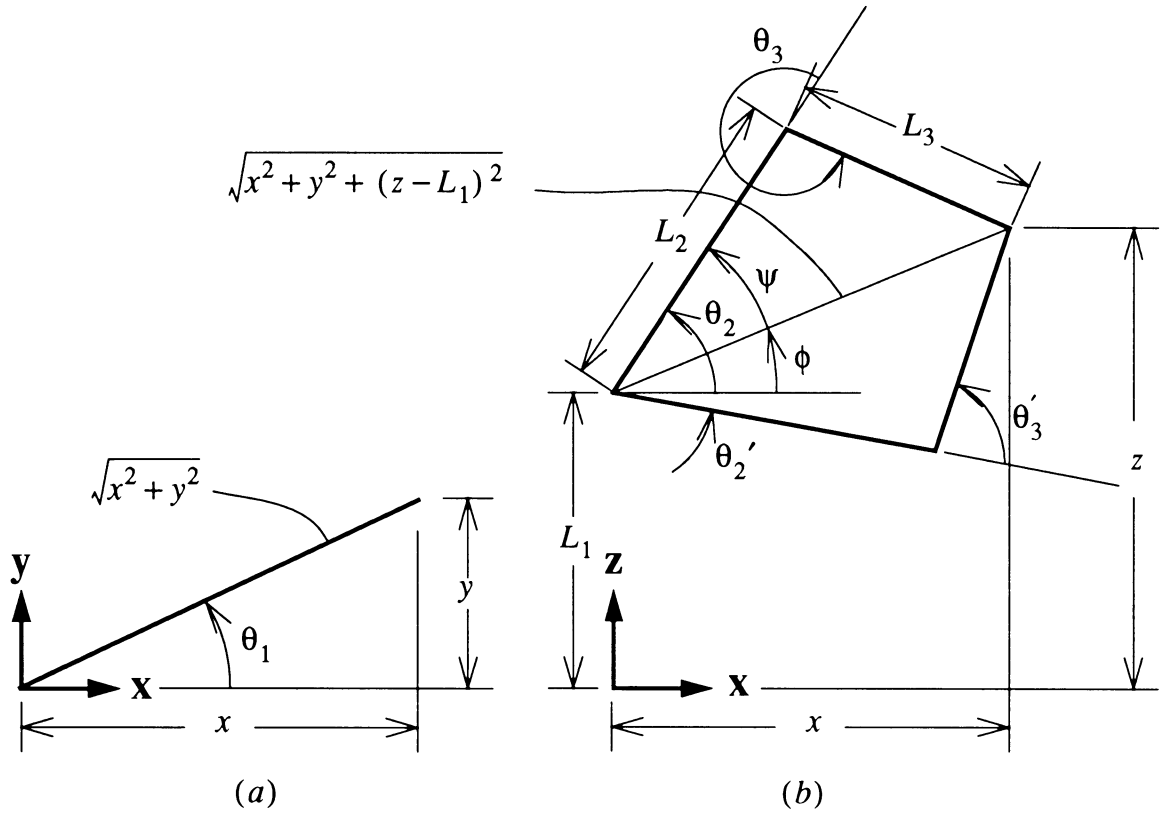


Figure 3.5 Geometry for calculating θ_2 and θ_3, (a) top view, (b) side view

$$\phi = \text{atan}\left(\frac{z-L_1}{\sqrt{x^2+y^2}}\right)$$

$$L_3^2 = x^2+y^2+(z-L_1)^2+L_2^2-2L_2\sqrt{x^2+y^2+(z-L_1)^2}\cos\psi$$

$$\psi = \text{acos}\left(\frac{x^2+y^2+(z-L_1)^2+L_2^2-L_3^2}{2L_2\sqrt{x^2+y^2+(z-L_1)^2}}\right)$$

$$\theta_2 = \phi \pm \psi \tag{3.16}$$

The third joint variable can also be calculated by using the geometric representation of Figure 3.5. Starting with the law of cosines,

$$x^2 + y^2 + (z - L_1)^2 = L_2 + L_3 - 2L_2L_3\cos(180° - \theta_3)$$

$$\cos(180° - \theta_3) = -\cos\theta_3$$

which results in,

$$\theta_3 = \pm\text{acos}\left(\frac{x^2 + y^2 + (z - L_1)^2 - L_2^2 - L_3^2}{2L_2L_3}\right) \tag{3.17}$$

Now that angles θ_1, θ_2, and θ_3 are known, the angle θ_4 can be found using elements from the matrix of Equations 3.12 and 3.13, which results in,

$$\theta_4 = \text{atan}\left(\frac{\sqrt{r_{13}^2 + r_{23}^2}}{-r_{33}}\right) - \theta_2 - \theta_3 \tag{3.18}$$

Finally, θ_5 can be also found by from Equations 3.12 and 3.13 through the following derivation, resulting in Equation 3.19.

$$a = \cos\theta_1\cos(\theta_2 + \theta_3 + \theta_4)$$

$$b = \sin\theta_1$$

$$c = \sin\theta_1\cos(\theta_2 + \theta_3 + \theta_4)$$

$$d = \cos\theta_1$$

$$\theta_5 = \text{atan}\left(\frac{(r_{11} + r_{21})(b - d) - (r_{12} + r_{22})(a + c)}{(r_{11} + r_{21})(a + c) + (r_{12} + r_{22})(b - d)}\right) \tag{3.19}$$

3.3 Path Generation

Path generation refers to the methods used to define the trajectory that the robot's manipulator will follow in three dimensional space with respect to time. Two different types of path generation will be discussed. The first involves specifying an interpolation scheme that operates directly on the joint variables, usually called joint space interpolation. This can be as simple as linear interpolation of joint angles from one control point to the next, or more complex, involving polynomial splines and multiple control points. Another type of path generation operates on a path defined in terms of cartesian variables, referred to as cartesian space interpolation. This type of path generation may involve moving the manipulator in a perfectly straight line or generating an arc or circle, and is used in applications like welding, painting, applying adhesives, and some types of assembly.

Of the two, the joint space method is less complicated to use and is less computationally intensive. Is used in situations when the cartesian path that the manipulator follows between control points is not critical. It is important to note that, in general, linear joint space paths will not result in linear cartesian movement of the manipulator. In addition, obtaining a kinematic solution using cartesian space methods is not always possible since a user may unknowingly define a linear manipulator path that passes outside the robot's workspace, or one that exceeds its acceleration limits. However, generating a kinematic solution between control points is always guaranteed when using joint space interpolation. Furthermore, it is possible to approximate cartesian path generation by using joint space interpolation and a large number of control points. These issues are important to consider when deciding which method to use. This thesis will focus on joint space schemes since they are relatively easy to implement on most robot controllers, and since they provide good control for many positioning operations.

Joint Space Methods

There are many types of joint space interpolation schemes, most are based on polynomial splines. In order to get continuous velocity and acceleration between segments, a polynomial of at least third degree is required. Splined cubic polynomials are the lowest order polynomial that satisfies this requirement.

Cubic Splines. A cubic spline is a set of cubic polynomials combined together, that can be made to have continuous first and second derivatives at the intermediate (or via) control points as shown in Figure 3.6. This figure shows one segment with three via points and four splined cubic polynomials (c1 through c4). The segment start and end points are defined to have zero velocity. The velocity of each via point is determined by its position in the segment. The complete path that the robot follows will be made out of many segments of this type.

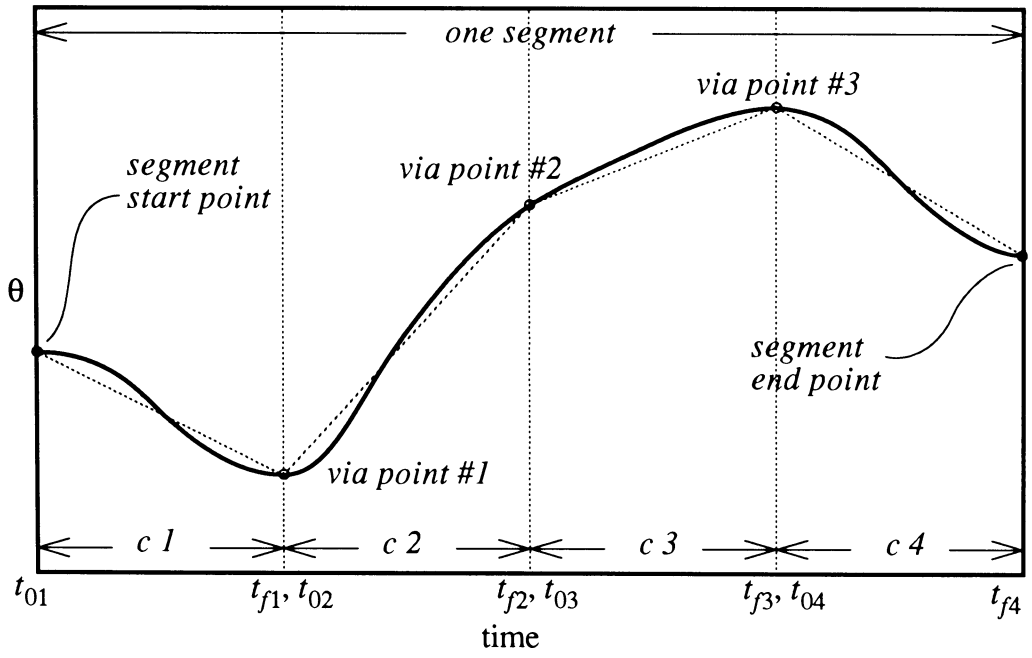


Figure 3.6 A splined segment with intermediate control points

The general form of a cubic polynomial is,

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{3.20}$$

which gives velocity and acceleration equations of,

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2 \tag{3.21}$$

$$\ddot{\theta}(t) = 2a_2 + 6a_3 t \tag{3.22}$$

In order to calculate values of a_0 through a_3 for each cubic in the segment, four constraint variables must be specified: initial and final position, and initial and final velocity, as listed in Equations 3.23 to 3.26,

$$\theta(0) = \theta_0 \tag{3.23}$$

$$\theta(t_f) = \theta_f \tag{3.24}$$

$$\dot{\theta}(0) = \dot{\theta}_0 \tag{3.25}$$

$$\dot{\theta}(t_f) = \dot{\theta}_f \tag{3.26}$$

where t ranges from 0 to t_f during each cubic of the segment. Using these conditions, the cubic coefficients are:

$$a_0 = \theta_0 \tag{3.27}$$

$$a_1 = \dot{\theta}_0 \tag{3.28}$$

$$a_2 = \frac{3}{t_f^2}(\theta_f - \theta_0) - \frac{1}{t_f}(2\dot{\theta}_0 + \dot{\theta}_f) \tag{3.29}$$

$$a_3 = -\frac{2}{t_f^3}(\theta_f - \theta_0) + \frac{1}{t_f^2}(\dot{\theta}_f - \dot{\theta}_0) \tag{3.30}$$

In order to make cubic splines easier to use, $\dot{\theta}_0$ and $\dot{\theta}_f$ can be assigned automatically. The simplest way of doing this is by assigning a velocity at a via point based on the average velocity between adjacent points, and assigning zero velocity at the start and end of the segment.

For most types of robots, the maximum angular velocity and acceleration will be specified instead of Δt for each cubic in the segment. When the maximum angular acceleration or deceleration, $\ddot{\theta}_m$, is specified for the starting or ending point of the cubic, Δt can be calculated as shown in Equations 3.31 and 3.32 for the acceleration and deceleration cases respectively,

$$\Delta t = \frac{-(2\dot{\theta}_0 + \dot{\theta}_f) \pm \sqrt{(2\dot{\theta}_0 + \dot{\theta}_f)^2 + 6\ddot{\theta}_m(\theta_f - \theta_0)}}{\ddot{\theta}_m} \tag{3.31}$$

$$\Delta t = \frac{(2\dot{\theta}_f + \dot{\theta}_0) \pm \sqrt{(2\dot{\theta}_f + \theta_0)^2 - 6\ddot{\theta}_m(\theta_f + \theta_0)}}{\theta_m} \tag{3.32}$$

If the value of Δt calculated by Equation 3.31 or 3.32 causes the maximum angular velocity, $\dot{\theta}_m$, to be exceeded, Δt can be recalculated as follows,

$$A = \theta_f - \theta_0$$

$$B = \dot{\theta}_m - \dot{\theta}_0$$

$$C = \dot{\theta}_f + \dot{\theta}_0$$

$$D = 2\dot{\theta}_0 + \dot{\theta}_f$$

$$\Delta t = \frac{2A\,(B+D) \pm \sqrt{(2A\,(B+D))^2 - 12A^2\,(CB + \frac{D^2}{3})}}{2\,(CB + \frac{D^2}{3})} \tag{3.33}$$

The sum of all individual Δt's will be used to give an estimate of the cycle time for a particular task.

More complex methods that assure continuous acceleration for cubic splines are presented by Fu et al. [13]. Other types of joint space interpolation including higher order polynomials and combinations of linear segments with parabolic blends, are presented by Craig [17]. Deciding on which type of interpolation to use depends on the capabilities of the control system used to drive the robot. For the 5-DOF robot used in this study, the cubic spline will be used to interpolate the joint variables.

4. SOFTWARE DEVELOPMENT

This chapter discusses the creation of off-line programming software based on the kinematic positioning and path generation principles developed in Chapter 3. Included is: 1) a summary of requirements for viable off-line programming in a graphical environment, 2) a basic review of computer graphics concepts, 3) a description of the user interface controls, and 4) an explanation of the major subsystems included in the robot simulator. Specific information on the operation of the program is presented in the preliminary users manual in Appendix A.

4.1 Graphical Off-line Programming Requirements

In order to create a basic graphical off-line programming environment, the following concepts and functions need to be addressed.

- Objects of the workcell must be represented as solid models.
- Viewing position cameras must be defined.
- The user must be given the ability to interactively place objects anywhere within the robot's workcell.
- Dynamic control of the forward and inverse kinematic positioning functions must be implemented.
- A method of assigning and saving path control points must be defined.
- A method of associating positions of objects with the position of the robot's manipulator at any location along its path must be defined.

- A method for translation of graphical simulation data to device control code must be implemented.
- File input and output functions must be included.

Each of these topics will be discussed in this chapter.

4.2 Basic Computer Graphics Concepts

Object Modeling

In order to visualize the objects that make up the workcell, data describing the 3-D geometry must be created and imported into the computer. This data can be represented either as polygons or as curves and surfaces (NURBS for example). Polygonal data is much less complex and can be rendered at higher update rates than curve and surface data. For this reason, the polygon based data structure will be used to represent the object models usually referred to as "solid" models. There are many commercial packages available for creating solid models. The solid model of the robot tested in this thesis, shown in Figure 4.1, was created using the I-DEAS solid modeling package. Although solid modelers may have different methods of representing data internally, most have the ability to write out this data as a list of 3-D points and a list describing how these points are connected to form the polygons that make up each object. The method used to import the polygon information into the robot simulator is based on the BYU format, and was chosen due to its relatively compact data representation [19].

Cameras

Once the polygonal data describing the object models has been imported, viewing parameters must be set to display this data. In order to create an interactive environment for working with solid models, the user must be given the ability to dynamically change the viewing position that is displayed on the screen. A viewing position, or camera, is generally described by a look-at point (the point in 3-D space at which the camera is

pointing) and a look-from point (the location where the camera itself is positioned). These points will be internally represented as a 4×4 viewing matrix. Another 4×4 matrix called the perspective matrix, which defines the field of view, will also be associated with each camera. Recall from Chapter 3 that all location of the links of the robot were also described by 4×4 matrices. These matrices will be multiplied by the viewing matrix and the perspective matrix in order to obtain the graphical image of the object model projected onto the computers two dimensional screen. In addition, camera positions can be interpolated from one position to another. Cameras can also be constrained to move with, or follow, the robot's manipulator. This feature can give the user the ability to follow a particular object throughout a manufacturing or assembly process.

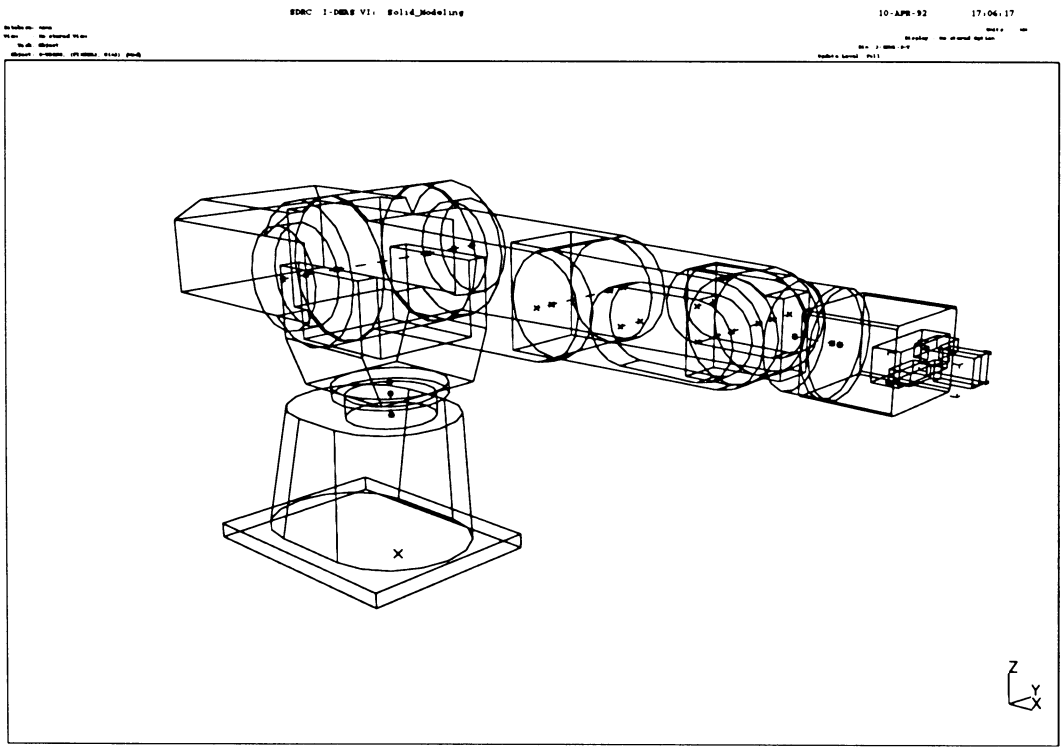


Figure 4.1 Solid model of the Mitsubishi RV-M1 robot

Rendering, Lighting, and Materials

The final step in visualizing the solid models is to determine how they will be displayed. The rendering mode refers to the way an object is drawn, or shaded. The simplest method of drawing an object is to display it in wireframe, which consists of drawing lines connecting the points that make up the object. A more realistic method of shading an object involves defining materials (or set of color properties) and lighting sources. The material assigned to an object defines how light is reflected from its surface. The light source defines color and the direction of the light. This information can then be used to shade, or fill in, the polygons that make up the object model. Two types of shading are used by the robot simulator: flat shading and smooth (Gouraud) shading. A complete description of rendering is beyond the scope of this thesis. A detailed review can be found in any text dealing specifically with computer graphics, for example, Foley et al. [20] or the SGI Graphics Library Programming Guide [21].

Having defined the basic concepts of representing and viewing object models, the operational structure of the off-line programming interface for the robot simulator, RS, will be the focus of the remainder of this chapter.

4.3 Interface Controls

A primary issue when creating a graphical interface is deciding how to dynamically control an input variable. Dynamic control of a variable (not to be confused with the dynamics of motion generation) is the ability to continuously calculate the function dependent on that variable and visualizing the change as it happens, as opposed to making a change and then waiting for the update to occur after the request is entered. This is analogous to turning up the volume on a radio and hearing the sound level increase as you turn the knob, instead of finding out how much you changed the volume after you let go of the knob. This type of feedback leads to more efficient control and convergence to a desired solution. One of the most useful dynamic controls in computer graphics is a mov-

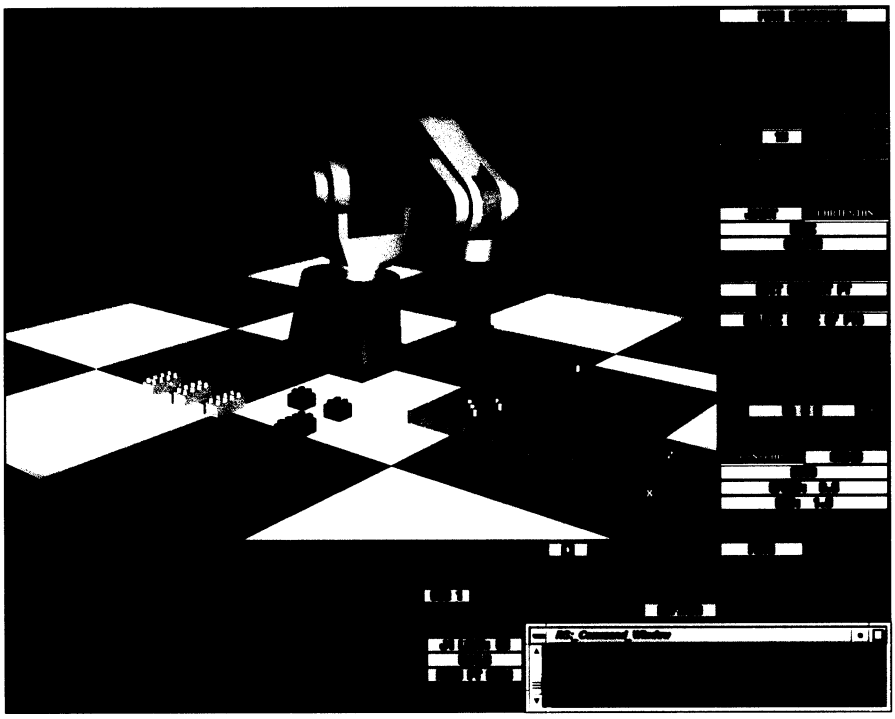
ing input device like the slider of Figure 4.1. Input devices like the slider are controlled by positioning to cursor on the slider and holding down a mouse (or keyboard) button while dragging the cursor across the screen. Many of the parameters in the robot simulator are changed by controls of this type.

Figure 4.2 A slider.

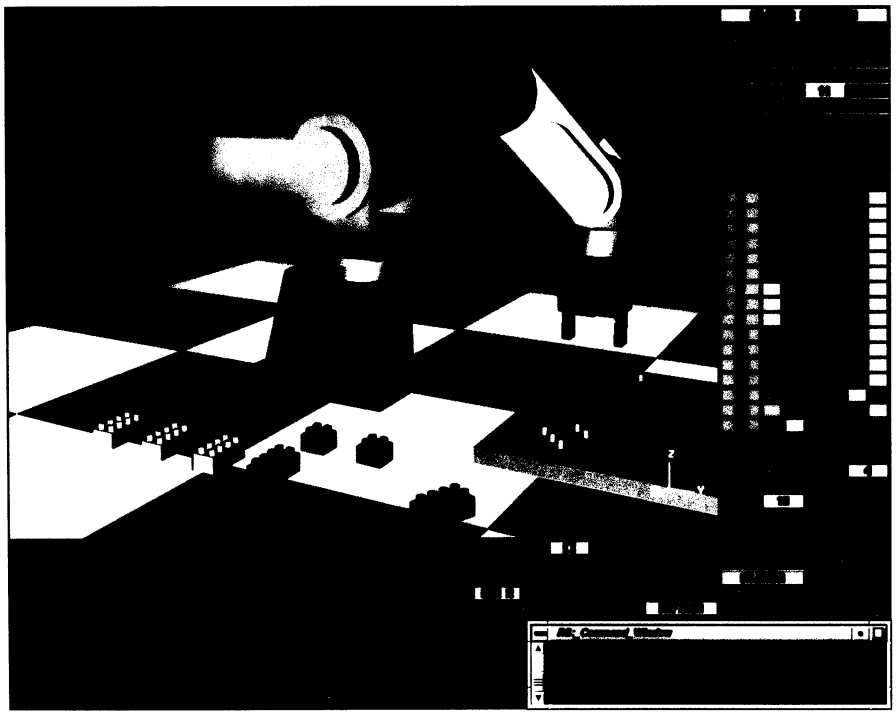
The basic layout of the simulator screen is divided into four main areas. The main portion of the screen is taken up by the viewing window, which displays the solid model geometry, as shown in Figure 4.2. The lower left of the screen is the area where the slider controls for each subsystem will be displayed. Near the center of the screen (just under the lower right of the viewing window) is a set of buttons and a slider for controlling cameras, rendering modes, and animating the simulation (using the VCR button arrangement). The lower right of the screen is taken up by a command window which displays commands as they are executed. Just above the command window is the main menu which gives access to each of the subsystems in the simulator. The area above the main menu is used to display the controls of each subsystem menu. Keyboard function keys are used to define and control the viewing parameters associated with each camera.

4.4 Major Subsystems

The basic functions listed in the beginning of this chapter are divided into several subsystems which are accessed from the main menu. The design and functions of these subsystems are described in the following sections.



(a)



(b)

Figure 4.3 Graphical interface: (a) path subsystem, (b) objects subsystem

Path Subsystem

The path subsystem, which is displayed in Figure 4.2a, controls forward and inverse kinematic positioning as well as the selection of path control points and the computation of splined segments. This is the primary subsystem used to create a simulation.

Kinematics. The set of sliders in the lower left corner of the screen are used to control manipulator positioning. The first three sliders are used to control x, y, and z cartesian space positioning of the manipulator, and are tied directly to the inverse kinematics algorithm. The next six control each of the robot's joints independently (the sixth one is not used for the RV-M1, since it is a 5-DOF robot). These sliders are tied to the forward kinematics algorithm. (Actually, the forward kinematics algorithm is also called by an update to the cartesian sliders; the inverse kinematics code calculates the joint angles, which are then used by the in the forward kinematics calculations to find the 4×4 transformation matrix for each link.) The last slider in the set controls whether the gripper (or tool) is open or closed. If the user changes a variable to a position that is out of the robot's workspace, the program will reset the variable (and move the slider) back to the last correct value. The options menu to the left of these sliders controls the inverse kinematics positioning mode. Three kinematic modes are available: world and tool coordinate positioning, and a wrist roll compensation mode (which corrects for wrist rotation when the z-axis of the tool aligns with the stationary z-axis).

Path Generation. The buttons on the right of the screen control the path generation functions. Currently, only the joint space interpolation mode is operative, and the only joint space interpolation available is the cubic spline with continuous velocity at the via points. The joint positions along the path (i.e., starting, ending, and via points) are stored within the program in two lists, called point and path. The point list contains the values associated with each control points. The path list is calculated after all the point values have been set and contains the joint variables associated with each animation frame in the simulation. These values are determined by the cubic coefficients derived in Chapter 3

(see Equations 3.27 through 3.30). The method of applying these equations uses a value of Δt determined by a maximum speed and acceleration set by the user and the actual joint velocity specifications supplied by the robot manufacturer (see Appendix B). The value of Δt is set for each segment of the path, not for each individual via point (for example, if a segment contains three via points, the time per cubic will be $\frac{\Delta t}{4}$ for each of the four cubics in that segment).

The method of storing joint variables for the path requires the 4×4 transformation matrices to be calculated while the animated simulation is being displayed. This method is not as computationally efficient as storing the matrices, since the same set of matrices must be recalculated each time the simulation is played. The major advantage of storing only the joint variables is reduced memory usage, since only one variable needs to be stored instead of the 16 elements that make up each transformation matrix. An option will be provided later to give the user a choice of storing the 4×4 transformation matrices for increased speed when memory is not in limited supply.

Objects. In order to allow the robot to manipulate objects during a simulation, objects may be constrained to the tool position at any start point of a segment. This constraint is defined when the object(s) is selected from the current objects menu, by associating the current object position with the current tool position. When an object is first selected, a transformation matrix is created to relate the position of the object to the tool, ${}^{T}_{O}T$, as shown by Equation 4.1. The ${}^{T}_{O}T$ matrix can then be used to calculate the position of the object relative to the stationary reference frame at each update position of the tool, as shown by Equation 4.2.

$$ {}^{T}_{O}T = {}^{S}_{T}T^{-1} \; {}^{S}_{O}T \tag{4.1} $$

$$ {}^{S}_{O}T' = {}^{S}_{T}T' \; {}^{T}_{O}T \tag{4.2} $$

Where ${}^{S}_{T}T$ and ${}^{S}_{O}T$ are the initial object pickup positions, and ${}^{S}_{T}T'$ and ${}^{S}_{O}T'$ are these positions at a later time. A set of these matrices will be defined for each object that is associated with the current tool position. Diagrams of these frame transformations are shown in Figure 4.3.

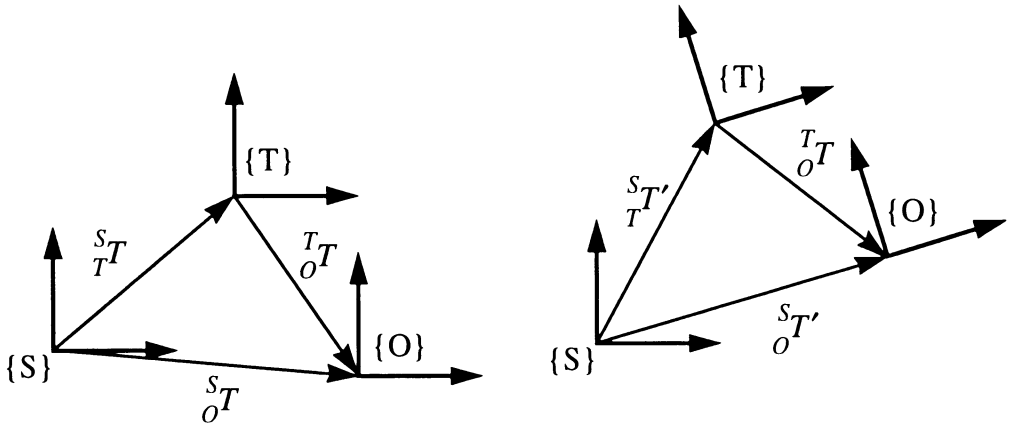


Figure 4.4 Object transformation diagrams

Objects Subsystem

The objects subsystem, shown in Figure 4.2b, handles positioning, material assignment, and rendering mode for all objects in the workcell (except the robot). The sliders for controlling position are located in the lower left. These six variables (x, y, z, roll, pitch, and yaw) are used to set or change the initial transformation matrix of an object relative to the stationary reference frame. The material associated with each object is chosen from the pallet of available materials (which is defined in the materials subsystem). Rendering modes for each object are also defined within this subsystem. These rendering definitions are activated when the "DEFINED" option is selected from the main rendering mode menu. The rendering modes available are, bounding box (which is a simple block representation of the object), wireframe, flat shaded, and smooth (Gouraud) shading.

Robot Subsystem

The robot subsystem operates in much the same way as the object subsystem for defining positioning, material assignment, and rendering mode for each robot. This subsystem will eventually allow the user to load multiple or different types of robots (currently only the RV-M1 model is functional). This feature will allow the user to create graphical position data for one type of robot and then use the same data to evaluate of different types of robots performing the same specified task.

Material and Lighting Subsystems

The combination of the material and lighting parameters define the appearance of all components of the workcell. The material subsystem allows the user to define a pallet of materials that can be applied to objects and robot solid models. This subsystem controls the ambient, diffuse, specular, and emitted properties of light falling on any solid model assigned to a material. The lighting subsystem allows definition of up to eight light sources. The lighting parameters that can be modified are the light source positions and color definitions.

File Input/Output Subsystem

Input and Output of status, points, and device control files are handled by the file input/output subsystem. The status file contains information about the workcell environment (lighting, camera positions), and about objects that make up the workcell (robots, fixturing). The points file contains the joint variables in the points list, and information about speed and acceleration settings. The device control file contains the operating commands written in the language of the robot's controller. Sample files that can be read or written by the simulator are given in Appendix A.

ınimation Subsystem

This subsystem allows the user to send frames from an animated simulation to a file oı ɔ a frame storage device (for example, an Abekas A60). These frames can then be playec t a rate of thirty frames per second to create a true "real time" animation a robot simula- ion.

5. RESULTS

The off-line programming software developed over the course of the research was evaluated at several stages during its development. The initial evaluations dealt with the function of the interface and the efficiency of the inverse kinematics algorithm. The second phase of testing looked at the efficiency of the off-line programming aspects of the software from a user point of view.

5.1 Using the Interface

Connecting the position and path generation equations with the graphical interface presented many challenges, some concerning the methods used in obtaining the position solutions, and others pertaining to the functions available for creating the path that the robot will follow in three dimensional space.

Inverse Kinematics Testing

As described earlier, the method selected to obtain inverse kinematics solutions was the closed form approach. Testing was also preformed using the numerical approach, but it did not lend itself the environment required for off-line programming. A brief description of this testing may provide useful insight as to the reasons why the closed form solution technique is the preferred solution method.

Interactive inverse kinematic solutions using numerical methods can be generated in many ways, but all rely on making an initial guess at the joint solutions for a particular

cartesian goal, and then iterating by comparing these solutions to those generated by the forward kinematics equations. Several methods of applying numerical techniques to computer graphics also exist. One involves updating the solution set once for every cycle through the graphics update loop. This type algorithm will continue iterating as long as the user continues to hold down the mouse button after a position change has been requested. This allows the user to decide when the solution is good enough to stop the iteration process. Application of this technique proved to be to slow for interactive use. Updating the position in this manner is very inefficient since the graphics must be updated for each iteration. A more efficient method involves setting a tolerance on an acceptable error for each joint and then updating the graphics only after an adequate solution has been reached. This method may become more usable for dynamic interaction as computers become faster, but even an extremely fast computer will have trouble converging on a solution when the desired position is far away from the current position. Some of these convergence problems can be solved by giving the numerical algorithm a set of heuristic instructions to follow for getting itself out of trouble. For example, if the desired solution is to far away from the current position, a set of intermediate positions could be defined so that a final solution could be arrived at by first solving the intermediate steps. Solving a set of intermediate positions for each joint obviously slows down the calculation even more.

Closed form solutions do not have these types of convergence problems. Application of the closed form solution for the 5-DOF robot presented in Chapter 3 resulted in fast update rates that were efficient enough to allow dynamic interaction. For these reasons, the closed form method was developed for use in the robot simulator.

Generating Paths

The only method currently available in RS for generating a path uses a continuous velocity cubic spline. Other joint space methods (including different versions of the cubic

spline), and cartesian space methods are planned for future development. The addition of these types of path generation functions would give the simulator better compatibility with many different types of commercially available robots. Fortunately, the cubic spline closely approximates the joint space control of the Mitsubishi RV-M1, on which all testing was performed.

5.2 Off-line Programming Testing

Testing of the software consisted of a variety of simple tasks involving graphically simulating a Mitsubishi RV-M1 robot positioning objects in the workcell, then translating the graphical commands to the robot's device control code, and transferring the code to the robot's control computer. Testing of this code was then performed on the actual robot. A diagram showing the test equipment and transfer of information is shown in Figure 5.1.

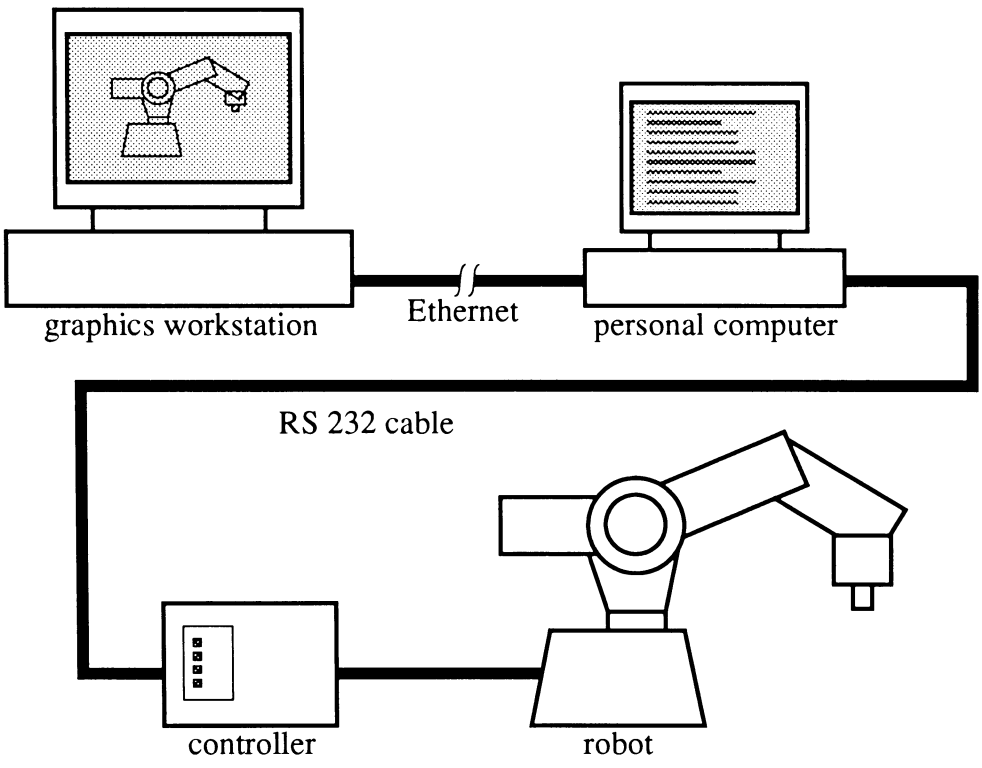


Figure 5.1 Test equipment

The reason the PC is included in the loop is because it is dedicated to controlling only the devices of a particular workcell. Using an expensive graphics workstation to do this task would be an inefficient use of resources. But recently, with the drastic reduction in the price of entry level graphical workstations, the PC may soon be replaced by a dedicated workstation that can perform graphical off-line programming as well as workcell device control.

Tasks Performed

The actual tasks used for testing initially consisted of moving and stacking wooden blocks within the workcell, then proceeded to palletizing objects, (Figure 5.2), and then assembling structures with interlocking "Lego" blocks (Figure 5.3). The object palletizing and assembly tasks were chosen in order to test general positioning and data transformation accuracy, and because they are relatively common types of tasks for the RV-M1 robot.

Time Savings

Results showed that programming these types of tasks could be performed off-line in approximately 25% of the time required to create the same program using on-line programming techniques (i.e., using a teach pendant). Much of this time savings came from the ability to easily recover from programming mistakes. Graphical simulation can easily point out positioning errors, that can be quickly modified. In addition, when graphically programming a robot off-line, the user can accidently force the manipulator through the floor or into other objects without the fear of damaging the robot. Whereas in the actual workcell, these types of mistakes may cause serious damage to the robot (and its surrounding) or at least require the user return the robot to its home position to reset joint sensors.

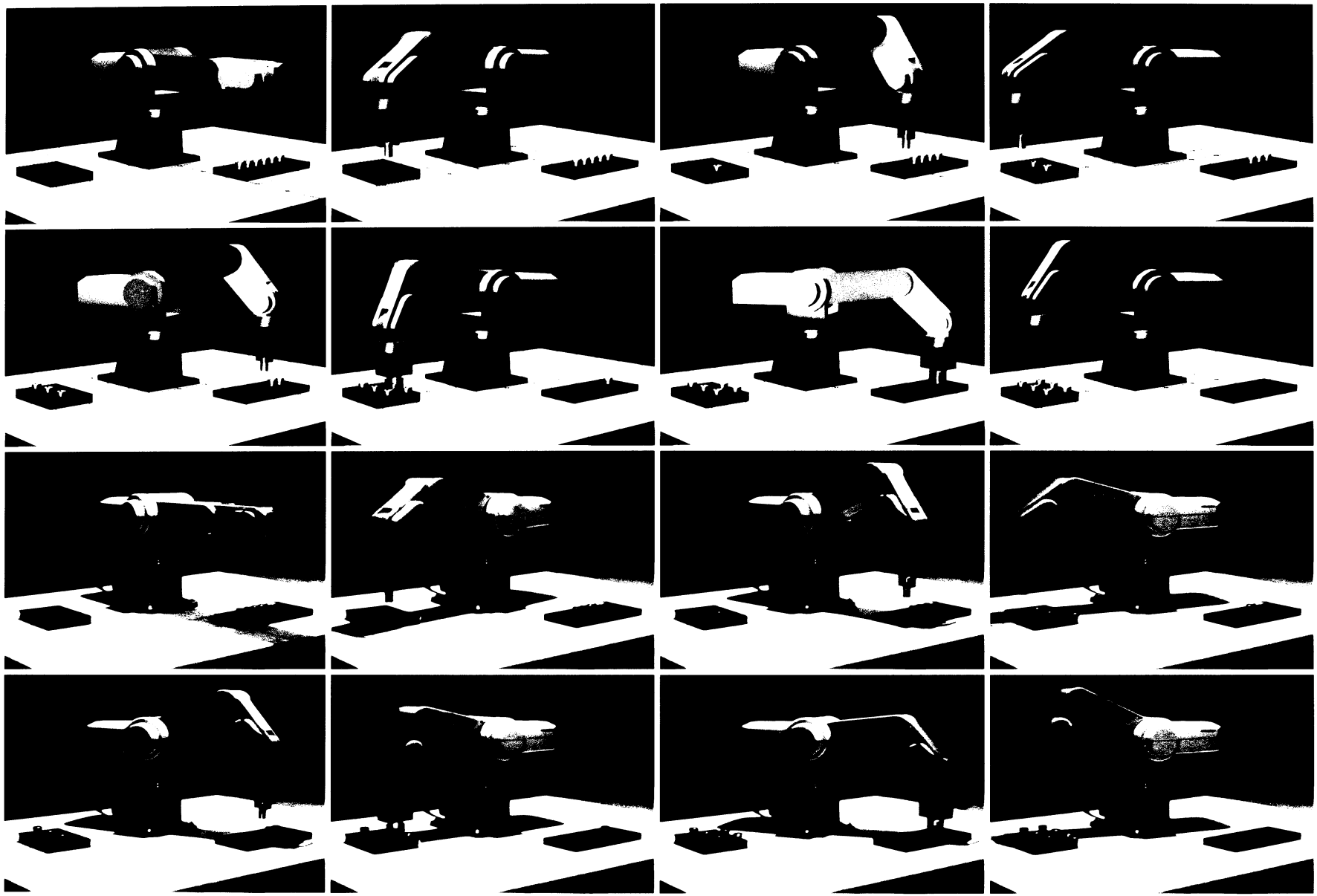


Figure 5.2 Comparison of simulated and actual component palletizing

Figure 5.3 Comparison of simulated and actual Lego block assembly

Another source of time savings comes from the ability to quickly position the robot's manipulator at goal points, and to easily modify these points. Special positioning functions that are not available on the teach pendant can be added to the simulator to reduce programming time even further. An example of one type of special position function that was added to the RV-M1 position control interface was a joint lock which could be used to disable wrist rotation.

Cycle Time Estimations

Cycle times predicted by the simulations were underestimated by about 20%. This error was due to the lack of data on the acceleration characteristics of the robot's servo motors. Knowledge of the actual acceleration parameters and further testing should reduce this error.

Position Accuracy

After the device control code was transferred to the robot and tested, it was sometimes necessary to do some minor on-line, or touch up, positioning in order to get the robot to perform the desired task. The touch up programming was usually limited to a few millimeters and was attributed mainly to inaccurate positioning (or inaccurate measurement of positions) of the objects in the actual workcell.

Touch up programming was required in the palletizing case, since the holes in the fixturing were somewhat irregular and did not precisely match the solid model. The Lego assembly, on the other hand, worked correctly without any on-line touch up. This can be attributed to the consistent shape of the Lego blocks and accurately constructed solid models.

Calibration

Another source of error was due to individual robot calibration variations and the lack of a signature model in the robot simulator to correct for it. The addition of signature models will allow the user to modify the ideal goal position data set generated by the robot simulator to an equivalent set of positions designated for the calibration of a specific robot. This feature would allow the user to create one program that could be transferred to many robots (of the same type) without making on-line program modifications to compensate for the differences of individual robots. This type of data transformation would take place just before the device control code was written out for a specific robot.

A diagram of one type of signature model is shown in Figure 5.4. This model is used to alter graphical position values before they are written into the device control code file. The graphical position changes in this model are based on an error map of the robot's workspace. Creating the map involves moving the robot's manipulator through a set of positions defined in a device control file and then comparing the positions to those measured by an external device.

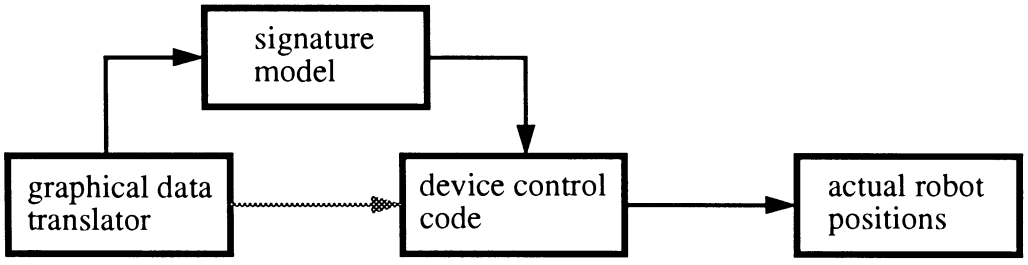


Figure 5.4 Signature model

In the absence of a signature model, fixture positions can sometimes be modified or purposely misaligned in order to compensate for minor calibration errors, but this is only a temporary solution.

Manipulator Force and Collisions

Another problem encountered was that of controlling the pressure applied by the robot's gripper during the simulation. It is very difficulty to judge how much gripper force is needed to lift an object without knowing information about the contact surfaces and the weight of the object. This was another area that occasionally needed on-line fine tuning. A more accurate prediction of gripper force could be obtained if gravity and specific object parameters (like mass and friction coefficients) were taken into account.

Collisions between the robot and objects in the workcell are sometimes difficult to predict without some type of collision detection algorithm. In a graphical environment, a problem can be addressed by having the user zoom in close enough to the area of interest to visually detect most types of interference. However, in the actual workcell, collisions may still occur. Collisions may go undetected when the cameras are not positioned correctly to view a particular operation, or when the simulation update rate is too large and skips over a trouble spot. Some of these types of collisions can be solved by varying programming techniques. For example, while assembling Legos, most types of collisions were avoided by approaching goal points from different orientations and avoiding situations where linear movement was required over a large distance (since control of the RV-M1 is limited to joint space movement). Another way to reduce collisions is by using some type of collision detection algorithm, which could be set up to give a warning when two objects are within a specified range of each other. This feature would notify the user of potential problems that may arise during the actual robot operation that may have been overlooked in the simulation.

User Comments

Some of the comments and suggested improvements expressed by people who have used RS for off-line programming are expressed below:

- A quick method of adjusting the fine-tuning (or resolution) of the sliders is needed. One possibility would be to have one of the mouse buttons control large position changes and use another button for fine positioning.
- The user should be allowed to enter shortcut command sequences from the command window.
- Menu layout is somewhat cluttered.
- Conveyors and other independently moving devices should be added to the simulator.
- A workspace map is needed to help with initial positioning of objects within the work-cell.
- Better device control code translation is needed (i.e., a complete signature model for each robot).
- A help menu should be added.
- The user should be allowed to measure positions off the screen by selecting points with the mouse.

6. CONCLUSIONS AND RECOMMENDATIONS

This thesis presents the initial stages of development of an interactive graphical robot simulator with off-line programming capabilities.

Forward and inverse kinematic equations were developed to control positioning of a five degree of freedom robot. Cubic splines were applied to interpolate joint variables for path generation.

A graphical interface was developed to allow dynamic control over kinematic positioning and path generation equations. This interface allows the user to graphically simulate a workcell and then translate simulation data to device control code for use on the actual robot.

Testing was performed on a Mitsubishi RV-M1 robot to verify the software functionality, and to suggest possible improvements to the interface as well as overall capabilities.

6.1 Future Work

This is a partial list of improvements and additional functions planned for future development of the robot simulator.

- Additional joint space and cartesian space path generation capabilities should be developed.
- Additional testing should be performed in a manufacturing environment.

- Additional robot models should be added to create a library of available models to choose from. Each model must include a solid model of the robot, forward and inverse kinematic equations, and a device control code translator.
- Robot signature models should be added to compensate for differences among individual robots, thereby reducing on-line touch up programming.
- Collision detection algorithms can be added in order to notify the user of interference as well as to let the robot know if it has something in its grasp.
- Multi-body dynamics post processing of kinematically defined paths would allow more accurate simulation of robots in which mass and inertia properties play important roles.
- The program could be modified to allow feedback from sensors to be simulated.
- The ability to directly create programs with proprietary languages, like VAL II and AML, could be incorporated.
- An independent language could be created to control all programmable devices in the workcell (like conveyers), as well as dealing with logical programming (like error recovery). Ideally, this type of programming could be done with graphical icons.
- An independent language would also allow users to attach their own control algorithms and task level or artificial intelligence programs.

6.2 Closing Remarks on Computer Graphics

It has become more apparent to me over the course of my research that the area of computer graphics is beginning to play a more important role in engineering. The field of computer graphics was once considered only a specialized area of computer science, but is now becoming a tool that can be used to change the way people in many different fields think about finding solutions. Since visualizing the solutions that computers generate is such a powerful tool for finding solutions quickly, it has become increasingly important to design software with computer graphics in mind. The ability to dynamically change a variable (like grabbing a slider with the cursor) also has tremendous potential in the teaching of engineering concepts.

BIBLIOGRAPHY

[1] Ackerman, W. B. and Dennis, J. B. "VAL - A Value-Oriented Algorithmic Language, Preliminary Reference Manual." MIT Laboratory for Computer Science Report No. MIT/LCS/TR-218. MIT Press, 1979.

[2] Woodcock, R. "Robot Basic Integrates Functions to Facilitate Off-line Programming." *Electronics*, v 57, n 14, pp 124-127, July 1984.

[3] Derby, S. "GRASP From Computer Aided Robot Design to Off-line Programming." *Robotics Age*, v 5, n 2, pp 11-13, February 1984.

[4] Patt, T. J. and Derby, S. "The Off-line Programming and Simulation of a Dual Manipulator Assembly Robot." *Computers in Engineering*, Conference Proceedings published by ASME, pp 365-375, 1988.

[5] Mirolo, C. and Pagello, E. "A Solid Modeling System for Robot Action Planning." *Computer Graphics and Applications*, pp 55-69, January 1989.

[6] Phillips, C. B., Zhao, J., and Badler, N. I. "Interactive Real-time Articulated Figure Manipulation Using Multiple Kinematic Constraints." Computer Graphics, SIGGRAPH Conference Proceedings, ACM Press, v 24, n 4, pp 245-250, July 1990.

[7] Webster, J. "Simulating the Factory Floor." *Computer Graphics World*, v 13, n 6, pp 54-59, June 1990.

[8] Mogal, J. S. "IGRIP - A Graphics Simulation Program for Workcell Layout and Off-Line Programming." *Robots 10 Conference proceedings*, published by Robots International of the SME, pp 65-77, 1986.

[9] Chan, S. F., Webster, R. H., and Case, K. "Robot Simulation and Off-Line Programming." Computer Aided Engineering Journal, Loughborough Univ. of Technology, v 5, n 4, pp 157-162, August 1988.

[10] Denavit, J. and Hartenberg, R. S. "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices." *J. Applied Mechanics*, pp 215-221, 1955.

[11] Ho, C. Y. and Sriwattanathamma, J. *Robot Kinematics: Symbolic Automation and Numerical Synthesis*. Ablex Publishing Corporation, Norwood, NJ, 1990.

[12] Stone, H. W. *Kinematic Modeling, Identification, and Control of Robotic Manipulators*. Kluwer Academic Publishers, Boston, 1987.

[13] Fu, K. S., Gonzalez, R. C., and Lee, C. S. G. *Robotics: Control, Sensing, Vision, and Intelligence*. McGraw-Hill, New York, 1987.

[14] Nikravesh, P. E. *Computer-Aided Analysis of Mechanical Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[15] Isaacs, P. and Cohen, M. F. "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics." Computer Graphics, SIGGRAPH Conference Proceedings, ACM Press, v 21, n 4, pp 215-224, July 1987.

[16] Davis, J. L. "Analytical and experimental dynamic characterization of the Mitsubishi RV-M1 robot." M.S. Thesis, Iowa State University, Ames, IA, 1991.

[17] Craig, J. J. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, Reading, MA, 1986.

[18] Latombe, J. C. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA 1991.

[19] Christiansen, H. N. *Movie.BYU Training Text.* Community Press, Provo, UT, 1986.

[20] Foley, J. D., vanDam, A., Feiner, S., and Hughes, J. D. *Computer Graphics Princi ples and Practice*. 2nd ed. Addison Wesley, Menlo Park, CA, 1990.

[21] Silicon Graphics Inc. *Graphics Library Programming Guide*. Mountain View, CA 1991.

APPENDIX A

USERS MANUAL AND SAMPLE FILES

- Preliminary RS Users Manual
- RS input status file (needed to load initial data into the robot simulator)
- RS output points file (list of path points)
- RS output device control code (in BASIC for the RV-M1)

Preliminary RS Users Manual

This manual is intended to give the first time user the basic information necessary to use RS to create a robotic simulation and to write out a device control code file.

Starting RS

RS can be started in two way, by typing *rs*, or by typing *rs* and the name of a status file, for example, *rs blocks.olp*. The status file contains information about the workcell environment (lighting, camera positions) and about objects that make up the workcell (robots, fixturing), see the sample status file at the end of this manual. Note in the sample status file under the OBJECT GEOMETRY section, that geometry files must be read into RS using the BYU format. An updated status file can be written out from within RS to save any changes that may have been made during a session.

Once RS has started up, notice that the display is divided into four main areas, as shown in Figure A1. The majority of the screen is taken up by the viewing window which displays the workcell as seen through the current camera. The command window in the lower right of the screen displays commands that have been executed. Just above the command window is the main menu that selects which subsystem is currently active, this menu is visible from any of the subsystems. Above the main menu is the section of the screen used to display the menus of the current subsystem. To the left of the main menu, just below the viewing window, is a bank of buttons and a slider that are used to control animation, cameras, and global rendering mode. The lower left of the screen is taken up by sliders which control positioning of the robot manipulator, this section will change form depending on which subsystem is currently active.

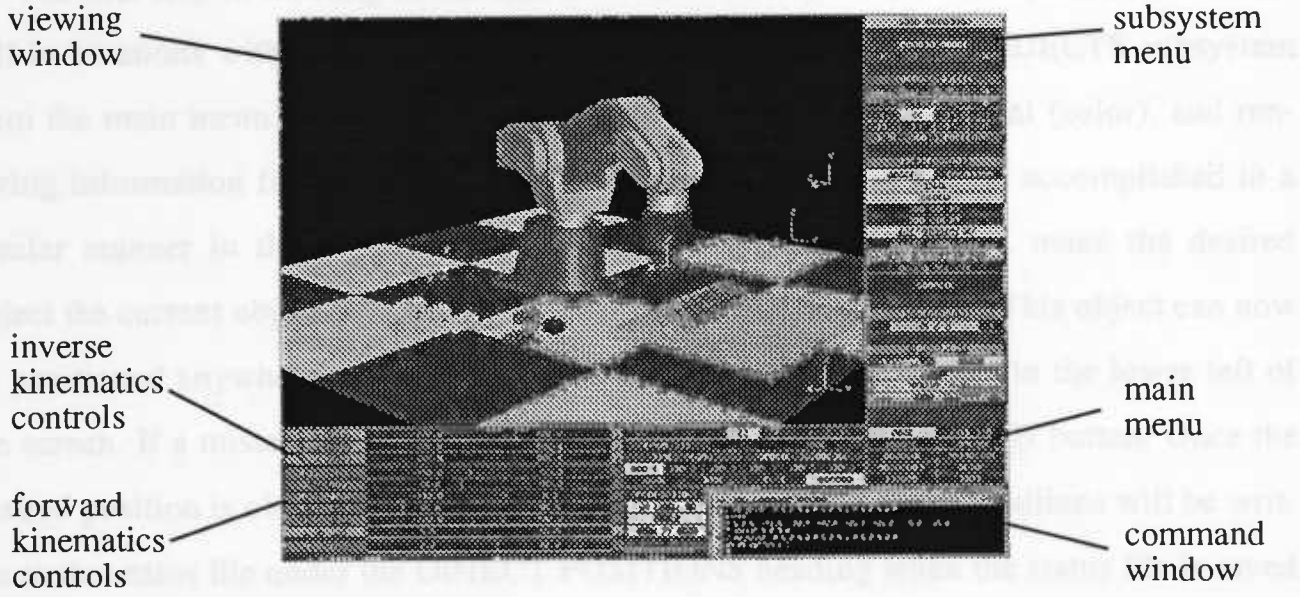


Figure A1 RS screen layout

The rest of this manual will describe the basic steps necessary to create a device control file off-line, which includes; positioning objects in the workcell, creating a path, and writing out a device control file. The robot used in the example will be a Mitsubishi RV-M1 (other robot models will be added later).

Positioning Cameras

The camera positions can be set by using the function keys: F1 translates, F2 zooms in and out, F3 rotates, F4 changes viewing distance, and F5 controls the clipping planes. Six camera positions can currently be defined, and will be saved in the status file when it is written out. Additional camera functions (splined camera movements, for example) will be added to the CAMERAS subsystem later.

Positioning Objects

The first step in creating an off-line simulation is to position the objects of the workcell in locations within the workspace of the robot. First pick the OBJECTS subsystem from the main menu. This subsystem controls object position, material (color), and rendering information for all objects other than the robot (which can be accomplished in a similar manner in the ROBOTS subsystem). To position an object, make the desired object the current object by selecting it from the current object menu. This object can now be positioned anywhere in the workcell by moving the object sliders in the lower left of the screen. If a mistake is made, click the RESET OBJECT POSITION button. Once the desired position is obtained, click SET OBJECT POSITION. These positions will be written to the status file under the OBJECT POSITIONS heading when the status file is saved (in the FILE I/O subsystem).

Creating A Path

Now that all the positions of the objects are set, switch to the PATH subsystem. This subsystem controls selection of control points used to create a path, as well as calculating the updated positions along the path itself. By default, the system is in the joint space mode with a cubic splined path selected (which is currently the only type available in RS). A path will be created by selecting several segments in which the robot will come to a complete stop at the end points. Intermediate "via" points can be placed within a segment at locations in space through which the robot must pass through, but not stop. (Note, don't use via points when simulating the RV-M1, since its device controller can't interpret them.) When staring a path the current point display will read 0:0 0, the first number indicates the current point, the second indicates the current segment, and the last refers to the current via point in the segment.

To create path positions, use the sliders in the lower left of the screen to position the manipulator. The three sliders at the top of this section (labeled x, y, z) control cartesian

space movement. Sliders J1 through J6 control the joint variables independently (J6 is not operative for the RV-M1, since it is a 5-DOF robot). The status of the tool (opened or closed) is controlled by the last slider. Trying to move the robot's manipulator out of the workspace in either the cartesian mode or joint mode will cause an error message to be displayed in the command window.

Once the desired position of the manipulator has been reached, click on the ADD S/E POINT button, then move the manipulator to the next position. The starting and ending points of each segment must be made with S/E (start, end) points. Intermediate via points can be added within a segment with the ADD VIA POINT button. To pick up one or more of the objects within the workcell, select the object or objects from the CURRENT OBJECTS menu just before clicking the ADD S/E POINT button. To release the objects from the manipulator, deselect them from the CURRENT OBJECTS menu just after clicking the ADD S/E point button. When all the points of a path have been selected, click the COMPUTE button. This will compute the interpolated joint positions for the path (a cubic spline in this case), and update the number of frames displayed in the frame slider. To view the simulation, click RESET OBJECTS to put the objects back at their staring positions and then click the forward arrow button (>) on the VCR like control panel to view the animated simulation. Additional segments may be added to the end of the path by following the same procedure. At any time while creating a path the user can switch back to the OBJECTS subsystem to reposition objects. When the simulation is performing the desired tasks correctly, a device control file can then be written.

Writing Device Control Code

Click on the FILE I/O button from the main menu, this subsystem will allow the user to read and write status, point, and device control files, as well as writing animation files that can be used with other animation software packages. Click on the WRITE DEVICE CONTROL FILE button; a menu will appear that asks for information about the robot,

type of communication, and robot signature model (to be added later). When the correct parameters have been entered, click the WRITE button, and the device code will be written to a file. For the RV-M1, the device control code will be written out in BASIC. This code can then be transferred to the robot's control computer (by Ethernet or a floppy disk).

Sample Status File

```
# FILE: blocks.olp

CAMERA 1
fovy 350
aspect 1.5
clip 100.0 3000.0
polar 1500.0 900 0 0
lookat 0.0 0.0 150.0

CAMERA 2
fovy 350
aspect 1.5
clip 100.0 3000.0
polar 1500.0 300 -300 0
lookat 0.0 0.0 150.0

CAMERA 3
fovy 350
aspect 1.5
clip 100.0 3000.0
polar 1500.0 900 0 0
lookat 0.0 0.0 150.0

CAMERA 4
fovy 350
aspect 1.5
clip 100.0 3000.0
polar 1500.0 1800 0 0
lookat 0.0 0.0 150.0

CAMERA 5
fovy 350
aspect 1.5
clip 500.0 3500.0
```

```
polar 3000.0 0 0 0
lookat 0.0 0.0 150.0

CAMERA 6
fovy 350
aspect 1.5
clip 500.0 3500.0
polar 3000.0 0 0 0
lookat 0.0 0.0 0.0

LIGHT 1
on
.8 .8 .8 .1 .1 .1 0. 1. 1. 0.

LIGHT 2
on
.8 .8 .8 .1 .1 .1 0. -1. 1. 0.

LIGHT 3
on
.5 .5 .5 .1 .1 .1 -1. 0. 1. 0.

LIGHT 4
on
.5 .5 .5 .1 .1 .1 -1. 0. 1. 0.

MATERIALS
1 .1 .1 .1 .0 .4 .2 .5 .5 .5 10
2 .1 .1 .1 .0 .2 .4 .5 .5 .5 10
3 .1 .1 .1 .2 .4 .0 .5 .5 .5 10
4 .1 .1 .1 .4 .0 .4 .5 .5 .5 10
5 .1 .1 .1 .4 .4 .2 .5 .5 .5 10
6 .1 .1 .1 .2 .4 .4 .5 .5 .5 10
7 .1 .1 .1 .0 .4 .2 .5 .5 .5 10
8 .1 .1 .1 .8 .4 .4 .5 .5 .5 10
9 .1 .1 .1 .0 .0 .8 .5 .5 .5 10
10 .1 .1 .1 .8 .8 .8 .5 .5 .5 10
11 .1 .1 .1 .2 .6 .5 .5 .5 .5 10
```

```
12 .1 .1 .1 .3 .5 .6 .5 .5 .5 10
13 .1 .1 .1 .4 .4 .7 .5 .5 .5 10
14 .1 .1 .1 .5 .3 .1 .5 .5 .5 10
15 .1 .1 .1 .6 .2 .2 .5 .5 .5 10
16 .1 .1 .1 .7 .1 .3 .5 .5 .5 10

BGCOLOR

ROBOT MODELS
1 RVM1

ROBOT GEOMETRY
1 geom/rvm1sf.byu

OBJECT GEOMETRY
geom/blocks.byu

ROBOT POSITIONS
1 BS 0.0 0.0 0.0 90.0 0.0 0.0
1 TW 0.0 0.0 179.0 0.0 0.0 180.0

OBJECT POSITIONS
1 350.0 0.0 0.0 0.0 0.0 0.0
2 350.0 80.0 0.0 0.0 0.0 0.0
3 350.0 160.0 0.0 0.0 0.0 0.0
4 0.0 350.0 0.0 0.0 0.0 0.0
5 80.0 350.0 0.0 0.0 0.0 0.0
6 160.0 350.0 0.0 0.0 0.0 0.0

ROBOT ATTRIBUTES
1 smooth
2 smooth
3 smooth
4 smooth
5 smooth
6 smooth
7 flat
8 flat
```

```
9 flat
10 flat

OBJECT ATTRIBUTES
1 flat
2 flat
3 flat
4 flat
5 flat
6 flat
```

Sample Points File

```
# POINTS FILE: blocks.pts

JOINT ANGLES
1 1 0 9.0 1.0 0.0 0.0 0.0 90.0 0.0 0.0 0.0
2 2 0 9.0 1.0 -90.0 24.4 -64.5 40.2 0.0 0.0 60.0
3 3 0 9.0 1.0 -90.0 7.4 -57.8 50.4 0.0 0.0 60.0
4 4 0 9.0 1.0 -90.0 7.4 -57.8 50.4 0.0 0.0 40.0
5 5 0 9.0 1.0 -90.0 24.2 -64.5 40.3 0.0 0.0 40.0
6 6 0 9.0 1.0 -6.5 23.8 -63.3 39.5 -6.5 0.0 40.0
7 7 0 9.0 1.0 -6.5 14.9 -61.2 46.2 -6.5 0.0 40.0
8 8 0 9.0 1.0 -6.5 14.9 -61.2 46.2 -6.5 0.0 60.0
9 9 0 9.0 1.0 -6.5 23.8 -63.3 39.5 -6.5 0.0 60.0
10 10 0 9.0 1.0 -77.1 22.4 -59.3 37.0 12.9 0.0 60.0
11 11 0 9.0 1.0 -77.1 5.7 -52.2 46.4 12.9 0.0 60.0
12 12 0 9.0 1.0 -77.1 5.7 -52.2 46.4 12.9 0.0 40.0
13 13 0 9.0 1.0 -77.1 22.4 -59.3 37.0 12.9 0.0 40.0
14 14 0 9.0 1.0 -18.9 19.9 -52.4 32.5 -18.9 0.0 40.0
15 15 0 9.0 1.0 -18.9 11.3 -50.0 38.7 -18.9 0.0 40.0
16 16 0 9.0 1.0 -18.9 11.3 -50.0 38.7 -18.9 0.0 60.0
17 17 0 9.0 1.0 -18.9 19.9 -52.4 32.5 -18.9 0.0 60.0
18 18 0 9.0 1.0 -65.4 15.8 -41.4 25.6 24.6 0.0 60.0
19 19 0 9.0 1.0 -65.4 -1.1 -31.4 32.5 24.6 0.0 60.0
20 20 0 9.0 1.0 -65.4 -1.1 -31.4 32.5 24.6 0.0 40.0
21 21 0 9.0 1.0 -65.4 15.8 -41.4 25.6 24.6 0.0 40.0
22 22 0 9.0 1.0 -12.9 28.3 -58.1 29.8 -12.9 0.0 40.0
23 23 0 9.0 1.0 -12.9 20.7 -59.2 38.5 -12.9 0.0 40.0
24 24 0 9.0 1.0 -12.9 20.7 -59.2 38.5 -12.9 0.0 60.0
25 25 0 9.0 1.0 -12.9 28.3 -58.1 29.8 -12.9 0.0 60.0
26 26 0 9.0 1.0 0.0 0.0 0.0 90.0 0.0 0.0 0.0

OBJECT STATUS
1 1 0 0 0 0 0 0 0
2 2 0 0 0 0 0 0 0
3 3 0 0 0 0 0 0 0
```

4	4	0	1	0	0	0	0	0	
5	5	0	1	0	0	0	0	0	
6	6	0	1	0	0	0	0	0	
7	7	0	1	0	0	0	0	0	
8	8	0	0	0	0	0	0	0	
9	9	0	0	0	0	0	0	0	
10	10	0	0	0	0	0	0	0	0
11	11	0	0	0	0	0	0	0	0
12	12	0	0	1	0	0	0	0	
13	13	0	0	1	0	0	0	0	
14	14	0	0	1	0	0	0	0	
15	15	0	0	1	0	0	0	0	
16	16	0	0	0	0	0	0	0	
17	17	0	0	0	0	0	0	0	
18	18	0	0	0	0	0	0	0	
19	19	0	0	0	0	0	0	0	
20	20	0	0	0	1	0	0	0	
21	21	0	0	0	1	0	0	0	
22	22	0	0	0	1	0	0	0	
23	23	0	0	0	1	0	0	0	
24	24	0	0	0	0	0	0	0	
25	25	0	0	0	0	0	0	0	
26	26	0	0	0	0	0	0	0	

Sample Device Control File

```
OPEN "COM1:9600,E,7,2,CS5000,DS5000" FOR RANDOM AS #1
PRINT #1, "PD 1, 0.0, 589.0, 300.0, 0.0, 0.0"
PRINT #1, "PD 2, 350.4, 0.0, 121.2, -89.9, 0.0"
PRINT #1, "PD 3, 349.9, 0.0, 29.9, -90.0, 0.0"
PRINT #1, "PD 4, 349.9, 0.0, 29.9, -90.0, 0.0"
PRINT #1, "PD 5, 350.1, 0.0, 120.0, -90.0, 0.0"
PRINT #1, "PD 6, 39.9, 349.9, 120.1, -90.0, 6.5"
PRINT #1, "PD 7, 39.8, 349.6, 69.6, -90.1, 6.5"
PRINT #1, "PD 8, 39.8, 349.6, 69.6, -90.1, 6.5"
PRINT #1, "PD 9, 39.9, 349.9, 120.1, -90.0, 6.5"
PRINT #1, "PD 10, 350.3, 80.2, 120.2, -89.9, -12.9"
PRINT #1, "PD 11, 349.5, 80.1, 29.8, -90.1, -12.9"
PRINT #1, "PD 12, 349.5, 80.1, 29.8, -90.1, -12.9"
PRINT #1, "PD 13, 350.3, 80.2, 120.2, -89.9, -12.9"
PRINT #1, "PD 14, 119.9, 350.1, 120.1, -90.0, 18.9"
PRINT #1, "PD 15, 119.9, 350.1, 69.9, -90.0, 18.9"
PRINT #1, "PD 16, 119.9, 350.1, 69.9, -90.0, 18.9"
PRINT #1, "PD 17, 119.9, 350.1, 120.1, -90.0, 18.9"
PRINT #1, "PD 18, 349.9, 160.2, 119.9, -90.0, -24.6"
PRINT #1, "PD 19, 350.0, 160.2, 30.2, -90.0, -24.6"
PRINT #1, "PD 20, 350.0, 160.2, 30.2, -90.0, -24.6"
PRINT #1, "PD 21, 349.9, 160.2, 119.9, -90.0, -24.6"
PRINT #1, "PD 22, 80.1, 349.9, 160.0, -90.0, 12.9"
PRINT #1, "PD 23, 80.2, 350.0, 109.8, -90.0, 12.9"
PRINT #1, "PD 24, 80.2, 350.0, 109.8, -90.0, 12.9"
PRINT #1, "PD 25, 80.1, 349.9, 160.0, -90.0, 12.9"
PRINT #1, "PD 26, 0.0, 589.0, 300.0, 0.0, 0.0"
PRINT #1, "SP 9, H"
PRINT #1, "MO 1, C"
PRINT #1, "MO 2, O"
PRINT #1, "MO 3, O"
PRINT #1, "MO 4, C"
PRINT #1, "MO 5, C"
PRINT #1, "MO 6, C"
```

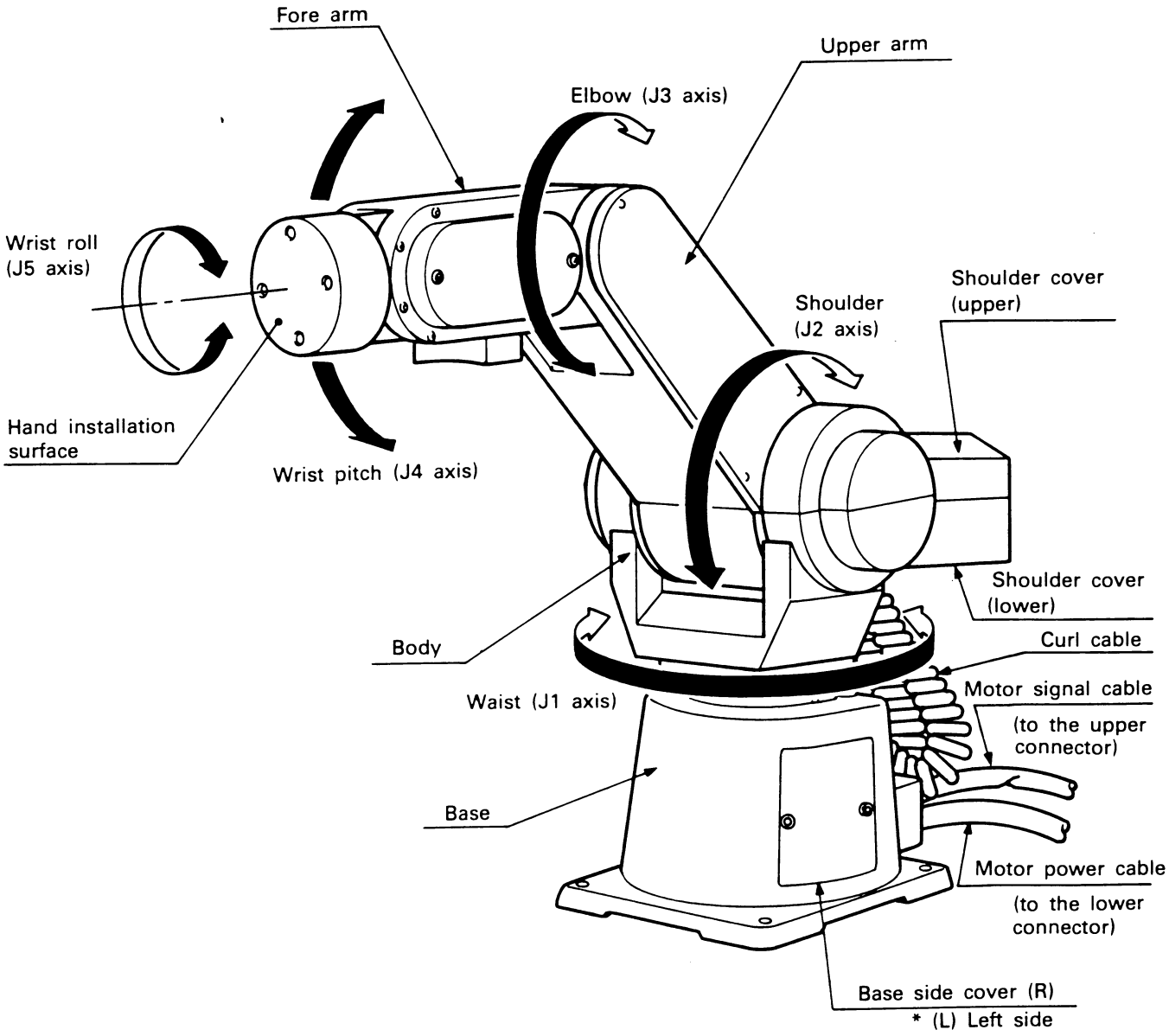
```
PRINT #1, "MO 7, C"
PRINT #1, "MO 8, O"
PRINT #1, "MO 9, O"
PRINT #1, "MO 10, O"
PRINT #1, "MO 11, O"
PRINT #1, "MO 12, C"
PRINT #1, "MO 13, C"
PRINT #1, "MO 14, C"
PRINT #1, "MO 15, C"
PRINT #1, "MO 16, O"
PRINT #1, "MO 17, O"
PRINT #1, "MO 18, O"
PRINT #1, "MO 19, O"
PRINT #1, "MO 20, C"
PRINT #1, "MO 21, C"
PRINT #1, "MO 22, C"
PRINT #1, "MO 23, C"
PRINT #1, "MO 24, O"
PRINT #1, "MO 25, O"
PRINT #1, "MO 26, C"
END
```

APPENDIX B

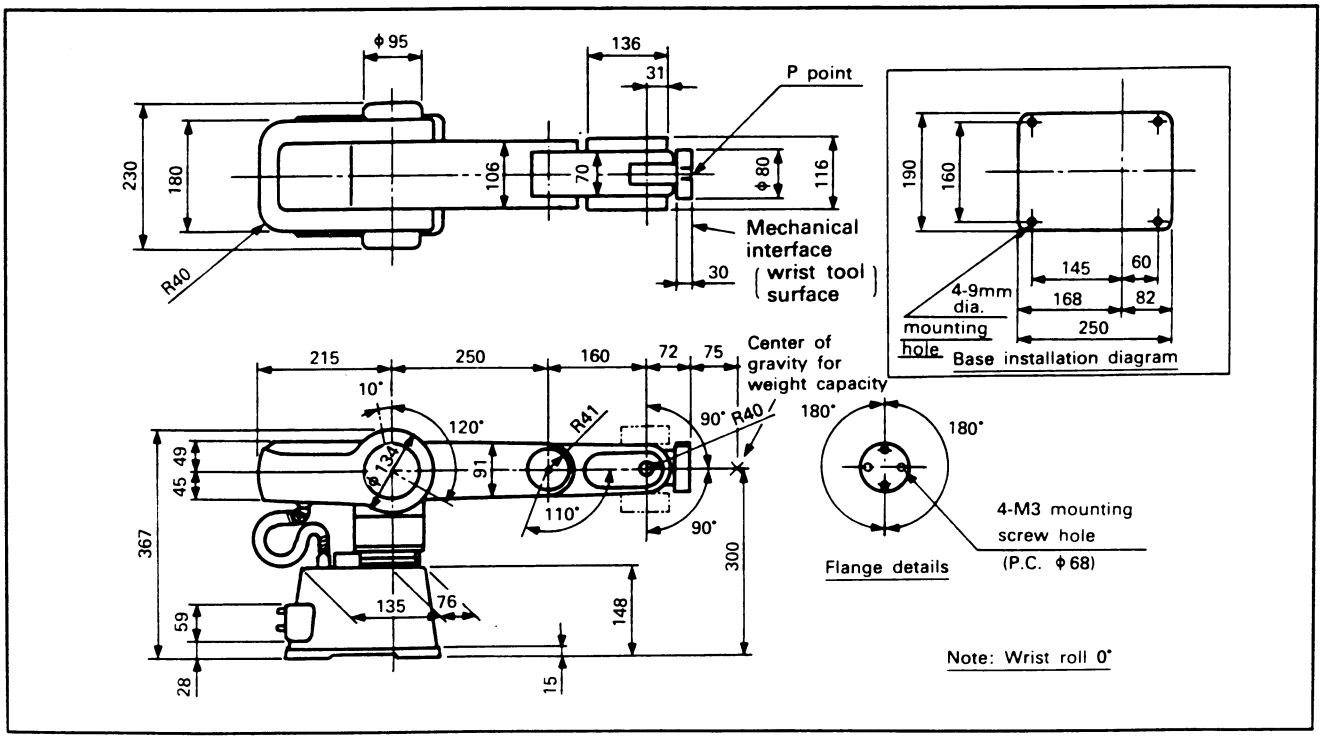
ROBOT SPECIFICATIONS

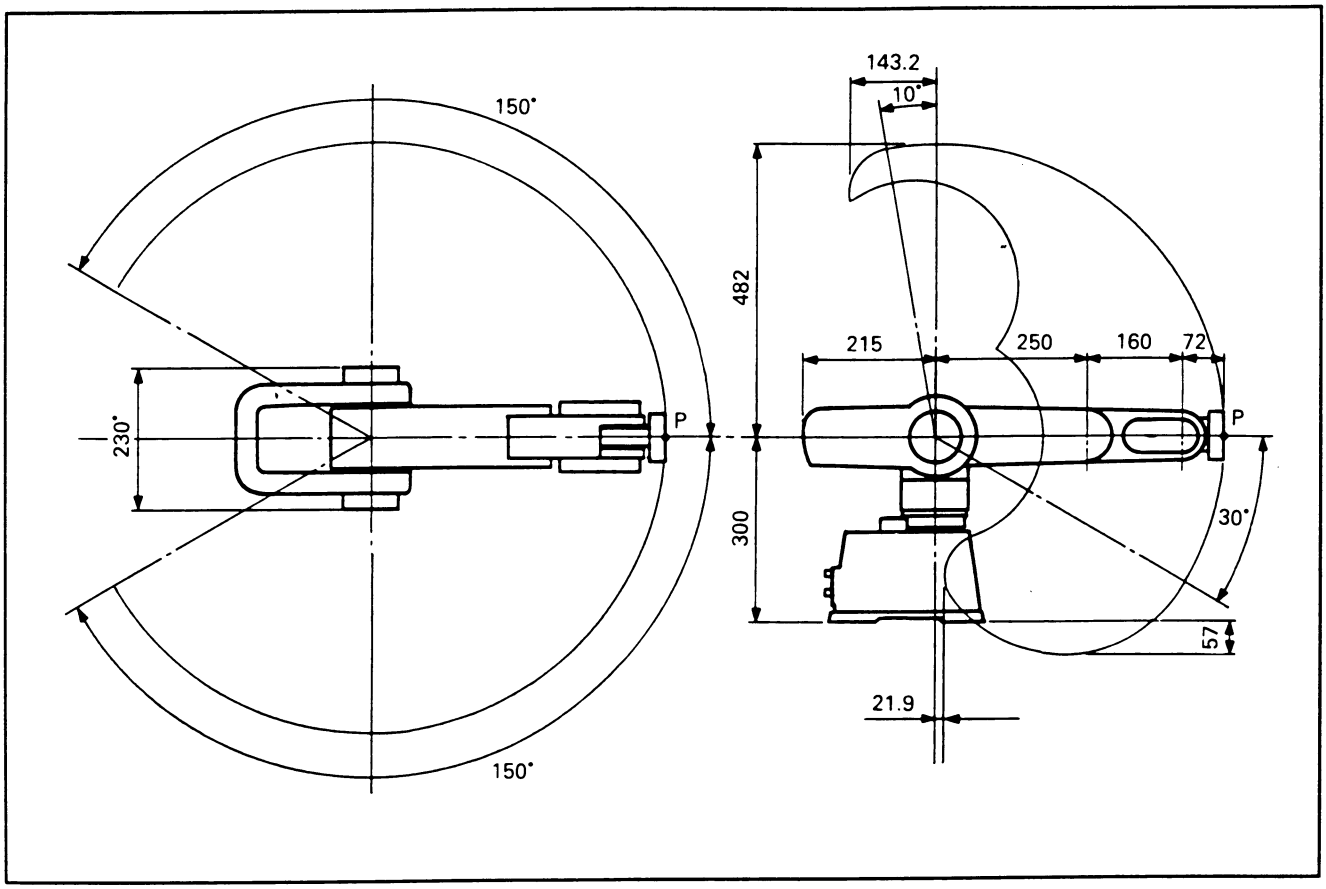
Mitsubishi RV-M1 Specifications

Robot Specifications



Item		Specifications	Remarks
Mechanical Structure		5 degrees of freedom, vertical articulated robot	
Operation range	Waist rotation	300° (max. 120°/sec)	J1 axis
	Shoulder rotation	130° (max. 72°/sec)	J2 axis
	Elbow rotation	110° (max. 109°/sec)	J3 axis
	Wrist pitch	±90° (max. 100°/sec)	J4 axis
	Wrist roll	±180° (max. 163°/sec)	J5 axis
Arm length	Upper arm	250mm	
	Fore arm	160mm	
Weight capacity		Max. 1.2kgf (including the hand weight)	75mm from the mechanical interface (center of gravity)
Maximum path velocity		1000mm/sec (wrist tool surface)	Speed at point P in Fig. 1.3.4
Position repeatability		0.3mm (roll center of the wrist tool surface)	Accuracy at point P in Fig. 1.3.4
Drive system		Electrical servo drive using DC servo motors	
Robot weight Motor capacity		Approx. 19kgf J1 to J3 axes: 30W; J4, J5 axes: 11W	




150°
230°
P
150°
143.2
10°
482
215
250
160
72
P
300
30°
57
21.9

Manipulator Specifications

Item	Specifications	Remarks
Type	HM-01	The holding power can be set in 16 steps.
Drive system	DC servo motor drive	
Opening/closing stroke	0 to 60mm	
Grip power	Max. 3.5kgf	
Ambient temperature	5 to 40°C	
Service life	More than 300,000 times	
Weight	600gf